Proceedings of the 3rd Workshop on Scalable Model Driven Engineering

July 23, 2015. L'Aquila, Italy

BigMDE 2015

Dimitris Kolovos
Davide Di Ruscio
Nicholas Matragkas
Jesús Sánchez Cuadrado
Istvan Rath
Massimo Tisi

Copyright © 2015 for the individual papers by the papers' authors. Copying permitted only for private and academic purposes. This volume is published and copyrighted by its editors.

Preface

As Model Driven Engineering (MDE) is increasingly applied to larger and more complex systems, the current generation of modelling and model management technologies are being pushed to their limits in terms of capacity and efficiency. As such, additional research and development is imperative in order to enable MDE to remain relevant with industrial practice and to continue delivering its widely-recognised productivity, quality, and maintainability benefits.

The third edition of the BigMDE workshop (http://www.big-mde.eu/) has been co-located with the Software Technologies: Applications and Foundations (STAF 2015) conference. BigMDE 2015 provided a forum for developers and users of modelling and model management languages and tools where to discuss different problems and solutions related to scalability aspects of MDE, including

- Working with large models
- Collaborative modelling (version control, collaborative editing)
- Transformation and validation of large models
- Model fragmentation and modularity mechanisms
- Efficient model persistence and retrieval
- Models and model transformations on the cloud
- Visualization techniques for large models

Many people contributed to the success of BigMDE 2015. We would like to truly acknowledge the work of all Program Committee members, and reviewers for the timely delivery of reviews and constructive discussions given the very tight review schedule. Finally, we would like to thank the authors, without whom this workshop would not exist.

July 23, 2015
L'Aquila, Italy

Dimitris Kolovos
Davide Di Ruscio
Nicholas Matragkas
Jesús Sánchez Cuadrado
Istvan Rath
Massimo Tisi

Organisation

Chairs

Dimitris Kolovos	University of York, UK
Davide Di Ruscio	University of L'Aquila, Italy
Nicholas Matragkas	University of Hull, UK
Jesús Sánchez Cuadrado	Universidad Autónoma de Madrid, Spain
Istvan Rath	Budapest University of Technology and Economics, Hungary
Massimo Tisi	Ecole des Mines de Nantes, France

Program Committee

Eugene Syriani	University of Montreal, Canada
Alfonso Pierantonio	University of L'Aquila, Italy
Orlando Avila-Garcia	Open Canarias S.L., Spain
Marcos Didonet Del Fabro	Universidade Federal do Paraná, Brazil
Salvador Trujillo	IKERLAN, Spain
Esther Guerra	Universidad Autónoma de Madrid, Spain
Rubby Casallas	University of los Andes, Colombia
Manuel Wimmer	Vienna University of Technology, Austria
Kevin Lano	King's College London, UK
Loli Burgueño	University of Malaga, Spain
Jeff Gray	University of Alabama, USA
Goetz Botterweck	Lero, University of Limerick, Ireland
Marco Brambilla	Politecnico di Milano, Italy
Seyyed Shah	University of Oxford, UK
Markus Scheidgen	Humboldt-Universitt zu Berlin, Germany
Gerson Sunyé	Université de Nantes, France
Jesus García-Molina	Universidad de Murcia, Spain
Dániel Varrò	Budapest University of Technology and Economics, Hungary
Marko Boger	University of Konstanz, Germany
Tony Clark	University of Middlesex, UK
Harald Störrle	Technical University of Denmark, Denmark
Gabriele Taentzer	Philipps-Universität Marburg, Germany

Table of Contents

Generation of Large Random Models for Benchmarking **1**
Markus Scheidgen . . . .

Approach to Define Highly Scalable Metamodels Based on JSON **11**
Markus Gerhart, Julian Bayer, Jan Moritz Höfner, and Marko Boger . . . . . .

Scalable model exploration through abstraction and fragmentation strategies **21**
Antonio Garmendia, Antonio Jiménez-Pastor, and Juan de Lara . . . .

An Efficient Computation Strategy for allInstances() **32**
Ran Wei and Dimitrios S. Kolovos . . . .

Decentralized Model Persistence for Distributed Computing **42**
Abel Gómez, Amine Benelallam, and Massimo Tisi . . . .

Parallel In-place Model Transformations with LinTra **52**
Loli Burgueño, Javier Troya, Manuel Wimmer, and Antonio Vallecillo . . . . .

Beyond Information Silos Challenges in Integrating Industrial Model-based Data **63**
Ali Shahrokni and Jan Söderberg . . . .

Generation of Large Random Models for Benchmarking

Markus Scheidgen[1]

Humboldt Universität zu Berlin, Department of Computer Science,
Unter den Linden 6, 10099 Berlin, Germany
{scheidge}@informatik.hu-berlin.de

Abstract. Since model driven engineering (MDE) is applied to larger and more complex system, the memory and execution time performance of model processing tools and frameworks has become important. Benchmarks are a valuable tool to evaluate performance and hence assess scalability. But, benchmarks rely on reasonably large models that are unbiased, can be shaped to distinct use-case scenarios, and are "real" enough (e.g. non-uniform) to cause real-world behavior (especially when mechanisms that exploit repetitive patterns like caching, compression, JIT-compilation, etc. are involved). Creating large models is expensive and erroneous, and neither existing models nor uniform synthetic models cover all three of the wanted properties.
In this paper, we use randomness to generate unbiased, non-uniform models. Furthermore, we use distributions and parametrization to shape these models to simulate different use-case scenarios. We present a meta-model-based framework that allows us to describe and create randomly generated models based on a meta-model and a description written in a specifically developed generator DSL. We use a random code generator for an object-oriented programming language as case study and compare our result to non-randomly and synthetically created code, as well as to existing Java-code.

1 Introduction

In traditional model driven software engineering, we are not concerned about how much memory our editors consume or how long it takes to transform a model; models are small and execution is instantaneous. But when models become bigger, their processing requires substantial resources. Up to the point, where we began to conceive technology that is specifically designed to deal with large models. In this context, memory consumption and execution times are of central concern and *BigMDE* technology is valued by its performance. Consequently, *benchmarks* that enable sound comparison of a method's, framework's, or tool's performance are valuable and necessary tools in evaluating our work.

In computer science, we define a *software benchmark* as the measurement of a certain performance property taken in a well defined process under a well defined workload in a well defined environment. Where the benchmark mimics certain application scenarios. In MDSE scenarios, workloads comprise input models and tasks that are performed on these models.

Input models characteristics have an influence on the quality of the benchmark. First, input models need to be unbiased, i.e. they must not deliberately or accidentally ease the processing by one technology and burden another. Secondly, they need to be *real* enough to invoke behavior that is similar to behavior caused by actual input models. Non random synthetic models for example are often overly uniform and can therefore fool compression, caching, or runtime optimization components (often contained in BigMDE technology) into unrealistic behavior. Thirdly, benchmarks mimic different application scenarios. Different scenarios require input models with different *shapes*. Here, the concrete form of possible shapes depends on the meta-model. E.g. shapes can be expressed as sets of model metrics. Fourthly and finally, input models need to scale, i.e. we need to find input models of arbitrary size. Only with scalable input, we can use benchmarks to access the scalability of MDSE technology.

It is common practice to either use existing models (e.g. the infamous Grabats 09 models) or non random synthetically generated models. Where the former yields in realistic, somewhat unbiased, but only given shapes and scale, the later results in arbitrary large, but utterly unrealistic models. Our approach is to use randomness as a tool to achieve both scalability and configurability as well as a certain degree of realism and reduced bias. We present a generator description language and corresponding tools that allows us to describe and perform the generation of random models with parameterizable generator rules.

The paper is structured as follows. We start with a discussion of related work in the next section. After that, we introduce our generator language. The next section, demonstrates our approach with a generator for object-oriented program code models. The paper closes with conclusions and suggestions for further work.

2 Related Work

Benchmarking with Existing Models

The idea to generate large models for benchmarking large model processing technologies is new and to our knowledge there is no work specifically targeting this domain. However, benchmarking of large model processing technologies has been done before; mostly by authors of such technologies. Common practice is the use of a specific set of large existing models. If we look at benchmarking model persistence technology for NoSQL databases, which is our original motivation for this work, the set of the Grabats 09 graph transformation contest example models are exclusivly used by all work known to us [7,1,8,2].

This benchmarking practice has itself established as a quasi standard to evaluate and compare model persistence technology, even though it exhibits all of the previously stated flaws of using existing models for benchmarks. First, the Grabats models aren't exactly large ($<10^7$ objects), at least when compared to the target scale of the benchmarked technology. Secondly, there is no meaningful mathematical relationship between size metrics of the models in the set, e.g. there is no usable ramp-up in model-size. Even though the models show an increasing size, the models have internally different structure, which makes them non-comparable. Some models represent Java

code in full detail, others only cover declarations. This makes it impossible establish a meaningful formal relationship between size-metric and performance measurements. Thirdly, we have a very small set of 4 models and all models are models of the same meta-model. This makes the example set biased (towards reverse-engineered Java code models) and again makes it difficult to establish relationships between metrics and performance. Forthly, the internal structure of the models makes it impossible to import all the models into CDO. At least no publication presented any measurements for CDO and the biggest two of the four Grabats models. This is especially bad, since CDO as most popular SQL-based persistence technology, presents the most reasonable (and only used) baseline.

More concrete benchmarks including the precise definition of tasks exist for model queries and transformations [11]. The Grabats 09 contents actually formulates such benchmarks. In [11,5] the authors define frameworks to precisely define tasks for model transformation and query benchmarks and provide the means to evaluate the created benchmarks in that domain.

Model Generation for Test

Before benchmarking became an issue for MDSE, models were generated to provide test subjects for MDSE technology. We can identify three distinct approaches.

SAT-solver For most test scenarios, not only syntactically but also static semantically correct models are needed. Brottier et al. [3] and Sen et al. [9] propose the use of SAT-solvers to derive meta-model instances from the meta-model and it's static semantic constraint. Meta-model and constraints are translated into logical formula, solutions that satisfy these are translated back to corresponding meta-model instances. Since each solution "solves" the meta-model and it's constraints, the solution represents a semantically correct instance of the given meta-model. The non-polynomial complexity of SAT problems and the consequently involved heuristics do not scale well.

Existing graph generators Mougenot et al. [6] use existing graph generators and map nodes and edges of generated graphs to meta-model classes and associations. While this approach scales well, it does not allow to shape the generated model. The used algorithms provide uniform looking random graphs and result in uniform models. In reality, most models cover different aspects of a system with completely different structural properties. E.g. the graph that represents package,class,method declarations has different properties/metrics as a graph that describes the internals of a method implementation.

Constructive formalisms Models can be generated with constructive formalisms like formal grammars or graph grammars. Ehrig et al. [4] propose to use graph grammars and random application of appropriate graph grammar rules to generate models. Our own approach (which is very similar to context-free grammars) fits this category as well. In comparison it lacks the formal properties of the graph grammar approach and is limited to context-free constructs (e.g. no static semantic constraints), but scales better due to the simpler formalism. In practice graph-grammars introduce

further restrictions, since graph grammars become inherently complex, especially if one tries to map static semantic constraints into the rules.

3 Generator Language

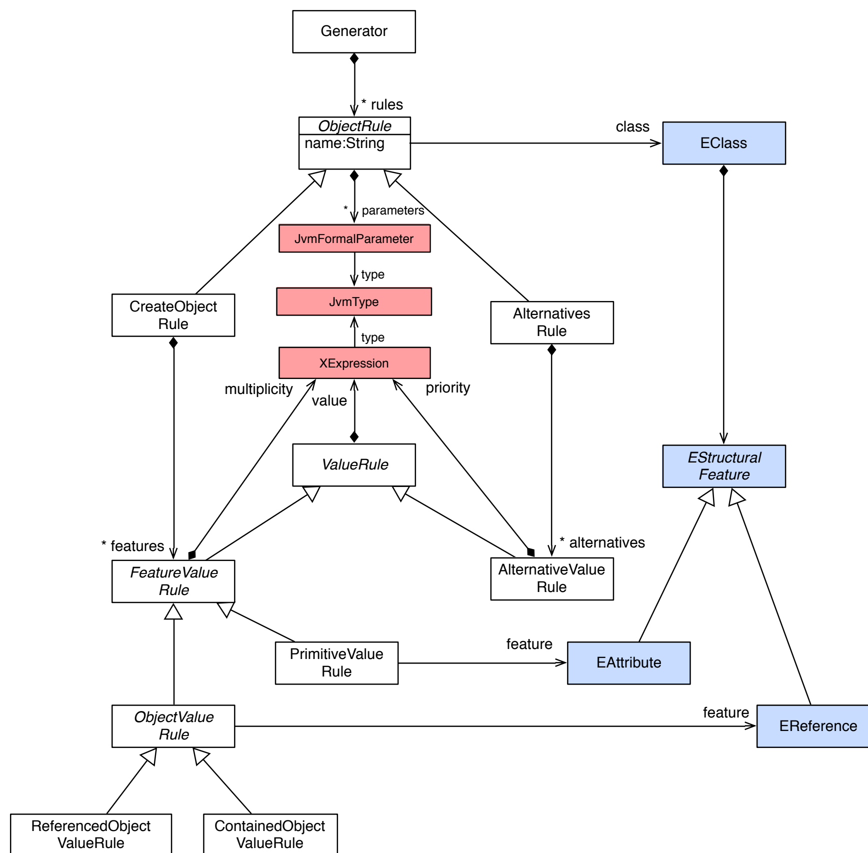


Fig. 1: Meta-model of the proposed model generators description language.

We developed the model generator description language *rcore*[1] as an external domain specific language based on EMF, xText, and xBase with the consequent eclipse-based tool support. Fig. 1 depicts the underlying meta-model of our generator language. *RCore* follows a declarative approach and uses production rules to describe

[1] The source code and examples can be obtained here: http://github.com/markus1978/RandomEMF

possible models. Similar to a formal grammar that can be used to generate strings over an alphabet, we generate models over a meta-model. In contrast to grammars, *rcore* rules govern rule application via expressions that present concrete choices (i.e. concrete multiplicities or chosen alternatives). These expressions can either use fix-values (e.g. to generate synthetic models) or call random number generators following different distribution functions (i.e. to generate random models). Build in variables (e.g. the generated model in progress, or depth of rule application) and custom rule parameters can also be used within expressions.

Generally, we distinguish between `ObjectRule`s that can generate model elements (i.e. EMF objects) and `FeatureValueRule`s that assign values to an object's structural features (i.e. EMF attributes and references). Cascading application of `ObjectRule-FeatureValueRule-ObjectRule-Fea...` allows clients to generate containment hierarchies, i.e. the spine of each EMF model.

Each *rcore* description comprises an instance of `Generator` and is associated with an ecore-package that represents the meta-model that this generator is written for. Each `Generator` consist of a set of `ObjectRule`s, where the first `ObjectRule` is the start rule. Each of these `ObjectRule`s is associated with a `EClass` that determines the meta-type of the objects generated by this rule. There are two concrete types of `ObjectRule`s: `CreateObjectRule`s that describe the direct creation of objects (i.e. meta-class instances, a.k.a model-elements) and `AternativesRule`s that can be used to randomly refer object creation to several alternative rules. Further, `ObjectRule`s can have parameters.

While `ObjectRule`s are used to describe object generation, `ValueRule`s are used to determine values that can be used within `ObjectRule`s. `ValueRule`s determine concrete values via an xBase `XExpression` that can evaluate to a primitive value (e.g. to be assigned to an attribute, `PrimitiveValueRule`), or that calls an `ObjectRule` (e.g. to create a value for a containment reference, `ContainedObjectValueRule`), or that queries the generated model for a reference target (e.g. to be assigned to a non-containment reference, `ReferencedObjectValue`), or that refers object creation to an `ObjectRule` (e.g. to create an alternative for an `AlternativesRule`, `AlternativeValueRule`).

Concrete `FeatureValueRule`s are associated with a `EStructuralFeature` and are used to assign values to the according feature of the object created by the containing `CreateObjectRule`. Each `FeatureValueRule` also has an expression that determines the multiplicity of the feature, i.e. that determines how many values are assigned to the feature, i.e. how many times the value expression is evaluated. `ObjectValueRule`s are associated with `EReference` and are used to assign values to references, and `PrimitiveValueRules` are associated with `EAttribute` and are used to assign value to attributes.

`AlternativeValueRule`s have an additional expression that determines the priority of the alternative within the containing `AlternativesRule`. When applied the `AlternativesRule` will uniformly choose an alternative with priorities as weights. Only the chosen alternative is evaluated to provide an object.

Further static semantic constraints have to be fulfilled for a correct *rcore* description. (1) the value expressions of `AlternativeValueRules` must have compatible

type with the `EClass` associated with the containing `AlternativesRule`. The types of all value expressions in `FeatureValueRule`s must be compatible with the associated structural feature's type. The associated features of `FeatureValueRule`'s must be features of the `EClass` associated with the containing `CreateObjectRule`. All used `EClass`es must be contained in the meta-model that is associated with the `Generator`.

```
1.  package de.hub.rcore.example

3.  import org.eclipse.emf.ecore.EDataType
4.  import org.eclipse.emf.ecore.EcorePackage
5.  import static de.hub.randomemf.runtime.Random.*

7.  generator RandomEcore for ecore in "platform:/resource/org.eclipse.emf.ecore/model/Ecore.ecore" {
8.    Package: EPackage ->
9.      name := LatinCamel(Normal(3,2)).toLowerCase
10.     nsPrefix := RandomID(Normal(2.5,1))
11.     nsURI := "http://hub.de/rcore/examples/" + self.name
12.     eClassifiers += Class#NegBinomial(5,0.5);
13.
14.   Class: EClass ->
15.     name := LatinCamel(Normal(4,2))
16.     abstract := UniformBool(0.2)
17.     eStructuralFeatures += Feature#NegBinomial(2,0.5);
18.
19.   alter Feature: EStructuralFeature ->
20.     Reference(true) | Reference(false) | Attribute#2;
21.
22.   Reference(boolean composite):EReference ->
23.     name := LatinCamel(Normal(3,1)).toFirstLower
24.     upperBound := if (UniformBool(0.5)) -1 else 1
25.     ordered := UniformBool(0.2)
26.     containment := composite
27.     eType:EClass := Uniform(model.EClassifiers.filter[it instanceof org.eclipse.emf.ecore.EClass]);
28.
29.   Attribute:EAttribute ->
30.     name := LatinCamel(Normal(3,1)).toFirstLower
31.     upperBound := if (UniformBool(0.1)) -1 else 1
32.     eType:EDataType := Uniform(EcorePackage.eINSTANCE.EClassifiers.filter[it instanceof DataType]);
33. }
```

Fig. 2: Example generator description for Ecore models.

Fig. 2 presents an example *rcore* description. The described generator produces randomly generated Ecore models. Some of the expressions use predefined random number generator based method to create random ids, strings, numbers (based on different distributions such as normal, neg. binomial, etc.)). The method `Uniform(List<EObject>)` for example, is used to uniformly draw reference targets from a collection of given possible target objects. Another example is `LatinCamel(int)` that generates a camel case identifier with the given number of syllables. The two methods `Normal(double,double)` and `NegBinomial(double,double)` are exam-

ples for the use of random number generators to create normal and negative binomial distributed random numbers.

4 Example Results

```
1.  package dabobobues;

3.  class Dues {
4.
5.    DuBoBuTus begubicus;
6.    ELius brauguslus;
7.
8.    void Dues(Alius donus, FanulAudaCio aubetin) {
9.    }
10.
11.   void baGusFritus() {
12.     eudaguslius = "";
13.     bigusdaGubolius();
14.     if ("") {
15.       annulAugusaugusfrigustin("");
16.       albucio = Dues()<=++12;
17.       bi();
18.       eBoTor();
19.     } else {
20.       brauguslus = 9;
21.       baGusFritus();
22.       duLus = ""=="";
23.     }
24.   }
25.
26.   void aufribonulAubufrinus(Dues e) {
27.      dobubogutor();
28.      aubiguTus = 9;
29.   }
30. }
```

Fig. 3: Example of randomly generated code for an object-oriented programming language.

In this section, we want to demonstrate the use *rcore* for meta-models that are more complex than the *Ecore* example in the previous section. Based on their popularity, overall complexity, and well understood properties, we chose an object-oriented Java-like programming language with packages, classes, fields, methods, statements, expressions (incl. operators and literals). We omitted details that are either repetitive (template parameters, interfaces, anonymous classes) or not interesting from a randomized generation point of view (most flags and enums like abstract, overrides, visibility, etc.). We developed the model generator based on a ecore meta-model for the described example language, but we use a simple code generator to pretty print the generated models in a Java-like syntax for better human comprehension. Fig. 3 depicts a sample of the generated models in the serialized form.

Similar to our *Ecore* generator in Fig. 2, we use probability distributions to produce randomized models that exhibit certain characteristics such as average number

of methods per class and similar metrics. We compare the generated results to non synthetic instances of the same meta-model and to program code models of a real-life Java project. We look at two aspects: first the overall containment hierarchy and secondly the use of non-containment references exemplified by method calls (i.e. references between method call and respective method declarations).

Containment Hierarchies

Fig. 4 shows a sunburst chart representation of three packages of object-oriented code. One is synthetically generated, one is actual Java code taken from our EMF-fragments project, and one is randomly generated. The synthetically generated program comprises a fixed number of classes, each of which contains a fixed number of inner classes, fixed number of methods, etc. Multiplicity and choice of contained elements are determined by constants or values in a repetitive pattern. The result is a homogeneous repetitive non random structure. The actual Java code shows that containment in real code is varying: there are classes with more or less methods, methods can be short or longer. Tamai et al [10] suggest that multiplicities of containment references can be modeled with negative binomial distributed random variables. Our generator for random code uses such negative binomial distributed random variables. We chose parameters that yield in expected values that represent reasonable metrics for classes per package, methods per class, statements per methods, depth of expressions, etc. We chose metrics that resemble the empirically determined corresponding metric in the reference Java code of the prior example.

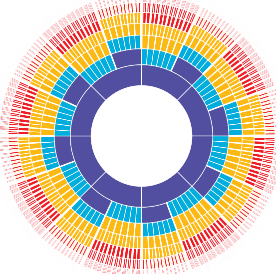
References and Dependencies

Fig. 5 show chord chart representations of method calls in three different object-oriented programs. In this chart each line represents a method call and the line end points represent calling and called method. Methods are clustered in blocks that represent the classes that contain them. The first program was randomly generated. We chose the called method for each method call uniformly from the set of all methods within the program. Hence, each class has a similar relative number of expected dependencies to each other class, including the calling class. The code taken from an actual program (again, the same EMF-fragments code as before) shows a different distribution. Calls of methods within the containing class are more likely and furthermore calls of methods within in certain other depending classes are more likely than calls of methods of other less depending classes. The last chart shows generated code gain. Now, we changed our generator to emulate the reference distribution of actual Java code. We do not chose methods uniformly from all methods, but put higher probability on methods of the calling class and of depending classes. We pseudo randomly chose pairs of depending classes based on the similarity of the hash code of their corresponding EMF objects.

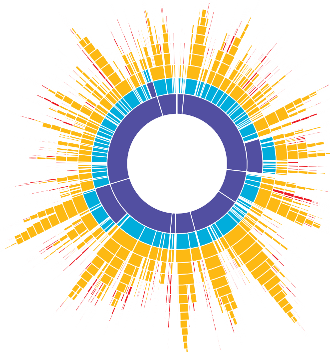
5 Conclusions

We presented the domain specific language *rcore* and a corresponding compiler that aids clients in the development of generators that automatically create instances of

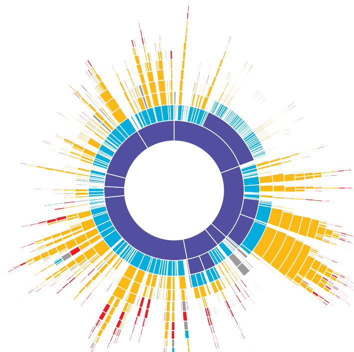
Classes/Interfaces
Methods
Statements
Expressions
others



syntheticly generated code

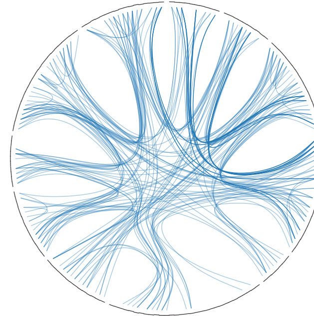


randomly generated code

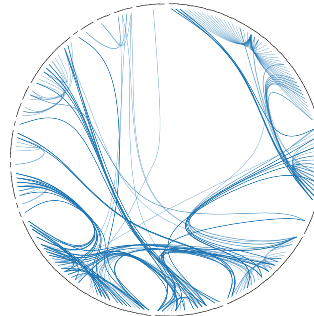


actual Java code

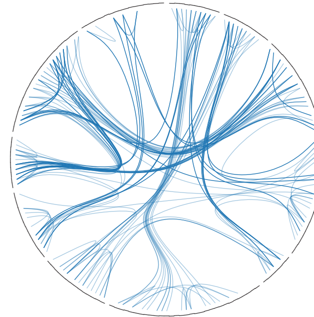
Fig. 4: Sunburst representations of containment hierarchies of generated and actual object-oriented code.



randomly generated code with
uniformly choosen called methods



actual java code



generated random code with
higher chance that the calling method or
methods from classes with similar hash code
are called

Fig. 5: Chord chart visualization of calling-called-method dependencies in generated and actual code.

given Ecore meta-models based on a set of generator rules. These parameterizable rules can be used to control the generation of model elements with random variables of certain probabilistic distributions. This gives clients randomness and configurability as means in their efforts to create instances of their ecore meta-models that are potentially unbiased, have a certain *shape*, and yet mimic real world models. Furthermore, *rcore* allows to describe arbitrary large models with a small set of generator rules. *Rcore*'s simple operational semantics of one executed element generating rule calling other element generating rule should lead to linear time complexity in the number of generated elements, unless the user defined functions used to determine the concrete feature values for the generated elements introduce higher complexities. Therefore, the actual amount of achieved bias, real world mimicry, configurability, and scalability depends on concrete generator descriptions. As future work, we need to create a larger set of case study generators and evaluate these generators for the proposed characteristics.

References

1. Barmpis, K., Kolovos, D.S.: Comparative analysis of data persistence technologies for large-scale models. In: Proceedings of the 2012 Extreme Modeling Workshop. pp. 33–38. ACM (2012)
2. Benelallam, A., Gómez, A., Sunyé, G., Tisi, M., Launay, D.: Neo4EMF, a scalable persistence layer for EMF models. In: Modelling Foundations and Applications, pp. 230–241. Springer (2014)
3. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Le Traon, Y.: Metamodel-based test generation for model transformations: an algorithm and a tool. In: Software Reliability Engineering, 2006. ISSRE'06. 17th International Symposium on. pp. 85–94. IEEE (2006)
4. Ehrig, K., Küster, J.M., Taentzer, G.: Generating instance models from meta models. Software & Systems Modeling 8(4), 479–500 (2009)
5. Izso, B., Szatmari, Z., Bergmann, G., Horvath, A., Rath, I.: Towards precise metrics for predicting graph query performance. 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings pp. 421–431 (2013)
6. Mougenot, A., Darrasse, A., Blanc, X., Soria, M.: Uniform random generation of huge metamodel instances. In: Model Driven Architecture-Foundations and Applications. pp. 130–145. Springer (2009)
7. Pagán, J.E., Cuadrado, J.S., Molina, J.G.: Morsa: a scalable approach for persisting and accessing large models. In: Proceedings of the 14th international conference on Model driven engineering languages and systems. pp. 77–92. Springer-Verlag (2011)
8. Scheidgen, M., Zubow, A., Fischer, J., Kolbe, T.H.: Automated and transparent model fragmentation for persisting large models. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). vol. 7590 LNCS, pp. 102–118 (2012)
9. Sen, S., Baudry, B., Mottu, J.M.: Automatic model generation strategies for model transformation testing. In: Theory and Practice of Model Transformations, pp. 148–164. Springer (2009)
10. Tamai, T., Nakatani, T.: Analysis of software evolution processes using statistical distribution Models. Proceedings of the international workshop on Principles of software evolution - IWPSE '02 p. 120 (2002), http://portal.acm.org/citation.cfm?doid=512035.512063
11. Varró, G., Schürr, A., Varró, D.: Benchmarking for graph transformation. Proceedings - 2005 IEEE Symposium on Visual Languages and Human-Centric Computing 2005, 79–88 (2005)

Approach to Define Highly Scalable Metamodels Based on JSON

Markus Gerhart, Julian Bayer, Jan Moritz Höfner, and Marko Boger

University of Applied Sciences Konstanz, ProGraMof,
Brauneggerstraße 55, 78462 Konstanz, Germany
{mgerhart,jubayer,jahoefne,marko.boger}@htwg-konstanz.de
http://www.htwg-konstanz.de

Abstract. Domain-specific modelling is increasingly adopted in the software development industry. While open source metamodels like Ecore have a wide impact, they still have some problems. The independent storage of nodes (classes) and edges (references) is currently only possible with complex, specific solutions. Furthermore the developed models are stored in the extensible markup language (XML) data format, which leads to problems with large models in terms of scaling.
In this paper we describe an approach that solves the problem of independent classes and references in metamodels and we store the models in the JavaScript Object Notation (JSON) data format to support high scalability. First results of our tests show that the developed approach works and classes and references can be defined independently. In addition, our approach reduces the amount of characters per model by a factor of approximately two compared to Ecore. The entire project is made available as open source under the name MoDiGen. This paper focuses on the description of the metamodel definition in terms of scaling.

Keywords: Metamodel definition, JavaScript Object Notation, JSON, Model scalability, Metamodel scalability, Model storage

1 Introduction

Tools for creating **D**omain-**S**pecific **M**odeling **L**anguages (DSML) are becoming more accepted in the software development industry to develop specific solutions for specific problems. These solutions are developed with tools such as Xtext[5], **M**eta **P**rogramming **S**ystem (MPS)[6], MetaEdit+ Modeler[7], Kybele[8], MagicDraw[9] or Eugenia[10]. The underlying metamodel of the tools is of crucial importance, because it serves as basis for all subsequent steps like code generation or programatic manipulation of the model data. Therefore, access to the metamodel has to be very simple and the memory consumption should be as low as possible.

Only if the metamodel is maintained at a high abstraction level, the subsequent programmatic processing can be implemented simple, clean, and clear like it is suggested by the KISS-principle ("Keep it simple, stupid"). However, existing open source tools cover only the needs of specific subject areas such as the

software development industry. Requirements of very complex metamodels for instance the statics of buildings can currently only be fulfilled through complex detours. Not considering commercial solutions, because they do not give insight into the structure of their metamodel and storage solution, another common problem is the storage consumption of very large models. Furthermore, a great weakness of existing open source solutions such as Ecore[1] is, that they do not scale well to very large models.

Our suggested approach allows the definition of metamodels in a simple and clear way. We offer the possibility, that nodes (classes) and edges (references) can exist independently and with equal rights. This leads to a variety of possibilities in the creation of metamodels. In addition, the data of metamodels and models is held in the **J**ava**S**cript **O**bject **N**otation (JSON)[17] file format. This allows the smooth scaling of the model data using existing solutions such as CouchDB[14], MongoDB[15], and RavenDB[16].

We demonstrate, that the storage of metamodels and models is possible without problems, even when involving well over 10,000 elements (classes and references).

The paper first reviews related work in the field in section 2, which is mostly other tools and techniques for the definition of meta models. Our general approach for the definition of meta-models based on JSON for high scalability is described in section 3. The core contribution of this publication is the developed metamodel with independent classes and connections and the data structure using JSON to store models for high scalability. Section 4 illustrates the results of our approach from different angles. Finally, we summarise the limitations of our research and draw conclusions in section 5.

2 Related Work

The Ecore metamodel of the **E**clipse **M**odeling **F**ramework (EMF) stores edges as parts of nodes. An EReference actually models one end of an edge [1]. This is also true for the **G**eneric **M**odeling **E**nvironment (GME) [2] and WebGME [3]. The disadvantages of this approach were already discussed in [11]. Storing edges as parts of nodes does not scale well for large numbers of edges. Accessing, loading and saving individual edges requires linear time and may pose a problem in terms of heap memory. As an alternative, [11] proposes storing edges as relations like a relational database, separating the edge from the node. That is the same basic principle we follow. Instead of an SQL-like structure as proposed in [11], we introduce the edge as a first-level-object that is stored in a NoSQL database.

The Ecore metamodel uses an E**x**tensible **M**arkup **L**anguage (XML) based format for model serialization [1]. The need for a model serialization format that is not based on XML was formulated in [12]. While our approach does not satisfy all the criteria for model storage set forth in that paper, we believe that JSON as the basic format for model storage will simplify implementing these criteria. A diagram is stored as a collection of JSON objects which can be addressed

through an identifier. Instead of loading large models all at once, objects could be loaded as needed using their identifier. The partial loading of model data based on JSON is covered by databases like CouchDB or MongoDB. An XML-based format would always have to be loaded in full apart from approaches such as partial parsing which significantly increases the runtime.

Approaches such as GEMSjax [4] or EMF-REST [19] provide **Re**presentational **S**tate **T**ransfer (REST) access to Ecore models and translate them into either JSON or XML. Compared to a method that stores JSON data in a document oriented database, these approaches require additional overhead, as the Ecore model has to be serialised into the target format on each call to the REST interface.

3 Approach

In this section we will give a detailed explanation of our approach. First we discuss and reason about the architecture of our metamodel and its components and then present an example.

3.1 Architecture

An overview of the MoDiGen metamodel architecture is given in Figure 1. The design is focused on universality and simplicity. We utilise standard conform JSON to store both model and instance data. This helps us to achieve programming language agnosticism and scalability. In the following paragraphs we will discuss the architecture depicted in Figure 1 and the corresponding metamodel components in detail.

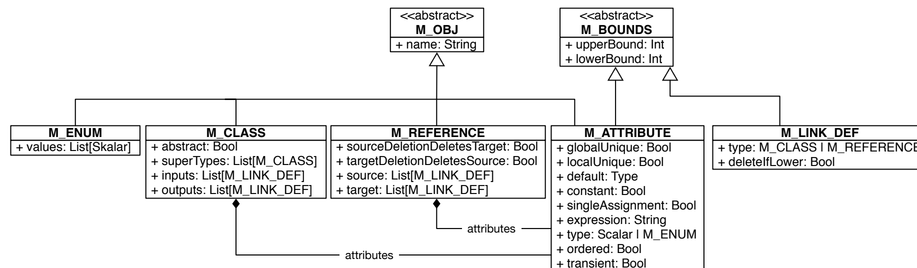


Fig. 1. MoDiGen Metamodel

M_Obj is the abstract base class of most of the metamodel's components. Its use is to provide a name attribute to all components that need to be identifiable. The name attribute is guaranteed to be unique on a model scope.

M_Bounds is abstract and defines *upperBound* and *lowerBound* attributes. Bound values can either be zero, a positive Integer, or -1 for infinity. By defining

an upper as well as a lower bound, one can model maxima, minima, as well as ranges. By default lowerBounds are set to 0 and upperBounds to -1.

The modelling of attributes in nodes and edges is done using **M_Attribute**. It extends M_Bounds so it can be defined to either be single-valued or an array with an optional maximum and/or minimum length. To further define its behaviour a number of mandatory attributes exist which will be discussed now. If *uniqueLocal* is set to true the M_Attribute behaves much like a Set data structure, for it is a collection which can not contain any duplicate values. The *uniqueGlobal* flag in contrast, guarantees that the attribute's values are unique on a model scope. This may be useful for modelling attributes like Social Security Numbers. The *default* attribute is a value of *type* and defines the initial value of the M_Attribute. Using *expression* one can define a simple arithmetic formula to derive the value of the attribute. This renders the attribute read only and can be useful in cases where one attribute depends on other attributes. Whether the M_Attribute is a String, Integer, Double, or Enum is defined by setting the *type* attribute. The *ordered* flag defines whether the attribute's values are ordered in some fashion. The attribute *transient* determines whether the attribute is transient. If set to true the attribute's value won't be stored when the model is being saved to a database. This might be useful for attributes which are the result of an expression. Single Assignment behaviour can be modelled by setting the *singleAssignment* flag, this causes the value to be settable exactly once. *Constant* on the other hand means that the value is *default* and may never be changed.

Nodes are modelled using **M_Classes**, which in addition to a number of mandatory attributes may contain an arbitrary number of user defined attributes. The mandatory attributes are defined in the following manner. *Abstract* denotes whether the M_Class is declared as abstract, meaning it may not be instantiated but only be used as a base for other M_Classes. Inheritance between M_Classes is modelled using the *superTypes* attribute. It contains all direct predecessors. By defining *superTypes* as a list we explicitly allow multiple inheritance. The *inputs* attribute is a list of all incoming M_Link_Defs and *outputs* is a list of all outgoing M_Link_Defs. On the other side M_Reference also has a *target* and *source* attribute which behaves like *inputs* and *outputs*.

M_Link_Def is used to define one endpoint of a connection and has a *type* attribute which is either M_Class or M_Reference as well as an upper- and lowerbound. The flag *deleteIfLower* defines whether the M_Class and M_Reference which contains the M_Link_Def should be deleted in case the number of values of the M_Link_Def drops to its lower bound. This means one can model a minimum count of Output/Input or Source/Target values a certain Class or Reference must have of any type.

An **M_Reference** denotes an edge between two M_Classes. By defining references as first-level classes our metamodel gains a couple of powerful modelling possibilities. For example edges may have an arbitrary number of custom attributes and allow n:m relationships. A set of standard attributes also exists which will be defined now. With *sourceDeletionDeletesTarget* set, in case the source of the M_Reference is deleted, the target, and the reference itself is also

deleted. Accordingly *targetDeletionDeletesSource* deletes the source and the reference in case the target is deleted. This can be useful to model containments and similar constructs where one end of a reference can not exist without the other end. The *source* attribute is a list of sources and the *target* attribute is a list of targets of the M_Reference. Both have the Type M_Link_Def, therefore M_References can be defined to be valid for a number of different source- and target-classes with dedicated bounds for each class in both directions. For example one can define an edge to be valid from A to B or C and further specify separate bounds for the number of edges from A to B or A to C as well as separate bounds for the number of Bs, Cs, and As involved in those edges.

Finally **M_Enum** is a simple Enum of a scalar type and might be used as a type for attributes.

3.2 Example

To demonstrate the capabilities of our metamodel, we will consider the following (oversimplified) family tree model. Three M_Classes, *Person*, *Male*, and *Female* exist, where Male and Female inherit from Person and Person is abstract. The following relationships exist between these classes: The Relationship *isHusband* has a Male source and a Female target, while the reverse relationship *isWife* has a Female source and a Male target. The Male class also has a relationship *isFather*, directed at Person and the Female class has the corresponding *isMother* relationship, also directed at Person. Figure 2 illustrates the setup.

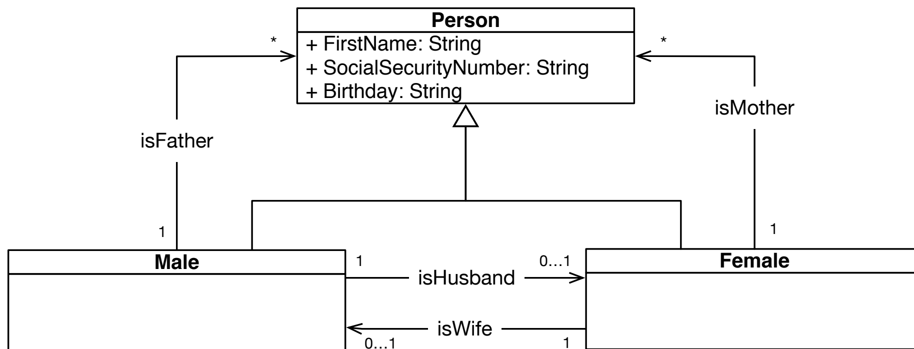


Fig. 2. Family Tree Model

Listing 1.1 is taken from the compressed familytree model and represents the M_Class Male. The *mtype* property is necessary because JSON has no type system and so this property is needed to declare that this is an M_Class. The *inputs* and *outputs* properties are M_Link_Defs linking to the respective M_Reference and indicating that at most one such relationship is allowed for one instance of

this M_Class. Because Male inherits all its attributes from Person *mAttributes* is empty.

```
"Male": {
  "mType": "mClass",
  "name": "Male",
  "superTypes": ["Person"],
  "mAttributes": [],
  "inputs": [
    { "type":"isWife",
      "upperBound": 1,
      "lowerBound": 0
    }
  ],
  "outputs": [
    { "type":"isHusband",
      "upperBound": 1,
      "lowerBound": 0
    },
    { "type":"isFather",
      "upperBound": -1,
      "lowerBound": 0
    }
  ]
},
```

Listing 1.1. The Male M_Class taken from the Family Tree example model

An M_Reference is given in listing 1.2. This is the *isHusband* M_Reference linking the Male source to the Female target. The *isHusband* reference links exactly one Male object to one Female object.

```
"isHusband": {
  "mType": "mRef",
  "name": "isHusband",
  "mAttributes": [],
  "source": [
    { "type":"Male",
      "upperBound": 1,
      "lowerBound": 1
    }
  ],
  "target": [
    { "type":"Female",
      "upperBound": 1,
      "lowerBound": 1,
    }
  ]
}
```

Listing 1.2. The isHusband M_Reference taken from the Family Tree example model

We instantiate this model with three persons. A Male instance and a Female instance, who are married to each other (using the isHusband and isWife references) and another Male, who is the child of the other two.

Listing 1.3 shows part of the JSON of an instance of the family tree model. Specifically it shows one instance of the Male class and one of the isHusband Reference. The complete JSON source for both the model and the instance can be found at MoDiGen[13].

```
"846bc8a2-00fc-401f-b626-0b0252516aee": {
  "mClass": "Male",
  "outputs": {
    "isFather": ["8e9b1093-a589-4ae4-8e1e-1b3d63a3f842"],
    "isHusband": ["ee204744-6322-49d4-928e-1442e8bc70c4"]
  },
  "inputs": {
    "isWife": ["666d4de7-e0f2-4620-8c19-d5469b40be1f"]
  },
  "mAttributes": {
    "First_Name": ["Hans"],
    "SocialSecurityNumber": ["12"],
    "Birthday": ["12-02-2015"]
  }
},

"ee204744-6322-49d4-928e-1442e8bc70c4": {
  "mRef": "isHusband",
  "source": {
    "Male": ["846bc8a2-00fc-401f-b626-0b0252516aee"]
  },
  "target": {
    "Female": ["a264a43b-6f97-4257-9243-baddbf745490"]
  }
}
```

Listing 1.3. Family tree instance

4 Evaluation

The presented approach makes it possible to create nodes and edges with equal rights. This results from the revised metamodel definition which is crucial for programmatic processing of the model data. The storage of metamodel and model information is done using JSON. This allows for easy integration and processing by conventional programming languages and web technologies. Furthermore, the number of characters and therefore the storage consumption for metamodel definitions and model instances was reduced, compared to the XML data structure of Ecore, by separating edges and nodes, and by using JSON.

The change of the number of characters based on the metamodel definition of the familytree example is shown in Figure 3. It turns out that the number

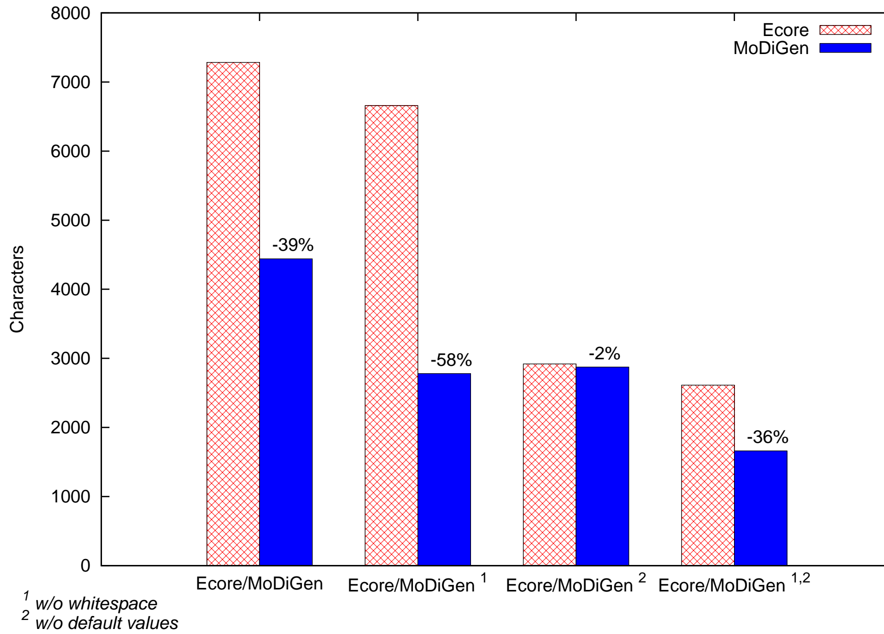


Fig. 3. Number of characters for the Familytree Metamodel

of characters were consistently reduced when compared to Ecore. The biggest difference can be revealed by removing white spaces. This comes at the expense of human readability but is irrelevant for machine processing. By removing default values a further reduction was achieved. Ecore applies these measures by default.

The development of the number of characters based on the model instance, depending on the number of nodes is shown in Figure 4. This is based on the smallest possible model instances (without whitespace and default values) for a model where all nodes are interconnected. It can be seen, that for smaller models the Ecore approach is more appropriate, but for larger models the presented approach has advantages. This is mainly caused by the changed handling of connections between objects. If only few connections are present in a model the advantage of our approach relativizes. We generated model instances with 10,000 interconnected nodes for Ecore as well as MoDiGen and found that the storage consumption of Ecore was 5,58 Gigabyte and the storage consumption of MoDiGen was 3,6 Megabytes.

Our approach has advantages regarding the scalability of big models. This is mainly due to the used data structure implemented in JSON. Data formats like JSON can easily be horizontally scaled using existing database solutions like CouchDB, MongoDB or RavenDB. This is additionally favoured by the lower memory consumption of the developed metamodel. In contrast to XML-based

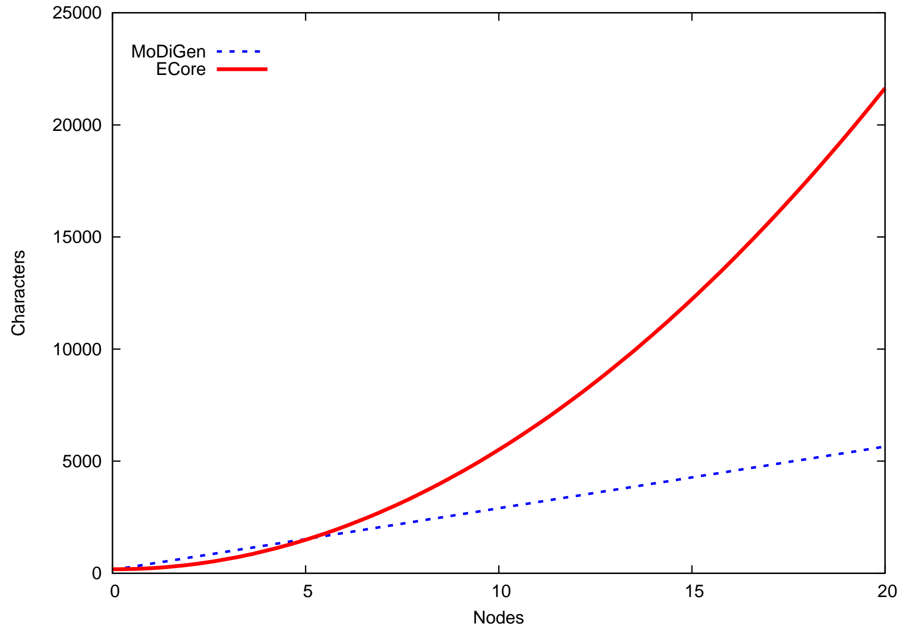


Fig. 4. Development of the number of characters for the familytree model instance according to the created nodes

data structures the JSON based data structure offers the possibility to access just parts of the stored model.

5 Conclusion and future work

We have introduced the MoDiGen metamodel and shown how its approach differs from other metamodels for DSMLs, such as Ecore or GME. Treating edges as first level objects instead of features of nodes allows for easy programmatic access to the edges. The use of JSON yields more compact models than XML, allows for seamless integration into web applications using JavaScript, and opens the door for improvements regarding scalability.

In comparison to Ecore, the MoDiGen metamodel lacks the possibility to define operations. While in the Ecore itself, EOperation is more of a placeholder, it can be given an implementation in the context of the Eclipse Modeling Framework. Edges as first level objects give easier access to references and permit the existence of stand-alone edges. However, this also means that the modeller has to explicitly state whether an edge must be automatically deleted upon the deletion of one of the connected nodes. This is not a problem in Ecore where references are deleted when the containing class is deleted. In its current form, the JSON representation of MoDiGen models still contains code that could be removed.

For example, attributes that have their default value or empty properties are still in the JSON. The JSON representation could be significantly compressed by removing that code.

In the future, we plan to implement a complete modeling framework on the basis of this metamodel and work on improving the JSON representation in terms of size. We will also use the MoDiGen metamodel for code generation projects. Furthermore, we plan on extending the metamodel to allow specification of constraints using the **O**bject **C**onstraint **L**anguage (OCL)[18].

References

1. Steinberg, D., Budinsky, F., Paternostro M., Merks E.: EMF Eclipse Modeling Framework, Second Edition. Addison-Wesley Professional (2008)
2. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The Generic Modeling Environment. In: Proceedings of WISP'2001, IEEE, Budapest (2001)
3. Maróti, M. Kereskényi, R, Kecskés, T., Völgyesi, P., Lédecyi, Á.: Online Collaborative Environment for Designing Complex Computational Systems. Procedia Computer Science, Volume 29, pp. 2432-2441 (2014)
4. Farwick, M., Agreiter, B., White, J., Forster, S., Lanzanasto, N., Breu R.: A Web-Based Collaborative Metamodeling Environment with Secure Remote Model Access. ICWE 2010, LNCS 6189, pp. 278-291. Springer-Verlag, Heidelberg (2010)
5. Itemis AG, `http://www.eclipse.org/Xtext`
6. MPS Meta Programming System, `https://www.jetbrains.com/mps/`, Accessed 2015-01-22
7. MetaEdit+ Modeler, `http://www.metacase.com/mep/`, Accessed 2015-01-22
8. Kybele GMF Generator a tool for developing GMF editors in a few steps, http://www.kybele.etsii.urjc.es/kyb_kybelegmfgen/, Accessed 2015-01-22.
9. No Magic Inc.: MagicDraw, https://www.magicdraw.com/, Accessed 2015-01-22.
10. Eugenia a tool to automatically generate a GMF editor, http://eclipse.org/epsilon/doc/eugenia/, Accessed 2015-01-22
11. Scheidgen, M.: Reference Representation Techniques for Large Models. In: BigMDE'13, ACM, Budapest (2013)
12. Kolovos, D., Rose, L., Matragkas, N., Paige, R., Guerra E., Cuadrado, J., De Lara, J., Ráth, I., Varró, D., Tisi, M., Cabot, J.: A Research Roadmap Towards Achieving Scalability in Model Driven Engineering. In: BigMDE'13, ACM, Budapest (2013)
13. MoDiGen, `http://www.modigen.de/publications/`
14. CouchDB, `http://couchdb.apache.org/`
15. MongoDB, `https://www.mongodb.org/`
16. RavenDB, `http://ravendb.net/`
17. Java Script Object Notation, `https://tools.ietf.org/html/rfc7159`
18. OMG Object Management Group: Object Constraint Language Version 2.4, `http://www.omg.org/spec/OCL/2.4/` (2014)
19. EMF-REST, `http://www.emf-rest.com`

Scalable model exploration through abstraction and fragmentation strategies

Antonio Garmendia**, Antonio Jiménez-Pastor, and Juan de Lara

Modelling and Software Engineering Research Group
`htpp://www.miso.es`
Computer Science Department
Universidad Autónoma de Madrid (Spain)

Abstract. Model-Driven Engineering (MDE) promotes the use of models to conduct all phases of software development in an automated way. However, for complex systems, these models may become large and unwieldy, and hence difficult to process and comprehend. In order to alleviate this situation, we explore the combination of model fragmentation strategies, to split models into more manageable chunks; and model abstraction and visualization mechanisms, able to provide simpler views of the models. The feasibility of this combination is confirmed based on an evaluation over a synthetic models, and the model sets of the GraBaTs'09 contest.

Keywords: Model-Driven Engineering, Model Scalability, Model Fragmentation, Model Visualization, Model Abstraction.

1 Introduction

Model Driven Engineering (MDE) promotes a model-centric approach for software development, where models are used to specify, design, test, and generate code for the final application. While models abstract details of the real system they represent, they may become large and unwieldy and therefore difficult to understand and process. Therefore, methods to cope with large models are key for a wider adoption of MDE in indutrial practice [6].

As a step in this direction, we present techniques, backed up by tools, for the scalable exploration and processing of large models. First, we show a method to specify strategies for fragmenting models. Taking inspiration from the way programming languages organize projects, our strategies organize a model as a project, which then can be divided into folders and files. Such strategies are specified over the meta-model, as "annotations" of the different classes [2].

Second, we present a method for the visual exploration of models. The method is based on filtering and abstracting models according to certain strategies, so that only a few nodes in the focus of interest are fully displayed, while others are aggregated into "abstract nodes". Then, different ways are provided

** Authors listed in alphabetical order.

to navigate through abstract nodes to the submodels they contain. Compared to fully representing a model on the screen, our approach permits higher space scalability (as fewer nodes are represented), but requires from algorithms to compute and navigate the abstractions.

We evaluate the approaches for large models and show how to combine them on the basis of two case studies. The first one is based on a synthetic generation of models, but based on a real case study of an EU project[1]. The second one is based on the large models (up to 5 million objects) provided by the GraBaTs'09 competition case study[2]. As a lesson from these experiments, we conclude that our visual exploration gives reasonable abstraction times (~ 2 secs.) for models up to roughly 10.000 objects. Beyond that point, even for a one-shot exploration, it is advisable to first fragment the model, and then apply the visual exploration.

The rest of the paper is organized as follows. Section 2 describes a method and tool support to define model fragmentation strategies. Section 3 introduces some techniques and support for model visualization and exploration. Section 4 evaluates the approaches with the two experiments. Section 5 compares with related research and Section 6 concludes.

2 Fragmenting models

We propose fragmenting models, following modular principles adopted by many programming languages and IDEs [2]. Therefore one model is organized as a Project. The model can then be fragmented into Packages (which are mapped to folders in the file system), which may hold Units (or these can be placed directly inside a project).

This kind of hierarchical organization permits structuring or defining different ways to fragment a model. Fragmentation strategies are specified at the meta-model level, where the different classes can be tagged as Project, Package and Unit, giving rise to different possible model organizations. Conceptually, the different model organizations are configured by instantiating the meta-model shown at the top of Figure 1, and then mapping such instantiation to the meta-model to which we want to apply the fragmentation strategy.

Figure 1 shows the application of the pattern to the Java JDTAST meta-model. In this case, the IJavaModel class is mapped to Project. The IJavaProject class is tagged as Package, this is possible because there is a composition relation from IJavaModel (the project) to IJavaProject, as the patterns demands by means of relation javaProjects. Another composition relation between IJavaProject and IPackageFragmentRoot allows classes which inherit from the latter (BinaryPackage-FragmentRoot and SourcePackageFragmentRoot) be tagged as Package. Finally, both IClassFile and ICompilationUnit are instantiated as Unit.

We have built tool support to apply such fragmentation strategies and to produce a modelling environment that splits monolithic instances of the meta-model according to the fragmentation strategy and supports the creation of mod-

[1] http://mondo-project.org

[2] http://www.emn.fr/z-info/atlanmod/index.php/GraBaTs_2009_Case_Study

Fig. 1: Pattern to describe the modular structure of a meta-model (top). Application to the JDTAST meta-model (bottom).

els according to such organization. Our tool is called EMF-Splitter, it is built atop of Eclipse and freely available at `http://antoniogarmendia.github.io/EMF-Splitter/`. Figure 2 shows the generated modelling environment. The environment shows an Eclipse project, named Projectset0, created from the model set0.xmi of the GraBaTs'09 contest. The project explorer shows the structure of folders and files generated from the model, which follows the specified fragmentation strategy. To the right, a tree editor shows the content of one file. The original model has about 70.000 model elements, while the fragmentation strategy fragments it into 1.800 files.

3 Exploring models

When working with models it is very useful to explore them to get some insight using our intuition, to analyse its different parts, or to find unusual or interesting features. However, big models are impossible to be completely represented in a computer monitor. Exploring models through the default tree editor of EMF is also cumbersome, as it lacks facilities to visualize, search and navigate. Moreover, many times, models lack a dedicated graphical editor providing visualization and exploration services.

To solve these problems we have developed a tool called SAMPLER (ScAlable Model exPLorER). It offers several features to visualize big models in the form of graphs, such as focusing on a specific point of the graph or making some general abstractions over the model before painting it. It allows to navigate along the visualization and even search a node through the whole model. The tool permits exploring models for which no concrete syntax has been defined, as it uses a default graph-based representation.

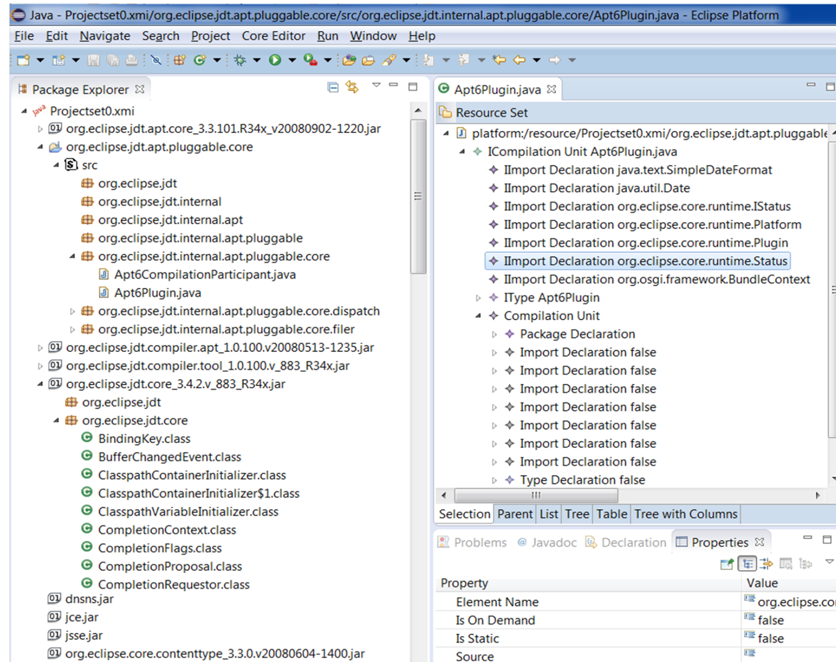


Fig. 2: Generated environment.

The main goal of SAMPLER is to draw the model without painting every element on the screen. For this purpose we have developed a composition strategy where we combine different kinds of abstractions which, executed in sequence, give a fast and compact view of the model. There are two basic operations in SAMPLER to make a model more readable: removing elements from the view, and grouping some elements in a big composite element. When we decide to remove an element, we can not view it when exploring the model, but when we compact some elements into one bigger node, we can expand it and explore the smaller elements as we wish. These operations are applied in three different steps:

- **Filters**: the first step to make our visualization easier is to apply some kind of *filter*. In many EMF models there are many intermediate objects, which may not provide the user with meaningful information, but they are technically needed to make the model conform to the meta-model. Hence, SAMPLER provides mechanism to select and filter those undesired objects, removing them from the view. When an object is filtered out, its incoming references are composed with its outgoing ones, so that the connectivity of the model is preserved.
- **Global abstraction strategies**: after filtering, there are others groups of elements which may share properties of interest, and hence it makes sense to cluster them into abstract nodes. This kind of composite operation is what we will call *global abstraction*. There are many possibilities to create a global

abstraction strategy. For example, we can unify the leaves of the containment tree of the model or we can use some cluster algorithms (like k-means).

- **Local abstraction strategies**: the last step of the SAMPLER abstraction strategy is what we call *local strategies*. After applying the previous steps, we may still have thousands of elements to draw, so that it is impossible to read anything on the screen. In this case, our approach is to focus on a part of the model at once. The local strategies focus the visualization in some point of the model (a set of elements that the user can choose), fully displaying the elements around that point, and compacting the other elements into abstract nodes. These abstract nodes do not take much space in the screen, but are explorable.

These three kinds of steps are put together to create visualization algorithms, which create a drawing of the model. In SAMPLER, we offer some basic algorithms, like just performing a global abstraction, or just a local one. These basic algorithms can be composed, and new algorithms can be incorporated by implementing a dedicated extension point.

Further than the visualization algorithms, SAMPLER provides a navigation utility. It allows, once a model has been painted on the screen, to navigate through the model. There are three navigation options:

- It is possible to expand a compacted abstract node so that, in the same screen, the first elements of the abstract node are shown together with the others elements present in the view.
- It allows to open an abstract node in another window and apply a common visualization algorithm to view this part of the model.
- It is also possible to open a new window with the containment subtree of an element of the model. As in the last option, in this new window, a common algorithm can be used to view the subtree.

Finally, SAMPLER offers a search functionality. It uses the filters algorithms described before, and allows to dynamically define different criteria for searching.

All these functionalities and tools have been implemented in a Eclipse plug-in available at `http://rioukay.github.io/sampler/`. The main elements included in the plug-in are the different Eclipse views (see Figure 3):

- The *View Preferences* view allows to change the configuration of the visualization algorithm that is being using at that moment. It also gives the possibility to change between the existing visualization algorithms.
- The *Node Information* view shows the information of the elements that have been clicked on in the canvas. If the node clicked is an abstract node, then it shows the information of its contained elements.
- The *Filter Information* view allows to add and configure additional filter steps to the end of the algorithm.

Figure 3 shows an example of how SAMPLER works. We can see the diagram visualization of the model where we have applied a local algorithm that shows the

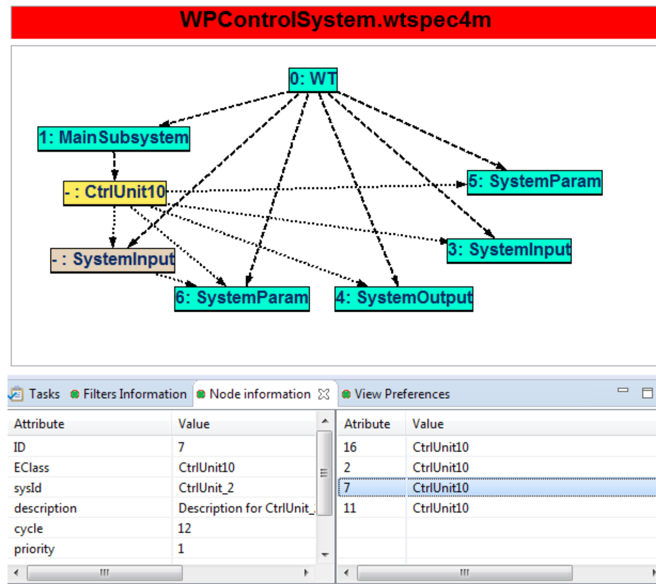


Fig. 3: Exploring a model with SAMPLER

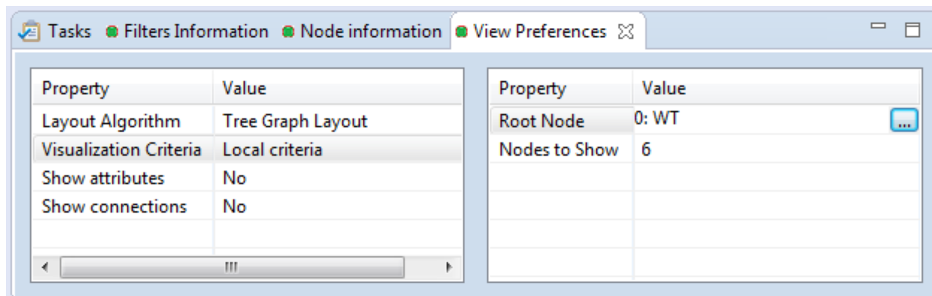


Fig. 4: Preview of the "Preferences View" of SAMPLER

root of the model (node WT) and the five nearest elements of the containment tree. The other elements are compacted according to their parents in that tree.

Each box in the diagram represent an element of the abstracted model. The blue boxes correspond to simple elements of the model and the brown ones are abstract nodes. There are two kinds of arrows connecting the nodes of the diagram: dot arrows represent references, and line arrows represent containment. Just below the canvas, we can see the views that we have described. Figure 3 shows the "Node Information" view. To the right we see the elements of the model contained in the selected compacted node (the yellow box in the diagram). To the left we show the attributes of the element selected on the right side of the view.

Figure 4 shows how the "View Preferences" view looks like. To the left we give the option to modify the generic options of the visualization and choosing

the abstraction algorithm. To the right, we allow changing the configuration of the algorithm. In the example, with the local algorithm, we can choose how many elements to show near the root, and which element of the model is the root of the visualization.

4 Evaluation

Next, we evaluate the performance of our tools to deal with large models. Our intention is to analyse to what extent large models can be explored with SAMPLER. When models become difficult to be visualized with the tool, we will fragment them first, using a fragmentation strategy, so that the smaller chunks can be visualized individually. Hence, we also perform an experiment to give an account for the incurred cost of fragmentation.

In all our tests, we used the following environment:

- Execution environment:
  - Operative System: Windows 7 Professional Service Pack 1.
  - Processor: Intel(R) Core(TM) i7-2600, 3.40GHz
  - RAM: 12 GB
- Java Virtual Machine Configuration:
  - Execution environment: Java SE 1.8 (*jre1.8.0_40*)
  - Initial memory: 512 MB
  - Maximum memory: 8 GB

4.1 Exploration performance

In this experiment, the goal is to check the performance of some of the SAMPLER abstraction strategies for large models. We generated models using an EMF random instantiator from the ATLANMOD team[3]. We used a meta-model taken from a case study of the EU project MONDO[4] in the domain of component-based embedded systems. We created 500 test models of each size. The sizes we have tested go from 100 to 6.000 model elements.

In each test, we have taken four measures, the time taken to read the model, and the time of execution of three of SAMPLER basic algorithms. Those algorithms are:

- A global algorithm that creates only one composite node with all elements inside it. This is a measure of how much time SAMPLER takes to explore the whole model (compactification algorithm)
- A global algorithm that explores the whole model detecting the leaves of the containment tree and compact them (global algorithm)
- A local algorithm that, given an object of the model, shows this element and n of its neighbours while the others are compacted (local algorithm).

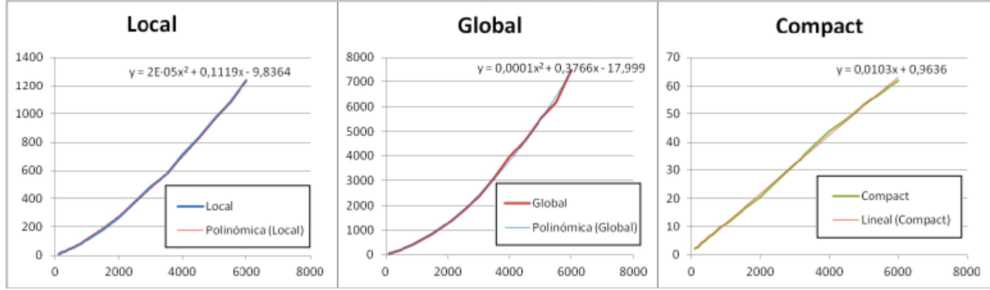


Fig. 5: Performance (ms) of the different algorithms of abstraction. From left to right: local algorithm, global algorithm and compactification algorithm.

The graphics in Figure 5 show that every algorithm takes a reasonable time to execute (no more than 10 seconds for 6.000 elements in the model) and that the local and global algorithm takes a quadratic polynomial time to execute.

After this synthetic tests, we have executed the same algorithms in the same conditions over the two first sets of JDTAST models of the GRaBaTs competitions, which have a larger size. Table 1 shows the results of the experiment for the three algorithms together with the estimation from the run of the smaller tests. As it can be noted, the time required to create the abstraction of the model is more than 25 minutes with the *set0* model and more than 5 hours with the *set1* model. Those times are not acceptable, and hence we resort to the application of another pre-drawing techniques, such as *fragmentation strategies*. The next subsection discusses on its performance.

Model	**Local Algorithm**		**Global Algorithm**		**Compactification**	
	Measure	Estimation	*Measure*	Estimation	*Measure*	Estimation
set0	1.527.543, 80	82.280, 29	1.224.024, 6,	786.525, 93	778, 8,	745, 04
set1	20.596.201	611.014, 12	13.689.961, 00	6.126.755, 18	2.080, 00	2.096, 68

Table 1: Performance (ms) of SAMPLER over some JDTAST models.

4.2 Fragmentation performance

Next, we evaluate the performance of model fragmentation. Figure 1 shows the fragmentation strategy that was applied to the JDTAST meta-model. After the application of the modularity pattern, we split all the models found in the GraBaTs'09 case study, turning each one of them into an Eclipse project.

Table 2 shows the results of our experiment. The columns depict the split time, merge time (merging all files of a fragmented model into one file), generated number of files, mean and maximum number of elements of each fragment,

[3] `http://modeling-languages.com/a-pseudo-random-instance-generator-for-emf-models/`
[4] `http://www.mondo-project.org`

Model	Split time	Merge time	# files	Avg	Max	# model elements
set0	94.362, 04	7.808, 04	1.779	40, 17	1.322	71.458
set1	231.143, 78	37.847, 10	6.240	32, 68	4.549	203.938
set2	544.609, 02	83.905, 74	6.050	345, 27	50.718	2.088.890
set3	747.739, 30	199.811, 98	4.460	1.031, 24	50.718	4.599.358
set4	808.351, 00	511.554, 59	5.068	980, 04	50.718	4.966.846

Table 2: Performance (ms) of EMF-Splitter over the test-cases of GraBaTs'09.

and the total number of elements in the whole model. We can observe that the maximum number of elements in a file is repeated for *set2*, *set3* and *set4*. This happens because this group of models was built by adding java classes incrementally. For example, *set2* is formed by *set1* and the addition of some java packages.

The results shows that, in average the exploration of the files with SAMPLER would become easier, because the largest average size is about 1.000 elements (which are easily explorable), while the maximum number of elements in a file is 50.718, which would take about a minute and a half to load.

5 Related work

In this section we focus on existing works dealing with model fragmentation, and model exploration and visualization of large graphs.

Due to the need to process large models, some authors have proposed to split models for solving different tasks. For instance, Scheidgen and Zubow [10] propose a persistence framework that allows automatic and transparent fragmentation to add, edit and update EMF models. This process is executed at runtime, with considerable performance gains. However, the user does not have a view of the different fragments as we have in *EMF Splitter*, which could help improving the comprehensibility of the fragments.

Other works [5, 11] decompose models into submodels for enhancing their comprehensibility. For example, in [5], the authors propose an algorithm to fragment a model into submodels (actually they can build a lattice of submodels), where each submodel is conformant to the original meta-model. The algorithm considers cardinality constraints but not general OCL constraints, and there is no tool support. Other works use Information Retrieval (IR) algorithms to split a model based on the relevance of its elements [11]. Therefore, splitting models that belong to the same meta-model can produce different structures.

Other works directed to define model composition mechanisms [3, 4, 12] are intrusive. These papers [3, 12] present techniques for model composition and realize the importance of modularity in models as a research topic to minimise the effort. Strüber et. al [4] present a structured process for model-driven distributed software development which is based on split, edit and merge models for code generation.

Regarding model visualization, in [9], the authors propose a framework call ELVIZ for model visualization, based on the transformation of input models

to appropriate output formats. For example, given a class diagram, they can extract the number of methods per class, and visualize such numbers as a bar chart. ELVIZ facilitates the generation of input models to different visualization outputs relying on mappings.

In [1], the authors present the tool Explen, which uses slicing techniques in order to visualize large meta-models. Similar to our approach, it is possible to focus on a given class, and select some slicing criteria (e.g., show the composition relations only, show only a certain radius of classes, or show the sub/super type hierarchy). They also include a flattening filter, which presents a hierarchy in the form of a unique class. SAMPLER supports the visualization of models and meta-models, and the abstractions/slice criteria are extensible. Moreover, we support different navigation strategies from abstracted models.

The analysis of large graphs arising in e.g., social networks have produced some summarization techniques, which try to encode in smaller graphs [7] or as a variety of statistics [8] the main features of the large graph. For this purpose, they find the most often occurring subtype graphs (cliques, starts, chains, etc) in graphs. In the context of MDE, this information is encoded in the meta-model. Other methods are more flexible, as they allow customization of the interesting attributes of nodes [13], and nodes with similar values are summarized in a single node. This would be similar to SAMPLERs global abstractions.

Altogether, to the best of our knowledge, our approach to combine model fragmentation and model visualization techniques is novel.

6 Conclusions and future work

In this work, we have proposed the combination of model fragmentation and model visualization techniques to explore large models. Model fragmentation is performed by applying fragmentation strategies at the meta-model level. Model exploration is done by applying different abstraction strategies to the model, and with the availability of model exploration techniques. We have performed an evaluation of the approach for large models. We have seen that for models in the range of up to roughly six thousand elements, abstraction gives good results. For large models, such as those of the GraBaTs'09, our proposal is fragmenting them first. In this case, fragments become of manageable size, and then can be visually explored.

In the future, we aim at the tighter integration of SAMPLER with the information provided by the fragmentation strategies. In particular, when exploring a fragmented model, we currently need to use the package explorer to move between fragments. In the future, we would like SAMPLER to use the fragmentation information as a (global) abstraction algorithm. This way, fragments would be explored transparently from within the SAMPLER visual canvas.

Acknowledgements. Work supported by the Spanish Ministry of Economy and Competitivity (TIN2011-24139, TIN2014-52129-R), the EU commission (FP7-ICT-2013-10, #611125) and the Community of Madrid (S2013/ICE-3006)

References

1. A. Blouin, N. Moha, B. Baudry, and H. A. Sahraoui. Slicing-based techniques for visualizing large metamodels. In *Second IEEE Working Conference on Software Visualization, VISSOFT*, pages 25–29. IEEE Computer Society, 2014.
2. A. Garmendia, E. Guerra, D. S. Kolovos, and J. de Lara. EMF splitter: A structured approach to EMF modularity. In *XM@MoDELS*, volume 1239 of *CEUR Workshop Proceedings*, pages 22–31. CEUR-WS.org, 2014.
3. F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler. On language-independent model modularisation. *T. Asp.-Oriented Soft. Dev. VI*, 6:39–82, 2009.
4. P. Kelsen and Q. Ma. A modular model composition technique. In *Proceedings of FASE'10*, volume 6013 of *LNCS*, pages 173–187. Springer, 2010.
5. P. Kelsen, Q. Ma, and C. Glodt. Models within models: Taming model complexity using the sub-model lattice. In *Proceedings of FASE'11*, volume 6603 of *LNCS*, pages 171–185. Springer, 2011.
6. D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, and J. Cabot. A research roadmap towards achieving scalability in model driven engineering. In *Proc. BigMDE '13*, pages 2:1–2:10, New York, NY, USA, 2013. ACM.
7. D. Koutra, U. Kang, J. Vreeken, and C. Faloutsos. VOG: summarizing and understanding large graphs. In *Proceedings of the 2014 SIAM International Conference on Data Mining*, pages 91–99. SIAM, 2014.
8. M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.
9. M. Ostendorp, J. Jelschen, and A. Winter. ELVIZ: A query-based approach to model visualization. In *Modellierung 2014*, pages 105–120, 2014.
10. M. Scheidgen, A. Zubow, J. Fischer, and T. H. Kolbe. Automated and transparent model fragmentation for persisting large models. In *Proceedings of MoDELS'12*, volume 7590 of *LNCS*, pages 102–118. Springer, 2012.
11. D. Strüber, J. Rubin, G. Taentzer, and M. Chechik. Splitting models using information retrieval and model crawling techniques. In *Proceedings of FASE'14*, volume 8411 of *LNCS*, pages 47–62. Springer, 2014.
12. D. Strüber, G. Taentzer, S. Jurack, and T. Schäfer. Towards a distributed modeling process based on composite models. In *Proceedings of FASE'13*, volume 7793 of *LNCS*, pages 6–20. Springer, 2013.
13. Y. Tian, R. A. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 567–580. ACM, 2008.

An Efficient Computation Strategy for allInstances()

Ran Wei and Dimitrios S. Kolovos

Department of Computer Science,
University of York, United Kingdom
{ran.wei, dimitris.kolovos}@york.ac.uk

Abstract. Contemporary model query and transformation engines typically provide built-in facilities for retrieving all instances of a particular type/kind regardless of their location in a model (i.e. OCL's allInstances()). When implemented in a naive manner, such facilities can be computationally expensive for large models. We contribute a novel approach for implementing allInstances()-like facilities for EMF models, which makes use of static analysis and metamodel introspection and we report on the results of extensive benchmarking against alternative approaches.

1 Introduction

As models involved in MDE processes get larger and more complex[1, 2], model query and transformation languages are being stressed to their limits [3, 4]. One of the most computationally-expensive operations that model query and transformation engines support is the ability to retrieve collections of instances of a particular type/kind regardless of their location in a model (i.e. OCL's *allInstances()*). In this paper we discuss existing strategies for computing such collections of instances and we highlight their advantages and shortcomings. We then contribute a novel computation strategy that makes use of static analysis and metamodel introspection to pre-compute and cache all such collections needed in the context of a query in one pass. We present an implementation of the proposed strategy on top of an existing model query language (Epsilon's EOL [5]) and benchmark it against alternative computation strategies.

2 Background and Motivation

The majority of contemporary model query and transformation languages provide support for retrieving collections of all model elements that are instances of a particular type/kind. For example, OCL, QVTr, ATL, and Acceleo provide the built-in *allInstances()* operation which can be invoked on a type to return a set containing all its instances (e.g. *Person.allInstances()*), Epsilon's EOL provides the *getAllOfType()* and *getAllOfKind()* operations, and QVTo the *objects(type : Type)* and *objectsOfType(type : Type)* operations that operate in a similar way.

We collectively refer to all such operations as *allInstances()* in the remainder of the paper.

For file-based EMF models, a naive strategy to implement *allInstances()* is to navigate the in-memory model element containment tree upon invocation, and collect and return all instances of the requested type. Repeatedly traversing the containment tree to fetch all instances of the same type for multiple invocations of the operation on that type is clearly inefficient, so the majority of model query and transformation engines provide support for caching and reusing the results of previous invocations of the operation (this is straightforward for side-effect free languages but requires some additional book-keeping for languages that can mutate the state of a model).

When a query (or a transformation) contains a large number of calls to *allInstances()* for different types, instead of traversing the containment tree for each of these calls/types on demand, it can be more efficient for the execution engine to pre-compute and cache all these collections in one pass at start-up instead (*greedy caching*). This can incur a higher upfront cost and increase the memory footprint, however, for a sufficiently high number of invocations on different types, it is very likely to pay off eventually – particularly as models grow in size.

Overall, when more than one calls to *allInstances()* are made for different types in the context of a query, the on-demand approach is sub-optimal in terms of performance. On the other hand, if a query only calls *allInstances()* on a small number of types (compared to the total number of types in the metamodel), greedy caching is wasteful.

3 Program- and Metamodel-Aware Instance Collection

Given in-advance knowledge of the metamodel of a model, and the types on which *allInstances()* is likely to be invoked in the context of a query (e.g. obtained through static analysis of the query itself) operating on that model, in this section we demonstrate how a query execution engine can efficiently pre-compute and cache the results of only these invocations by traversing the contents of the model only once.

We demonstrate the proposed algorithms and their supporting data structures with reference to a concrete OCL-like query language (Epsilon's EOL). For conciseness, we also restrict the discussion to EOL queries operating on a single EMF-based model which conforms to an Ecore metamodel comprising exactly one EPackage. However, the proposed approach is trivially portable to other query and transformation languages of a similar nature, and to queries that involve more than one models conforming to multi-EPackage metamodels.

3.1 Cache Configuration Model

Figure 1 demonstrates a data structure (in the form of a metamodel), an instance of which needs to be populated at compile-time (e.g. by statically analysing the

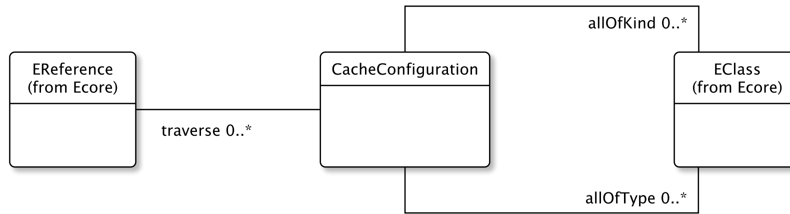


Fig. 1. Cache Configuration Metamodel

query of interest and by introspecting the metamodel of models on which it will be executed) in order to facilitate efficient execution of *allInstances()* at runtime.

CacheConfiguration acts as a *container* for the *EClasses* of the model's metamodel that the engine may need to retrieve all instances of in the context of the query of interest. *EClasses* of interest can be linked to a *CacheConfiguration* through the latter's *allOfKind* and *allOfType* references (EOL, like QVTo, support distinct operations for computing all direct and indirect instances of a given type). We intentionally refrain from discussion the *traverse* reference in Figure 1 for now.

3.2 Query Static Analysis

The first step of the process is to generate an initial version of the cache configuration model by statically analysing the query of interest. Figure 2 demonstrates the type-resolved abstract syntax graph of the example EOL program illustrated in Listing 1.1, which operates on models conforming to the metamodel of Figure 3. To compute the initial version of the cache configuration model we need to iterate through the abstract syntax graph and locate instances of:

- *MethodCallExpression* for which the name of the method called is *allOfKind*, *allOfType*, *allInstances* (alias of *allOfKind()*), the resolved type of their target expression is *ModelElementType*, and which have no parameter values;
- *PropertyCallExpression* for which the name of the property is *all* (alias of *allOfKind()*), and the resolved type of their target expression is *ModelElementType*.

Listing 1.1. An example EOL Program

```
1  WebPage.allOfType().println();
2  Member.allOfKind().println();
```

Having identified the calls of interest, we construct a new *CacheConfiguration* and for each call to *allOfType()* we create an *allOfType* link to its respective *EClass*. Similarly, for all other calls of interest we link the respective *EClasses* to the cache configuration via its *allOfKind* reference. The initial extracted cache configuration model for our running example is illustrated in Figure 5.

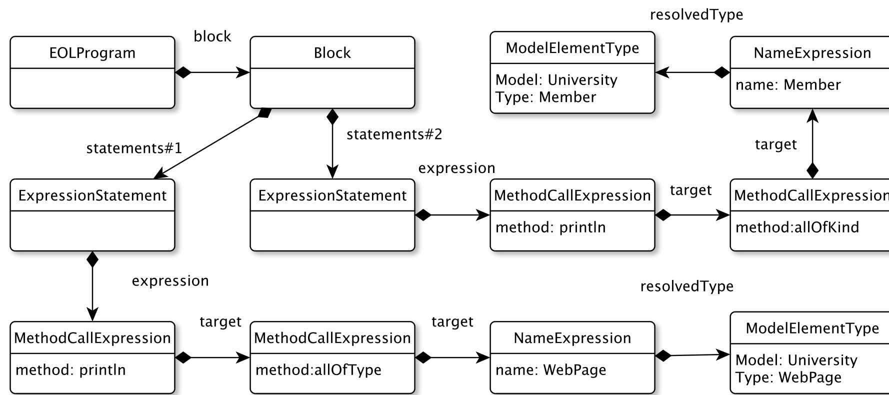


Fig. 2. The Abstract Syntax Graph of the EOL program of Listing 1.1

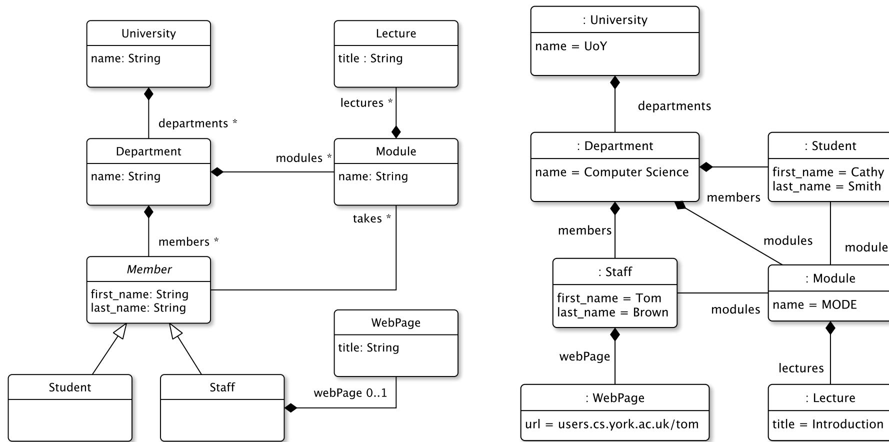


Fig. 3. The University Metamodel

Fig. 4. The University Model

3.3 Containment Reference Pruning

Following the process discussed above, the execution engine can now be aware of all the *allInstances()* collections it needs to pre-compute and cache (*Web-*

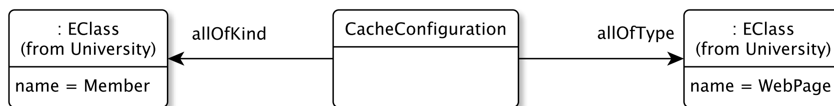


Fig. 5. Initial Extracted Cache Configuration Model

Page.allOfType() and *Member.allOfKind()* in our running example). The next step is to collect the model elements of interest in one pass and as efficiently as possible. A straightforward collection strategy would involve navigating the entire model containment tree, assessing whether each model element is of one of the types of interest and, if so, adding it to the appropriate cache(es).

However, by inspecting the example model in Figure 4, we observe that traversing the containment closure of the *modules* reference of the "Computer Science" Department model element is guaranteed not to reveal any model elements of interest (according to the metamodel of Figure 3 modules can only contain lectures and neither of these types of elements are of interest to the query). This observation can be generalised and exploited to prune the subset of the containment tree that the engine will need to visit in order to populate the caches of interest.

To achieve this we need to analyse the metamodel and compute the subset of containment references that can potentially lead to elements of interest. The proposed algorithm is illustrated in Algorithm 1. Please note that the algorithm has been simplified for presentation purposes and that implementations of the algorithm need to make use of memoisation to avoid infinite recursion that can be caused by circular containment references of no interest. Adding the computed containment references that need to be traversed at runtime to the (incomplete) cache configuration model of Figure 5, produces the (complete) configuration model of Figure 6.

3.4 Instance Collection and Caching

Having computed the cache configuration model, the final step includes traversing only the identified containment references of the in-memory model at runtime in a top-down recursive manner to collect and cache the elements of interest.

For example, with reference to the example model of Figure 4, the instance collection process starts at the top-level *:University* element. The element's EClass is not linked to the cache configuration via one of its *allOfType* or *allOfKind* references, and as such the element is not cached. Navigating the university's *departments* reference reveals a *:Department* element, which also does not need to be cached. The process does not need to navigate the department's *modules* reference as it is not linked to the cache configuration via the latter's *traverse* reference, and as such it proceeds with its *members* reference. Traversing the *members* reference reveals an instance of *Student* and an instance of *Staff*, both of which are cached in preparation for the *Member.allOfKind()* invocation. Similarly, the *webpage* reference of :Staff is traversed and reveals a *:WebPage*, which is also cached in preparation for the *WebPage.allOfType()* invocation.

```
let cm = the initial version of the configuration cache model;
let p = the EPackage that the model conforms to;
let refs = empty list of EReferences;

foreach non-abstract EClass c in p do
    foreach containment EReference r of c do
        call shouldBeTraversed(r);
    end
end

function shouldBeTraversed(r : EReference) : Boolean
    let types = transitive closure of r's type and all its sub-types;
    if types includes any of the EClasses in cm then
        add r to refs;
        return true;
    end
    else
        foreach containment EReference tr of each of the types do
            if shouldBeTraversed(tr) then
                return true;
            end
        end
        return false;
    end
end
```

Algorithm 1: Containment Reference Selection Algorithm

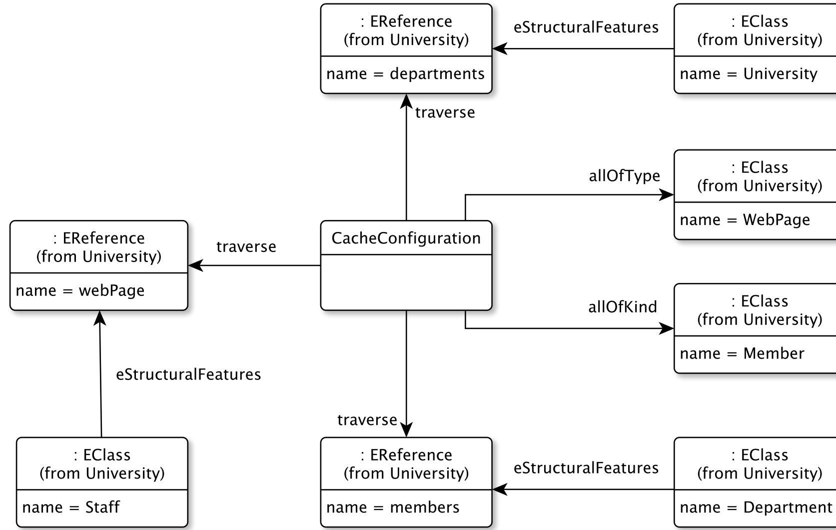


Fig. 6. Complete Cache Configuration Model

4 Evaluation

In this section we report on the results of benchmarks performed on four different strategies for computing *allInstances()*.

1. Lazy (on-demand) computation (L)
2. Greedy pre-caching (G)[1]
3. Type-aware pre-caching (T)[2]
4. Type-and-reference-aware pre-caching (TR)

Benchmarks were performed on a computer with Intel(R) Core(TM) i7 CPU @ 2.3GHz, with 8GB of physical memory, running OS X Yosemite. The version of the Java Virtual Machine used was 1.8.0_31-b13. Results are in seconds.

For our benchmarks, models of varying sizes obtained from reverse engineered Java code in the 2009 GraBaTs contest[3] are used. These models, named set0, set1, set2, set3 and set4 (9.2MB, 27.9MB, 283.2MB, 626.7MB, 676.9MB respectively) are stored in XMI 2.0 format and have been used for various benchmarks for different tools [6, 7].

4.1 Model Element Coverage

To quantify model coverage in our benchmarks, we counted the number of elements in each data set and then automatically generated EOL programs which exercise 20%, 40%, 60%, 80% and 100% of the total elements for each data set. An example generated EOL program is provided in Listing 1.2.

We then executed all the generated EOL programs and measured performance in terms of the time taken to load the models with the four different strategies and the time taken to execute the programs.

Listing 1.2. An example of generated EOL program for model element coverage

```
1 var size = 0;
2 var methodInvocation = MethodInvocation.all.first();
3 size = size + MethodInvocation.all.size();
4 var qualifiedName = QualifiedName.all.first();
5 size = size + QualifiedName.all.size();
6 ...
7 size.println();
```

[1] As discussed in Section 2, this approach naively pre-computes all possible *allOfType* and *allOfKind* caches.

[2] This approach makes use of static analysis as discussed in Section 3.2 but does not prune containment references and as such it needs to visit the entire containment tree at runtime. It is included in this benchmark only to assess the additional benefits of containment reference pruning.

[3] GraBaTs2009: 5th Int. Workshop on Graph-Based Tools, `http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/`

4.2 Results

The obtained results are presented in Table 1. Initials *L*, *G*, *T* and *TR* represents the approaches aforementioned (Lazy, Greedy, Type-Aware and Type-and-Reference-Aware). Since the execution time of the EOL programs for *G*, *T* and *TR* is practically the same[4], we only present one result for all three of them under the * columns. *Imp.* represents the performance improvement of a certain approach, *Load* represents the time it takes to load the models, whereas *Exec.* represents the time it takes to execute the EOL programs. Finally, *Total* represents the time it takes to load the model and execute an EOL program for a single experiment.

From the benchmarks we observe that with the *Greedy*, *Type-Aware* and *Type-and-Reference-Aware* approaches, programs execute significantly faster than with the *Lazy* approach. These approaches require more time to load the models due to the overhead incurred by their respective caching logic; such overhead affects the performance for small data sets (set 0 in this case). However, as the size of models gets larger, these approaches provide marginal benefits in terms of the time it takes to load a model and to execute an EOL program (total time). In general, *TR* provides better performance but for some cases in which *TR* needs to visit elements deep in the containment tree, *T* and *G* marginally outperform it. In terms of memory footprint, the three approaches behave very similarly and incur a small linear overhead compared to *L*.

5 Related Work

Several database-based model persistence prototypes have been proposed for persisting and loading large models, including Morsa [8], Neo4EMF [7], MongoEMF [9], EMF Fragments [10] and Hawk [6]. The general idea behind these prototypes is that they are able to load only the parts of a model that are needed for the task at hand (e.g. to compute particular queries), so that large models can be accessed efficiently both in terms of loading time and memory consumption.

Computing *allInstances()* in such systems typically does not require traversing the entire model and can be achieved through efficient internal queries expressed in the underpinning database's native query language (e.g. SQL, Cypher). Despite the clear technical advantages of database-based technologies, there are still valid reasons for using file-based formats (e.g. XMI) for model persistence in certain contexts, such as standards-compliance, tool interoperability, and compatibility with existing file-based version control systems such as Git and Subversion.

[4] This is expected as all three strategies populate all caches required before the EOL program executes.

Table 1. Benchmark results for Lazy, Greedy, Type-Aware, Type-and-Reference-Aware caching (* in the table represents the results for G,T and TR collectively).

Perc.	L		G	T	TR	*	Imp.G	Imp.T	Imp.TR	Imp.*
	Load	Exec.	Load	Load	Load	Exec.	Total	Total	Total	Exec.
	sec.		sec.	sec.	sec.	sec.	%	%	%	%
					Set0					
20%	0.552	0.015	0.652	0.554	0.572	0.001	-15.17%	2.12%	-1.06%	93.33%
40%	0.555	0.007	0.631	0.572	0.561	0.002	-12.63%	-2.14%	-0.18%	71.43%
60%	0.549	0.012	0.645	0.571	0.573	0.003	-15.51%	-2.32%	-2.67%	75.00%
80%	0.543	0.026	0.652	0.573	0.576	0.005	-15.47%	-1.58%	-2.11%	80.77%
100%	0.552	0.141	0.638	0.623	0.619	0.013	6.06%	8.23%	8.80%	90.78%
Perc.					**Set1**					
20%	1.643	0.606	1.856	1.653	1.672	0.01	17.03%	26.06%	25.21%	98.35%
40%	1.596	0.595	1.875	1.736	1.711	0.011	13.92%	20.26%	21.41%	98.15%
60%	1.587	0.556	1.843	1.786	1.773	0.013	13.39%	16.05%	16.66%	97.66%
80%	1.611	0.571	1.86	1.787	1.788	0.017	13.98%	17.32%	17.28%	97.02%
100%	1.606	0.626	1.866	1.852	1.852	0.021	15.46%	16.08%	16.08%	96.65%
Perc.					**Set2**					
20%	14.159	2.244	17.169	14.802	14.809	0.007	-4.71%	9.72%	9.68%	99.69%
40%	14.061	4.402	17.979	16.587	16.613	0.015	2.54%	10.08%	9.94%	99.66%
60%	14.456	3.305	16.96	16.276	15.851	0.02	4.40%	8.25%	10.64%	99.39%
80%	15.151	5.685	18.145	17.724	18.217	0.03	12.77%	14.79%	12.43%	99.47%
100%	15.223	6.2	17.32	17.769	17.839	0.036	18.98%	16.89%	16.56%	99.42%
Perc.					**Set3**					
20%	34.199	8.706	38.096	34.17	33.753	0.017	11.17%	20.32%	21.29%	99.80%
40%	31.786	9.756	37.552	35.086	34.809	0.028	9.54%	15.47%	16.14%	99.71%
60%	31.835	12.222	37.528	36.516	35.662	0.045	14.72%	17.01%	18.95%	99.63%
80%	32.417	11.456	39.301	39.302	37.795	0.068	10.27%	10.26%	13.70%	99.41%
100%	35.872	13.7	38.659	40.779	40.513	0.071	21.87%	17.59%	18.13%	99.48%
Perc.					**Set4**					
20%	36.133	7.586	43.745	39.477	37.278	0.018	-0.10%	9.66%	14.69%	99.76%
40%	37.99	12.973	43.515	41.044	41.01	0.039	14.54%	19.39%	19.45%	99.70%
60%	36.457	14.131	44.883	42.348	41.055	0.05	11.18%	16.19%	18.75%	99.65%
80%	37.782	11.762	41.932	44.038	45.168	0.065	15.23%	10.98%	8.70%	99.45%
100%	37.617	14.563	44.813	46.914	43.406	0.078	13.97%	9.94%	16.67%	99.46%

6 Conclusion and Future Work

In this paper we have proposed a novel approach for computation and caching of *allInstances()*-like operations (e.g. used in declarative model transformation rules) on in-memory EMF models. We have compared the proposed approach against three alternative approaches via extensive benchmarking and demonstrated the benefits it delivers in terms of aggregate model loading and query execution time. Such an approach brings benefits only to model management programs which trigger multiple calls to *allInstances()*.

In future iterations of this work, we wish to investigate how static analysis and metamodel introspection can be used to further improve performance of computationally-expensive queries at runtime (e.g. by constructing and maintaining in-memory indexes that can improve the performance of collection filtering operations applied to the results of *allInstances()*).

References

1. Parastoo Mohagheghi, Miguel A Fernandez, Juan A Martell, Mathias Fritzsche, and Wasif Gilani. MDE adoption in industry: challenges and success criteria. In *Models in Software Engineering*, pages 54–59. Springer, 2009.
2. Paul Baker, Shiou Loh, and Frank Weil. Model-Driven Engineering in a Large Industrial Context. In *Model Driven Engineering Languages and Systems*, pages 476–491. Springer, 2005.
3. Dimitrios S. Kolovos, Richard F. Paige, and Fiona AC Polack. Scalability: The holy grail of model driven engineering. In *ChaMDE 2008 Workshop Proceedings: International Workshop on Challenges in Model-Driven Software Engineering*, pages 10–14, 2008.
4. Marcel Van Amstel, Steven Bosems, Ivan Kurtev, and Luís Ferreira Pires. Performance in model transformations: experiments with ATL and QVT. In *Theory and Practice of Model Transformations*, pages 198–212. Springer, 2011.
5. Dimitrios S. Kolovos, Richard F Paige, and Fiona AC Polack. The Epsilon Object Language (EOL). In *Model Driven Architecture–Foundations and Applications*, pages 128–142. Springer, 2006.
6. Konstantinos Barmpis and Dimitris Kolovos. Hawk: Towards a scalable model indexing architecture. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, page 6. ACM, 2013.
7. Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, and David Launay. Neo4EMF, a scalable persistence layer for EMF models. In *Modelling Foundations and Applications*, pages 230–241. Springer, 2014.
8. Javier Espinazo Pagán, Jesúss Sánchez Cuadrado, and Jesús García Molina. Morsa: A scalable approach for persisting and accessing large models. In *Model Driven Engineering Languages and Systems*, pages 77–92. Springer, 2011.
9. Bryan Hunt. MongoEMF, 2014, https://github.com/BryanHunt/mongo-emf/wiki.
10. Markus Scheidgen. Reference representation techniques for large models. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, page 5. ACM, 2013.

Decentralized Model Persistence for Distributed Computing

Abel Gómez, Amine Benelallam, and Massimo Tisi

AtlanMod team (Inria, Mines Nantes, LINA), France
{abel.gomez-llana|amine.benelallam|massimo.tisi}@inria.fr

Abstract. The necessity of manipulating very large amounts of data and the wide availability of computational resources on the Cloud is boosting the popularity of distributed computing in industry. The applicability of model-driven engineering in such scenarios is hampered today by the lack of an efficient model-persistence framework for distributed computing. In this paper we present NeoEMF/HBase, a persistence backend for the Eclipse Modeling Framework (EMF) built on top of the Apache HBase data store. Model distribution is hidden from client applications, that are transparently provided with the model elements they navigate. Access to remote model elements is decentralized, avoiding the bottleneck of a single access point. The persistence model is based on key-value stores that allow for efficient on-demand model persistence.

Keywords: Model Persistence, Key-Value Stores, Distributed Persistence, Distributed Computing

1 Introduction

The availability of large data processing and storage in the Cloud is becoming a key resource for part of today's industry, within and outside IT. It offers a tempting alternative for companies to process, analyze, and discover new data insights, yet in a cost-efficient manner. Thanks to existing Cloud computing companies, this facility is extensively available for rent [11]. This ready-to-use IT infrastructure is equipped with a wide range of distributed processing frameworks, for companies that have to occasionally process large amounts of data.

One of the principal ingredients behind the success of distributed processing are distributed storage systems. They are designed to answer to data processing requirements of distributed and computationally extensive applications, i.e., wide applicability, scalability, and high performance. Appearing along with MapReduce [10], BigTable [9] strongly stood in for these qualifications. One of the most compliant open-source implementations of MapReduce and BigTable are Apache's Hadoop [17] and HBase [18], respectively.

Another success factor for widespread distributed processing is the appearance of high-level languages for simplifying distribution by a user-friendly syntax (mostly SQL-like). They transparently convert high-level queries into a series of

parallelizable jobs that can run in distributed frameworks, such as MapReduce, therefore making distributed application development convenient.

We believe that Model-Driven Engineering (MDE), especially the query/-transformation languages and engines, would be suitable for developing distributed applications on top of structured data (models). Unfortunately, MDE misses some fundamental bricks towards building fully distributed transformation/query engines. In this paper we address one of those components, i.e. a model-persistence framework for distributed computing. Several distributed model-persistence frameworks exist today [3,16]: for the Eclipse Modeling Framework (EMF) [6] two examples are Connected Data Objects (CDO) [3] that is based on object relational mapping[1], and *EMF fragments* [15], that maps large chunks of model to separate URIs. We argue that these solutions are not well-suited for distributed computing, exhibiting one or more of the following faults:

- Model distribution is not transparent: so queries and transformations need to explicitly take into account that they are running on a part of the model and not the whole model (e.g. *EMF fragments*)
- Even when model elements are stored in different nodes, access to model elements is centralized, since elements are requested from and provided by a central server (e.g. CDO over a distributed database). This constitutes a bottleneck and does not exploit a possible alignment between data distribution and computation distribution.
- The persistence backend is not optimized for atomic operations of model handling APIs. In particular files (e.g. XMI over HDFS [7]), relational databases or graph databases are widely used while we have shown in previous work [12] that key-value stores are very efficient in typical queries over very large models. Moreover key-value stores are more easily distributed with respect to other formats, such as graphs.
- The backend assumes to split the model in balanced chunks (e.g. *EMF Fragments*). This may not be suited to distributed processing, where the optimization of computation distribution may require uneven data distribution.

In this paper we present NeoEMF/HBase, a persistence backend for EMF built on top of the Apache HBase data store. NeoEMF/HBase is transparent w.r.t. model manipulation operations, decentralized, and based on key-value stores. The tool is open-source and publicly available at the paper's website[2]. This paper is organized as follows: Section 2 presents HBase concepts and architecture, Section 3 presents the NeoEMF/HBase architecture, data model and properties; and finally, Section 4 concludes the paper and outlines future work.

[1] CDO servers (usually called *repositories*) are built on top of different data storage solutions (ranging from relational databases to document-oriented databases). However, in practice, only relational databases are commonly used, and indeed, only *DB Store* [1], which uses a proprietary Object/Relational mapper, supports all the features of CDO and is regularly released in the *Eclipse Simultaneous Release* [2,4,5].

[2] `http://www.emn.fr/z-info/atlanmod/index.php/NeoEMF/HBase`

2 Background: Apache HBase

Apache HBase [18] is the Hadoop [17] database, a distributed, scalable, versioned and non-relational big data store. It can be considered an open-source implementation of Google's Bigtable proposal [9].

2.1 HBase data model

In HBase, data is stored in tables, which are sparse, distributed, persistent multi-dimensional sorted maps. A map is indexed by a row key, acolumn key, and a timestamp. Each value in the map is an uninterpreted array of bytes.

HBase is built on top of the following concepts [14]:

Table — Tables have a name, and are the top-level organization unit for data in HBase.
Row — Within a table, data is stored in *rows*. Rows are uniquely identified by their *row key*.
Column Family — Data within a row is grouped by *column family*. Column families are defined at table creation and are not easily modified. Every row in a table has the same column families, although a row does not need to store data in all its families.
Column Qualifier — Data within a column family is addressed via its *column qualifier*. Column qualifiers do not need to be specified in advance and do not need to be consistent between rows.
Cell — A combination of *row key*, *column family*, and *column qualifier* uniquely identifies a *cell*. The data stored in a cell is referred to as that cell's value. Values do not have a data type and are always treated as a `byte[]`.
Timestamp — Values within a cell are versioned. Versions are identified by their version number, which by default is the timestamp of when the cell was written. If the timestamp is not specified for a read, the latest one is returned. The number of cell value versions retained by HBase is configured for each column family (the default number of cell versions is three).

Figure 1 (extracted from [9]) shows an excerpt of an example table that stores Web pages. The row name is a reversed URL. The `contents` column family contains the page contents, and the `anchor` column family contains the

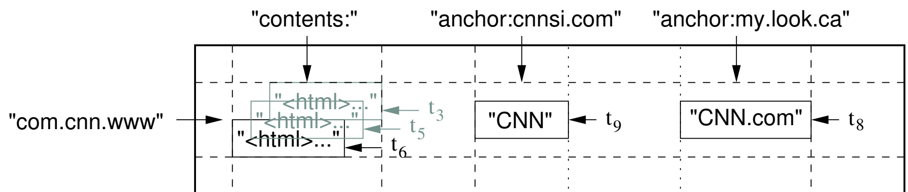


Fig. 1: Example of a table in HBase/BigTable (extracted from [9])

text of any anchors that reference the page. CNN's home page is referenced by both the *Sports Illustrated* and the *MY-look* home pages, so the row contains columns named `anchor:cnnsi.com` and `anchor:my.look.ca`. Each anchor cell has one version; the `contents` column has three versions, at timestamps t_3, t_5, and t_6.

2.2 HBase architecture

Fig. 2 shows how HBase is combined with other Apache technologies to store and lookup data. Whilst HBase leans on HDFS to store different kind of configurable size files, ZooKeeper [19] is used for coordination. Two kinds of nodes can be found in an HBase setup, the so-called *HMaster* and the *HRegionServer*. The *HMaster* is the responsible for assigning the regions (*HRegions*) to each *HRegionServer* when HBase is starting. Each *HRegion* stores a set of rows separated in multiple column families, and each column family is hosted in an *HStore*. In HBase, row modifications are tracked by two different kinds of resources, the *HLog* and the *Stores*. The *HLog* is a store for the write-ahead log (WAL), and is persisted into the distributed file system. The WAL records all changes to data in HBase, and in the case of a *HRegionServer* crash ensures that the changes to

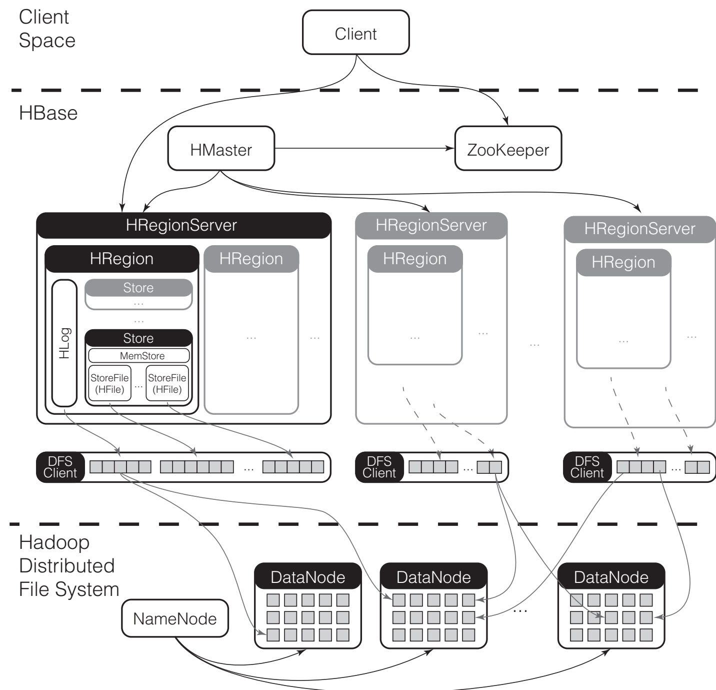


Fig. 2: HBase architecture

the data can be replayed. Stores in a region contain an in-memory data store (*MemStore*) and a persistent data stores (*HFiles*, that are persisted into the distributed file system) *HFiles* are local to each region, and used for actual data storage. The ZooKeeper cluster is responsible of providing the client with the information about both the *HRegionServer* and the *HRegion* hosting the row the client is looking up for. This information is cached at the client side, so that a direct communication could be directly setup for the next times without querying the *HMaster*. When an *HRegionServer* receives a write request, it sends the request to a specific *HRegion*. Once the request is processed, data is first written into the *MemStore* and when certain threshold is met, the *MemStore* gets flushed into an *HFile*.

2.3 HBase vs. HDFS

HDFS is the primary distributed storage used by Hadoop applications as it is designed to optimize distributed processing of multi-structured data. It is well suited for distributed storage and distributed processing using commodity hardware. It is fault tolerant, scalable, and extremely simple to expand. HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency, which makes it neither suitable nor optimized for atomic model operations. HBase is, on the other hand, a better choice for low-latency access. Moreover, HDFS resources cannot be written concurrently by multiple writers without locking and this results in locking delays. Also writes are always made at the end of the file. Thus, writing in the middle of a file (e.g. changing a value of a feature) involves rewriting the whole file, leading to more significant delays. On the contrary, HBase allows fast random reads and writes. HBase is row-level atomic, i.e. inter-row operations are not atomic, which might lead to a dirty read depending on the data model used. Additionally, HBase only provides five basic data operations (namely, *Get*, *Put*, *Delete*, *Scan*, and *Increment*), meaning that complex operations are delegated to the client appliaction (which, in turn, must implement them as a combination of these simple operations).

3 NeoEMF/HBase

Figure 3 shows the high-level architecture of our proposal for the EMF framework. It consists in a transparent persistence manager behind the model-management interface, so that tools built over the modeling framework would be unaware of it. The persistence manager communicates with the underlying database by a driver, and supports a pluggable caching strategy. In particular we implement the NeoEMF/HBase tool as a persistence manager for EMF on top of HBase and ZooKeeper. NeoEMF also supports other technologies, such as an embedded graph backend [8] and an embedded key-value store [12].

This architecture guarantees that the solution integrates well with the modeling ecosystem, by strictly complying with the EMF API. Additionally, the

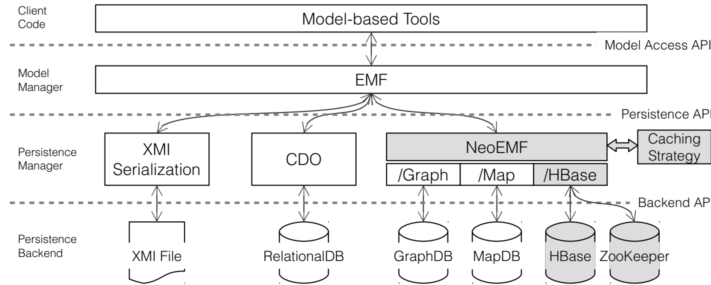


Fig. 3: Overview of the model-persistence framework

APIs are consistent between the model-management framework and the persistence driver, keeping the low-level data structures and code accessing the database engine completely decoupled from the modeling framework high level code. Maintaining these uniform APIs between the different levels allows including additional functionality on top of the persistence driver by using the decorator pattern, such as different cache levels.

NeoEMF/HBase offers lightweight on-demand loading and efficient garbage collection. Model changes are automatically reflected in the underlying storage, making changes visible to all the clients. To do so, first we decouple dependencies among objects by assigning a *unique identifier* to all model objects, and then:

- To implement *lightweight on-demand loading and saving*, for each live model object, we create a lightweight delegate object that is in charge of on-demand loading the element data and keeping track of the element's state. Delegates load and save data from the persistence backend by using the object's unique identifier.
- For *efficient garbage collection* in the *Java Runtime Environment*, we avoid to maintain hard Java references among model objects, so that the garbage collector can deallocate any model object that is not directly referenced by the application.

3.1 Map-based data model

We have designed the underlying data model of NeoEMF/HBase to minimize the data interactions of each method of the EMF model access API. The design takes advantage of the unique identifier defined in the previous section to flatten the graph structure into a set of key-value mappings.

Fig. 4a shows a small excerpt of a possible *Java* metamodel that we will use to exemplify the data model. This metamodel describes *Java* programs in terms of *Packages*, *ClassDeclarations*, *BodyDeclarations*, and *Modifiers*. A *Package* is a named container that groups a set of *ClassDeclarations* through the *ownedElements* composition. A *ClassDeclaration* contains a *name* and a set of

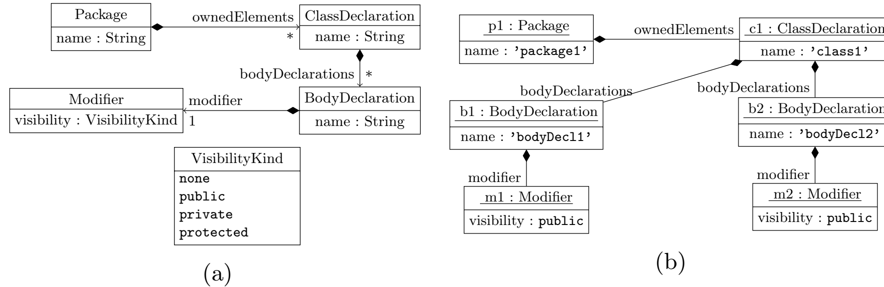


Fig. 4: Excerpt of the *Java* metamodel (4a) and sample instance (4b)

BodyDeclarations. Finally, a *BodyDeclaration* contains a *name*, and its *visibility* is described by a single *Modifier*.

Fig. 4b shows a sample instance of the *Java* metamodel, i.e., a graph of objects conforming with the metamodel structure. The model contains a single *Package* (`package1`), containing only one *ClassDeclaration* (`class1`). The *Class* contains the `bodyDecl1` and `bodyDecl2` *BodyDeclarations*. Both of them are `public`.

NeoEMF/HBase uses a single table with three column families to store models' information: (i) a *property column family*, that keeps all objects' data stored together; (ii) a *type column family*, that tracks how objects interact with the meta-level (such as the *instance of* relationships); and (iii) a *containment column family*, that defines the models' structure in terms of containment references. Table 1[3] shows how the sample instance in Fig. 4b is represented using this structure.

As Table 1 shows, row keys are the object *unique identifier*. The `property` *column family* stores the objects' actual data. As it can be seen, not all rows have a value for a given column. How data is stored depends on the *property type* and *cardinality* (i.e., upper bound). For example, values for single-valued attributes (like the *name*, which stored in the `name` column) are directly saved as a single literal value; while values for many-valued attributes are saved as an array of single literal values (Fig. 4b does not contain an example of this). Values for single-valued references, such as the *modifier* containment reference from *BodyDeclaration* to *Modifier*, are stored as a single value (corresponding to the identifier of the referenced object). Examples of this are the cells for ⟨`b1`, `modifier`⟩ and ⟨`b2`, `modifier`⟩ which contain the values `'m1'` and `'m2'` respectively. Finally, multi-valued references are stored as an array containing the literal identifiers of the referenced objects. An example of this is the *bodyDeclarations* containment reference, from *ClassDeclaration* to *BodyDeclaration*, that for the case of the *c1* object is stored as { `'b1'`, `'b2'` } in the ⟨`c1`, `bodyDeclarations`⟩ cell.

Structurally, EMF models are trees (a characteristic inherited from its XML-based representation). That implies that every non-volatile *object* (except the

[3] Actual rows have been split for improved readability

Table 1: Example instance stored as a sparse table in HBase

	property					
Key	eContents	name	ownedElements	bodyDeclarations	modifier	visibility
'ROOT'	'p1'					
'p1'		'package1'	{ 'c1' }			
'c1'		'class1'		{ 'b1', 'b2' }		
'b1'		'bodyDecl1'			'm1'	
'b2'		'bodyDecl'			'm2'	
'm1'						'public'
'm2'						'public'

	containment		type	
Key	container	feature	nsURI	EClass
'ROOT'			'http://java'	'RootEObject'
'p1'	'ROOT'	'eContents'	'http://java'	'Package'
'c1'	'p1'	'ownedElements'	'http://java'	'ClassDeclaration'
'b1'	'c1'	'bodyDeclarations'	'http://java'	'BodyDeclaration'
'b2'	'c1'	'bodyDeclarations'	'http://java'	'BodyDeclaration'
'm1'	'b1'	'modifiers'	'http://java'	'Modifier'
'm2'	'b2'	'modifiers'	'http://java'	'Modifier'

root *object*) must be contained within another *object* (i.e., referenced from another *object* via a containment *reference*). The `containment` *column family* maintains a record of which is the container for every persisted object. The `container` column records the identifier of the container object, while the `feature` column records the name of the *property* that relates the container object with the child object (i.e., the object to which the row corresponds). Table 1 shows that, for example, the container of the *Package p1* is `ROOT` through the *eContents* property (i.e., it is a root object and is not contained by any other object). In the next row we find the entry that describes that the *Class c1* is contained in the *Package p1* through the *ownedElements property*.

The `type` *column family* groups the type information by means of the `nsURI` and `EClass` columns. For example, the table specifies the element *p1* is an instance of the *Package* class of the *Java* metamodel (that is identified by the `http://java` *nsURI*).

3.2 ACID properties

NeoEMF/HBase is designed as a simple persistence layer that maintains the same semantics as the standard EMF. Modifications in models stored using NeoEMF/HBase are directly propagated to the underlying storage, making changes visible to all possible readers immediatly. As in standard EMF, no transactional support is explicitly provided, and as such, ACID properties [13] (Atomicity, Consistency, Isolation, Durability) are only supported at the object level:

Atomicity — Modifications on object's properties are atomic. Modifications involving changes in more than one object (e.g. bi-directional references), are not atomic.

Consistency — Modifications on object's properties are always consistent using a compare-and-swap mechanism. In the case of modifications involving changes in more than one object, consistency is only guaranteed when the model is modified to grow monotonically (i.e., only new information is added, and no already existing data is deleted nor modified).

Isolation — Reads on a given object always succeeds and always give a view of the object's latest valid state.

Durability — Modifications on a given object are always reflected in the underlying storage, even in the case of a *Data Node* failure, thanks to the replication capabilities provided by HBase.

These properties allow the use of NeoEMF/HBase as the persistence backend for distributed and concurrent model transformations, since reads in the source model are consistent and always success; and the creation of the target model is a building process that creates a model that grows monotonically.

4 Conclusion and Future Work

In this paper we have outlined NeoEMF/HBase, an on-demand, memory-friendly persistence layer for distributed and decentralized model persistence. Decentralized model persistence is useful in scenarios where multiple clients may access models when performing distributed computing. NeoEMF/HBase is built on top of HBase, a distributed, scalable, versioned and non-relationals big data store, specially designed to run together with Apache Hadoop.

NeoEMF/HBase takes advantage of the HBase properties by using a simple data model that minimizes data dependencies among stored objects. More specifically, NeoEMF/HBase exploits the row-locking mechanisms of HBase to provide limited ACID properties without requiring the use of transactions, which may increase latency in model operations. NeoEMF/HBase provides ACID properties at the object level, and guarantees that: (i) object queries always return the last valid state of an object; (ii) attribute modifications always succeed and produce a consistent model; and (iii) modifications of references which make the model grow monotonically always succeed and produce a consistent model.

Previous work [12] shows that key-value stores present clear benefits for storing big models, since model operations cost remains constant when models size grows. However, NeoEMF/HBase still lacks of a thorough performance evaluation. Hence, immediate future work is focused in the development of an evaluation benchmark. In this sense, we pursue to determine how the latency introduced by HBase – specially on write operations – affects the overall performance.

Additionally, a more advanced locking mechanism allowing arbitrary object locks will be implemented. Such a mechanism will provide multi-object ACID properties to the framework, allowing client applications to implement the synchronization logic to perform arbitrary, distributed and concurrent modifications.

Acknowledgments

This work is partially supported by the MONDO (EU ICT-611125) project.

References

1. CDO DB Store (2014), http://wiki.eclipse.org/CDO/DB_Store
2. CDO Hibernate Store (2014), http://wiki.eclipse.org/CDO/Hibernate_Store
3. CDO Model Repository (2014), http://www.eclipse.org/cdo/
4. CDO MongoDB Store (2014), http://wiki.eclipse.org/CDO/MongoDB_Store
5. CDO Objectivity Store (2014), http://wiki.eclipse.org/CDO/Objectivity_Store
6. Eclipse Modeling Framework (2014), http://www.eclipse.org/modeling/emf/
7. Hadoop Distributed File System (2015), http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
8. Benelallam, A., Gómez, A., Sunyé, G., Tisi, M., Launay, D.: Neo4EMF, A Scalable Persistence Layer for EMF Models. In: Modelling Foundations and Applications, Lecture Notes in Computer Science, vol. 8569, pp. 230–241. Springer (2014)
9. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7. pp. 15–15. OSDI '06, USENIX Association, Berkeley, CA, USA (2006), http://dl.acm.org/citation.cfm?id=1267308.1267323
10. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Commun. ACM. vol. 51, pp. 107–113. ACM, NY, USA (2008)
11. Garrison, G., Kim, S., Wakefield, R.L.: Success factors for deploying cloud computing. Commun. ACM 55(9), 62–68 (Sep 2012)
12. Gómez, A., Tisi, M., Sunyé, G., Cabot, J.: Map-based transparent persistence for very large models. In: Egyed, A., Schaefer, I. (eds.) Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science, vol. 9033, pp. 19–34. Springer Berlin Heidelberg (2015), http://dx.doi.org/10.1007/978-3-662-46675-9_2
13. Haerder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Comput. Surv. 15(4), 287–317 (Dec 1983), http://doi.acm.org/10.1145/289.291
14. Khurana, A.: Introduction to HBase Schema Design. ;login: The Usenix Magazine 37(5), 29–36 (2012), https://www.usenix.org/publications/login/october-2012-volume-37-number-5/introduction-hbase-schema-design
15. Markus Scheidgen: EMF fragments (2014), https://github.com/markus1978/emf-fragments/wiki
16. Scheidgen, M., Zubow, A.: Map/Reduce on EMF Models. In: Proceedings of the 1st International Workshop on Model-Driven Engineering for High Performance and CLoud Computing. pp. 7:1–7:5. MDHPCL '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2446224.2446231
17. The Apache Software Foundation: Apache Hadoop (2015), http://hadoop.apache.org/
18. The Apache Software Foundation: Apache HBase (2015), http://hbase.apache.org/
19. The Apache Software Foundation: Apache ZooKeeper (2015), https://zookeeper.apache.org/

Parallel In-place Model Transformations with LinTra

Loli Burgueño[1], Javier Troya[2], Manuel Wimmer[2], and Antonio Vallecillo[1]

[1] Universidad de Málaga, Atenea Research Group, Málaga, Spain
{loli,av}@lcc.uma.es
[2] Vienna University of Technology, Business Informatics Group, Vienna, Austria
{troya,wimmer}@big.tuwien.ac.at

Abstract. As software systems have grown large and complex in the last few years, the problems with which Model-Driven Development has to cope have increased at the same pace. In particular, the need to improve the performance and scalability of model transformations has become a critical issue. In previous work we introduced LinTra, a model transformation platform for the parallel execution of out-place model transformations. Nevertheless, in-place model transformations are required in several contexts and domains as well. In this paper we discuss the fundamentals of in-place model transformations in the light of their parallel execution and provide LinTra with an in-place execution mode.

Keywords: In-place Model Transformations, Performance, Scalability, Linda

1 Introduction

Model transformations (MT) are gaining acceptance as model-driven techniques are becoming commonplace [1]. While models capture the views on systems for particular purposes and at given levels of abstraction, model transformations are in charge of the manipulation, analysis, synthesis, and refinement of the models [2].

So far the community has mainly focused on the correct implementation of a model transformation, according to its specification [3–8], although there is an emergent need to consider other (non-functional) aspects such as performance, scalability, usability, maintainability and so forth [9]. In particular, the study of the performance of model transformations is gaining interest as very large models living on the cloud have to be transformed as well. Consequently, new approaches to parallelize model transformations are starting to appear [10–13]. Following this path, in a previous work we introduced LinTra [11, 14], a model transformation engine, together with its implementation in Java called jLinTra. LinTra encapsulates all the concurrent mechanisms needed for the parallel execution of model transformations, and in particular users do not need to worry about threads and their synchronization. The engine is based on Linda [15], a mature coordination language for parallel processes.

So far, LinTra only permitted out-place model transformations. In this kind of transformations, input and output models often conform to different metamodels and output models are created from scratch. However, there are many situations in which we need to evolve models, instead of creating them anew. For instance, when migrating and modernizing software using model-driven engineering (MDE) approaches [16, 17], (i)

software is reverse-engineered to obtain a model representation of the system, (ii) modernization patterns are applied on the model level, and (iii) the modernized model is translated back into code. Modernization at model level is typically achieved using in-place model transformations, where the initial model is evolved until the final target model is obtained. Models which are reverse-engineered from large systems may be huge, thus high-performing in-place transformation engines are needed. For this reason, in this paper we extend our LinTra language with an in-place semantics.

This paper is structured as follows. Section 2 shortly introduces LinTra and our reference non-recursive in-place semantics. Section 3 shows how LinTra realizes its in-place semantics, while Section 4 illustrates the benefits of parallel in-place transformations. Finally, Section 5 discusses related work before we conclude the paper in Section 6 with an outlook on future work.

2 Background

Previous Work on LinTra. LinTra [11, 14] is a parallel model transformation engine that encapsulates all the concurrent mechanisms, so that users do not need to worry about threads and their synchronization. It uses the Blackboard paradigm [18] to store the input and output models, as well as the required data to keep track of the MT execution that coordinates the agents that are involved in the transformation process. One of the keys of LinTra is the model and metamodel representation, where we assume that entities in the model are independent from each other. Thus, every entity is assigned an identifier, which is used to reference objects and to represent relationships between them. Relationships between entities are represented by storing in the source entity the identifier(s) of its target entity(ies).

In out-place model transformations, there might be dependencies between rules because the element(s) created in a rule are needed to initialize some properties of the elements created by other rules. In LinTra, traceability is implemented implicitly using a bidirectional function that receives as a parameter the entity identifier of the input model and returns the identifier of the output entity, regardless whether the output entities have already been created or not. This means that LinTra does not store information about the traces explicitly; thus, the performance is not affected by the access to memory and the search for trace information.

In order to carry out the transformation process in parallel, LinTra uses the Master-Slave design pattern [18]. The master's job is to launch slaves and coordinate their work. Each slave is in charge of applying the transformation to submodels of the input model (i.e., partitions) as if each partition is a complete and independent model. Since in LinTra's out-place mode the complete input model is always available, in case the slaves have data dependencies with elements that are not in the submodels they were assigned, they only have to query the Blackboard to retrieve them.

Non-recursive In-place Transformations. In-place transformations specify how the input model evolves to obtain the output one, i.e., how the input model has to change. There are two kinds of in-place model transformation strategies, non-recursive and recursive, depending on whether recursive matching takes place or not. By *recursive matching* we understand that the matches of rules are not solely computed based

on the initial input model but on the current model state which probably has been modified by previous application of rules. This is the typical strategy followed in graph or rewriting systems, where a set of rules modifies the state of a configuration of objects (representing the model) one-by-one. Thus, after the application of each rule, the state of the system is changed, and subsequent rules will be applied on the system on this new state. Therefore, the transformation navigates the target model, which is continuously updated by every executed rule.

Regarding *non-recursive matching*, it shares some characteristics with out-place transformations. In this strategy, there is one input model which is used to directly compute the output model without considering intermediate steps. This is the reason why we chose to follow a non-recursive approach for the LinTra in-place mode. Our decision was also inspired by the ATL *refining mode* [19, 20], used to implement in-place transformations. ATL supports both out-place and in-place modes. In both execution modes, source models are read-only and target models are write-only. This is an important detail that significantly affects the way in which ATL works in refining mode. Indeed, ATL in-place mode does not execute transformations as these are executed in graph or rewriting systems, as explained in detail in [21]. Thus, we follow as well non-recursive matching in LinTra where rules always read (i.e., navigate) the state of the source model, which remains unchanged during all the transformation execution.

3 In-place Model Transformations with LinTra

LinTra follows a non-recursive approach for executing in-place transformations, as the ATL refining mode does. In this section we discuss some semantic issues that might occur in rule-based in-place model transformations in general as they are indeed highly relevant for the parallel execution of in-place transformations.

3.1 Atomic Transformation Actions

When executing a non-recursive in-place transformation, the first decision concerns the elements for which the transformation does not specify what to do. We could either decide to exclude them from the target model or to include them as they are. In jLinTra we decided for the second option, which implies that if we want to exclude objects in the target model, the transformation will have to explicitly remove them. Thus, after the input model is loaded, and once the transformation phase starts, an initialization phase is needed where the identity transformation is applied so that the target area contains a copy of the input model.

After the model is copied, in the following we explain the three operations that may be applied to it: deletion of elements, creation of new elements, and modification of existing elements.

Elements Deletion. When an element is deleted, the outgoing relationships from such element to others are deleted too, since such information is stored as attributes in the deleted element. However, the situation is different when the deleted element has incoming relationships. In such case, the information about relationships to the deleted element is stored in the attributes of other elements. In this case, we can distinguish

two different semantics. Either all the incoming relationships are deleted, for which the engine needs to traverse the whole model searching for relationships pointing to the deleted element, or they are not deleted, causing dangling references and, consequently, an inconsistent model. In the former option, we need to keep track of all the deleted elements, so that the traversal is realized only once as the last step of the transformation. The latter option is useful in order to make the user aware that he/she is removing an element by mistake. LinTra permits both behaviors, since it is aimed at offering a flexible implementation.

Elements Creation. If the developer wants to create a new element, he/she has to create the instance and set its attributes and relationships. In case of bidirectional relationships, there are two alternatives: *(i)* the opposite reference is created automatically, or *(ii)* the creation of the opposite relationship must be explicitly specified by the developer. We permit both behaviors.

Elements Updates. Updating an attribute or an ongoing unidirectional relationship of an element is trivial, since the transformation only has to change the corresponding attribute of the updated element. However, there are again two choices when updating a relationship which is bidirectional, since the previous target element of the relationship would still have a relationship to the updated element unless something is done. Thus, *(i)* the relationship from the previous target element should be automatically removed and a new relationship from the new pointed element to the updated element should be automatically created, or *(ii)* the developer has to specify explicitly in the transformation that the corresponding relationships are removed and created respectively. Again, we permit both alternative behaviors.

3.2 Confluence conflicts

Confluence conflicts typically occur when two rules are applied to the same part of the model and they treat it differently [22]. Thus, the resulting model may vary depending on the order in which those rules are applied. The application of a rule can conflict with the application of another rule in four different ways. Let us explain them for the ATL refining mode which acts as blueprint for the LinTra in-place transformation strategy. For the explanations, let us imagine a transformation for reverse engineering Java code.

Update/Update. Imagine that a rule sets the public variables to private and capitalizes the name of the ones that are private. This case is not a problem for the confluence of non-recursive in-place transformations since only the source model provided by the user is read—the changes done by the rule that changes the visibility are not visible to the rule that capitalizes the names of the variables. On the contrary, if a rule sets the visibility of the variables to private and another rule sets them to public, the transformation may not be confluent. A possible way to prevent this situation is to force the precondition of the rules to be exclusive, which leads to non-overlapping matches. This was the solution adopted by ATL concerning the declarative part. Nevertheless, it is easy to fool ATL by using the imperative part, which is executed after the declarative part of the rule.

Delete/Update. Suppose that a rule sets the visibility of the variables to private and another rule removes all the variables. The situation is similar to the second case we presented for the conflict *Update/Update*. The two rules are a conflicting pair, thus

the language should prevent this situation from happening or should establish the behaviour of the transformation. Again, it is possible to produce this case in ATL by using the imperative part to set the visibility and writing a declarative rule that removes the variables. Both rules are executed so that the visibility is changed and the variables are removed. As a result, the variables are not present in the resulting model. Apparently, the objects are removed in a later execution phase, after having done all the updates and creations specified in the declarative and imperative parts.

Produce/Forbid. Imagine that a rule adds a variable to a class and another rule removes all the empty classes (classes with no variables) from the model. The first rule is producing an additional structure that is forbidden by the precondition of the second rule. Once again, the order in which the rules are executed influences the result. This time, if we try to implement this transformation with ATL using the imperative part of a rule to add the variables and a declarative rule to remove the empty classes, both rules are applied but the transformation does not fulfil the purpose for which it was written (since only the source model is read). As a result, the classes are removed but the newly created variables remain in the model without any container.

Delete/Use. This conflict appears when a rule deletes elements that produce a match with another rule. Thus, it is the opposite case to *Produce/Forbid*. Depending on the order in which the rules are executed, the transformation is able to execute a higher or lower number of rules.

We have illustrated the conflicts that may appear between rules and how ATL tries to solve them using non-overlapping matches, how they can be avoided or produced, and which is the final result of the execution. Enforcing to have non-overlapping rules is not the only solution; another possibility is to statically detect the conflicting rules using the critical pair analysis approach [23], and subsequently, to deal with the conflicts making use of layers which is also implicitly done in ATL by using different phases in the transformation execution.

As jLinTra is realized as an internal transformation language embedded in Java, we have opted for not imposing any restriction. Thus, our solution is completely flexible with respect to rule executions. The idea is that high-level model transformation languages (such as ATL [24], ETL [25], or QVTO [26]) are compiled to an extended set of the primitives [27] that automatically compile to jLinTra. In these primitives the rule scheduling is already given (the layers and the rules that belong to each layer). Therefore, in case that the critical pair analysis is needed, it can be done statically during the compilation process from the high-level model transformation language to the LinTra primitives.

4 Evaluation

To evaluate our approach we performed an experimental study concerning a transformation which, in reverse engineered Java applications, removes all the comments, changes the attributes from public to private and creates the getters and setters.

4.1 Research Questions

The study was performed to quantitatively assess the quality of our approach by measuring the runtime performance of the transformations. We aimed to answer the following research questions (RQs):

1. *RQ1—Parallel vs. sequential in-place transformations*: Is the parallel execution of in-place transformations faster in terms of execution times compared to using the state-of-the-art sequential execution engines? And if there is a positive impact, what is the speedup with respect to the used number of cores for the parallel transformation executions?
2. *RQ2—Parallel in-place vs. parallel out-place transformations*: Is the parallel execution of in-place transformations faster in terms of execution time compared to using their equivalent out-place transformations?

4.2 Experiment Setup

To evaluate our approach, we use an experiment in which Java models are obtained from Java code using MoDISCO [17]. The Java metamodel has a total of 125 classes from which 15 are abstract, 166 relationships among them and 5 enumeration types. As source model we have selected the complete Eclipse project, containing $4,357,774$ entities. In order to assess how the transformation scales up with this kind of input, we generated 11 smaller sample source models (with subsets of the Eclipse project) ranging from $100,000$ elements to the complete model.

We apply an extended version of the Public2Private transformation – the original one is available in the ATL Zoo[3] – that changes the visibility of every public variable to private and creates the corresponding getter and setter methods. In addition, the transformation also removes all the comments contained in the code. All artifacts can be downloaded from our website[4].

Let us show the effects of this transformation with an example. Listing 1.1 shows the Java code that declares a class called *MyClass*, a public attribute *name* and the class's constructor. The code contains some comments too. After applying the transformation, the Java code that the model represents should look like the fragment presented in Listing 1.2.

Listing 1.1. Code to be refactored

```
1 public class MyClass {
2     //Declaration of variable called name
3     public String name; /* This variable contains the name */
4     public MyClass() { /* Description @param ... */  ... }
5 }
```

Listing 1.2. Refactored code

```
1 public class MyClass {
2     private String name;
3     public String getName() { return name; }
4     public void setName(String name) { this.name = name; }
5     public MyClass(){ ... }
6 }
```

[3] http://www.eclipse.org/atl/atlTransformations/

[4] http://atenea.lcc.uma.es/index.php/Main_Page/Resources/LinTra

An excerpt of the code corresponding to the rules in jLinTra is shown in Listing 1.3. As stated in Section 2, every slave is in charge of transforming a chunk of the model. For efficiency reasons the changes are made permanent once the whole chunk has been transformed. In order to keep the temporary changes the structures `deletedElems`, `modifiedElems` and `createdElems` (lines 2, 8 and 9) are needed.

Listing 1.3. jLinTra transformation

```
1  if (ie instanceof Comment){
2      deletedElems.add(ie); //Delete Comment
3  } else if (ie instanceof FieldDeclaration){
4      String modId = ((FieldDeclaration) ie).getModifier();
5      Modifier mod = (Modifier) srcArea.read(modId);
6      String visibility = mod.getVisibility();
7      if (visibility.equals(PUBLIC)){
8          mod.setVisibility(PRIVATE); modifiedElems.add(mod); // Modify visibility
9          ... createdElems.add(...); // Create getters and setters }
10 }
```

We have run all our experiments on a machine whose operating system is Ubuntu 12.04 64 bits with 11.7 Gb of RAM and 2 processors with 4 hyperthreaded cores (8 threads) of 2.67GHz each. We discuss the results obtained for the different transformations after executing each one 10 times for every input model and having discarded the first 5 executions as the VM has a warm-up phase where the results might not be optimal. The Eclipse version is Luna. The Java version is 8, where the JVM memory has been increased with the parameter *-Xmx11000m* in order to be able to allocate larger models in memory.

4.3 Performance Experiments

The in-place transformation described before has been implemented and executed in jLinTra and in ATL, for which we have used the EMFTVM[5]. We have also developed an out-place transformation version in jLinTra in order to compare its performance with the proposed in-place version. Table 1 shows in its left-most column the number of entities of the source models of the transformation. The second, third, and fourth columns correspond to the execution times (in seconds) obtained for ATL and jLinTra (using the in-place and out-place modes), respectively. Note that we have only taken into account the time of the execution of the transformation, meaning that we do not consider the time used for loading the models into memory, nor the time used to serialize them to the disk. The fifth column presents the speedup of jLinTra with respect to ATL. We can see that the speedup is not constant: it grows with the size of the model, reaching a value of 955.23 for the complete model, meaning that value that jLinTra is 955.23 times faster than ATL for this concrete case. Finally, column six shows the speedup of the in-place and out-place modes of LinTra, where we can see that the in-place model transformation is on average 1.81 times faster than its out-place version.

We already mentioned in Section 3 that an initialization phase where the input model is copied to the target area is needed. However, if we moved that process to the loading phase so that both the source and target areas were loaded at the same time, we would only pay a minimum price (an overhead of 5% in the loading phase) and the

[5] `https://wiki.eclipse.org/ATL/EMFTVM`

	ATL	LinTra		Speedups	
No. elements	**EMFTVM**	**In-place (LI)**	**Out-place (LO)**	*LI–EMFTVM*	*LI–LO*
0.1$\times 10^6$	2.40	0.11	0.19	21.23	1.72
0.2$\times 10^6$	12.04	0.29	0.36	41.88	1.25
0.5$\times 10^6$	65.06	0.73	0.98	89.06	1.34
1.0$\times 10^6$	371.41	1.29	2.38	287.34	1.84
1.5$\times 10^6$	1042.41	2.06	2.61	506.71	1.27
2.0$\times 10^6$	2030.82	2.99	5.63	678.16	1.88
2.5$\times 10^6$	2952.46	3.92	9.64	754.14	2.46
3.0$\times 10^6$	4156.69	5.13	8.82	809.92	1.72
3.5$\times 10^6$	5527.96	6.26	13.77	883.37	2.20
4.0$\times 10^6$	6737.97	7.57	15.20	890.70	2.01
Complete	7238.70	7.58	17.18	955.23	2.27

Table 1. Execution results and speedups.

performance in the transformation phase would be improved reaching a speedup of 3.89 w.r.t. the out-place mode and speedup of $1,195$ w.r.t. ATL.

Regarding the gain of in-place MTs in LinTra w.r.t. the number of cores involved in the transformation, the speedups of using only one core w.r.t. using four, eight, twelve and sixteen are 1.19, 1.62, 1.97, 3.24, respectively.

We also planned to execute and compare this transformation with the original ATL virtual machine. However, although it supports the refining mode it does not support the imperative block, which is applied in the particular transformation used in this study.

Regarding the out-place transformation developed in LinTra, it explicitly specifies that all elements that are not modified must be copied, together with their properties. The out-place transformation counts on $3,302$ lines of Java code (we generated the code for the identity transformation using Xtend [6] and adapted the corresponding code to fit the needs of the Public2Private transformation), while the in-place transformation has only 194 lines.

For answering the two RQs stated above, we can first conclude that the parallel execution of in-place transformations reduces the execution time compared to using sequential execution and that the execution time can be further improved by adding more cores. Second, for typical in-place transformation problems, in-place transformation implementations are more efficiently executed than their equivalent out-place transformations using parallelization in both executions.

4.4 Threats to Validity

In this section, we elaborate on several factors that may jeopardize the validity of our results.

Internal validity – Are there factors which might affect the results of this experiment? The performance measures we have obtained directly relate to the experiment we have used for the evaluation. Therefore, if we had used different experiments other than the Public2Private transformation then the speedups between the executions of the different implementations would have probably been different. Besides, we have generated 11 smaller sample source models. Should we have generated different models,

[6] `https://eclipse.org/xtend/`

the results in Table 1 would also be different. As another threat, we have decided to use the executions after the 5th one in order to avoid the possible influence of the VM warming-up phase. However, if after the 5th execution the VM has not finished warming up, our results are then influenced. Finally, we are quite confident that we have correctly written the equivalent transformation in ATL due to our expertise with such language. Nevertheless, there may exist tiny differences which may have an influence on the execution times.

External validity – To what extent is it possible to generalize the findings? As a proof of concept of our approach, we have compared the execution times of our approach with the ATL implementation executed with the EMFTVM engine. We have chosen ATL for the comparison study because we have enriched LinTra with the same in-place semantics that ATL has. Therefore, since our study only compares LinTra and ATL, our results cannot be generalized for all non-recursive engines.

5 Related Work

In this paper, we focus on in-place model transformations running in batch mode. However, to deal with large models, orthogonal techniques may be applied as well. Especially, two scenarios have been discussed in the past in the context of speeding-up model transformation executions. First, if an output model already exists from a previous transformation run for a given input model, only the changes in the input model are propagated to the output model. Second, if only a part of the output model is needed by a consumer, only this part is produced while other elements are produced just-in-time. For the former scenario, *incremental* transformations [28–30] have been introduced, while for the latter *lazy* transformations [31] have been proposed.

An important line of research for executing transformations in parallel is the work on critical pair analysis [22] from the field of graph transformations as discussed in Section 3. This work has been originally targeted to transformation formalisms that do have some freedom for choosing in which order to apply the rules. Rules that are not in an explicit ordering are considered to be executed in parallel if no conflict is statically computed. However, most existing execution engines follow a pseudo-parallel execution of the rules, but there are already some approaches emerging which consider the execution of graph transformations in a recursive way on top of multi-core platforms [12, 13, 32]. A closer comparison concerning the commonalities and differences of recursive and non-recursive in-place semantics concerning parallelism is considered as subject for future work.

The performance of model transformations is considered as an integral research challenge in MDE [33]. For instance, Amstel et al. [9] considered the runtime performance of transformations written in ATL and in QVT. In [34], several implementation variants using ATL, e.g., using either imperative constructs or declarative constructs, of the same transformation scenario have been considered and their different runtime performance has been compared. However, these works only consider the traditional execution engines following a sequential rule application approach. One line of work we are aware of dealing with the parallel execution of ATL transformations is [35], where Clasen et al. outlined several research challenges when transforming models in

the cloud. A follow-up work is presented in Tisi et al. [36] where a parallel transformation engine for ATL is presented. However, they only consider out-place transformations while we tackled the parallel execution of in-place transformations.

6 Conclusion and Future Work

We have presented an extension for LinTra that allows the parallel execution of in-place model transformations. We have shown with experiments that the performance is improved w.r.t. other in-place MT engines and that in cases where in-place transformations can be achieved also by means of out-place transformations, the in-place transformations provide better performance and usability.

There are several lines of future work. First, we plan to provide a new in-place execution mode that supports recursive matchings. Second, we plan to extend our set of primitives so that in-place transformations written in any MT language may be compiled to and executed with LinTra.

Acknowledgments This work is funded by the Spanish Research Projects TIN2011-23795 and TIN2014-52034-R, by the European Commission under ICT Policy Support Programme (grant no. 317859), and by the Christian Doppler Forschungsgesellschaft and the BMWFW, Austria.

References

1. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers (2012)
2. Lucio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G., Syriani, E., Wimmer, M.: Model transformation intents and their properties. SoSyM (2014) 1–38
3. Gogolla, M., Vallecillo, A.: *Tract*able Model Transformation Testing. In: Proc. of ECMFA. (2011) 221–235
4. Amrani, M., Lúcio, L., Selim, G., Combemale, B., Dingel, J., Vangheluwe, H., Traon, Y.L., Cordy, J.R.: A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In: Proc. of VOLT Workshop @ ICST. (2012) 921–928
5. Vallecillo, A., Gogolla, M., Burgueño, L., Wimmer, M., Hamann, L.: Formal Specification and Testing of Model Transformations. In: Proc. of SFM. (2012) 399–437
6. Sánchez Cuadrado, J., Guerra, E., de Lara, J.: Uncovering errors in ATL model transformations using static analysis and constraint solving. In: Proc. of ISSRE. (2014) 34–44
7. Wimmer, M., Burgueño, L.: Testing M2T/T2M Transformations. In: Proc. of MODELS. (2013) 203–219
8. Burgueno, L., Troya, J., Wimmer, M., Vallecillo, A.: Static Fault Localization in Model Transformations. IEEE Transactions on Software Engineering **41**(5) (2015) 490–506
9. van Amstel, M., Bosems, S., Kurtev, I., Pires, L.F.: Performance in Model Transformations: Experiments with ATL and QVT. In: Proc. of ICMT. (2011) 198–212
10. Tisi, M., Perez, S.M., Choura, H.: Parallel Execution of ATL Transformation Rules. In: Proc. of MODELS. (2013) 656–672
11. Burgueño, L., Troya, J., Wimmer, M., Vallecillo, A.: On the Concurrent Execution of Model Transformations with Linda. In: Proc. of BigMDE Workshop @ STAF. (2013) 3:1–3:10
12. Imre, G., Mezei, G.: Parallel Graph Transformations on Multicore Systems. In: Proc. of MSEPT. (2012) 86–89

13. Krause, C., Tichy, M., Giese, H.: Implementing Graph Transformations in the Bulk Synchronous Parallel Model. In: Proc. of FASE. (2014) 325–339
14. Burgueño, L.: Concurrent Model Transformations based on Linda. In: Doctoral Symposium @ MODELS. (2013) 9–16
15. Gelernter, D., Carriero, N.: Coordination Languages and Their Significance. Commun. ACM **35**(2) (1992) 97–107
16. Bergmayr, A., Brunelière, H., Canovas Izquierdo, J., Gorronogoitia, J., Kousiouris, G., Kyriazis, D., Langer, P., Menychtas, A., Orue-Echevarria, L., Pezuela, C., Wimmer, M.: Migrating Legacy Software to the Cloud with ARTIST. In: Proc. of CSMR. (2013) 465–468
17. Brunelière, H., Cabot, J., Dupe, G., Madiot, F.: MoDisco: A model driven reverse engineering framework. Information and Software Technology **56**(8) (2014) 1012–1032
18. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. Wiley (1996)
19. Tisi, M., Martínez, S., Jouault, F., Cabot, J.: Refining Models with Rule-based Model Transformations. Research Report RR-7582 (2011)
20. Wagelaar, D., Tisi, M., Cabot, J., Jouault, F.: Towards a General Composition Semantics for Rule-Based Model Transformation. In: Proc. of MODELS. (2011) 623–637
21. Troya, J., Vallecillo, A.: A Rewriting Logic Semantics for ATL. JOT **10** (2011) 5:1–29
22. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of Typed Attributed Graph Transformation Systems. In: Proc. of ICGT. (2002) 161–176
23. Mens, T., Taentzer, G., Runge, O.: Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. Electr. Notes Theor. Comput. Sci. **127**(3) (2005) 113–128
24. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming **72**(1-2) (2008) 31–39
25. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Transformation Language. In: Proc. of ICMT. (2008) 46–60
26. OMG: MOF QVT Final Adopted Specification. Object Management Group. (2005)
27. Burgueño, L., Syriani, E., Wimmer, M., Gray, J., Vallecillo, A.: LinTraP: Primitive Operators for the Execution of Model Transformations with LinTra. In: Proc. of BigMDE Workshop @ STAF. (2014) 23–30
28. Jouault, F., Tisi, M.: Towards Incremental Execution of ATL Transformations. In: Proc. of ICMT. (2010) 123–137
29. Razavi, A., Kontogiannis, K.: Partial evaluation of model transformations. In: Proc. of ICSE. (2012) 562–572
30. Szárnyas, G., Izsó, B., Ráth, I., Harmath, D., Bergmann, G., Varró, D.: IncQuery-D: A Distributed Incremental Model Query Framework in the Cloud. In: Proc. of MODELS. (2014) 653–669
31. Tisi, M., Perez, S.M., Jouault, F., Cabot, J.: Lazy Execution of Model-to-Model Transformations. In: Proc. of MODELS. (2011) 32–46
32. Bergmann, G., Ráth, I., Varró, D.: Parallelization of Graph Transformation Based on Incremental Pattern Matching. ECEASST **18** (2009) 1–15
33. 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: Proc. of BigMDE Workshop @ STAF. (2013) 2:1–2:10
34. Wimmer, M., Martínez, S., Jouault, F., Cabot, J.: A Catalogue of Refactorings for Model-to-Model Transformations. JOT **11**(2) (2012) 2:1–40
35. Clasen, C., Didonet Del Fabro, M., Tisi, M.: Transforming Very Large Models in the Cloud: a Research Roadmap. In: Proc. of CloudMDE Workshop @ ECMFA. (2012)
36. Tisi, M., Perez, S.M., Choura, H.: Parallel Execution of ATL Transformation Rules. In: MODELS. (2013) 656–672

Beyond Information Silos

Challenges in Integrating Industrial Model-based Data

Ali Shahrokni, Jan Söderberg

ali.shahrokni@systemite.se, jan.soderberg@systemite.se

Abstract.
Fragmented data management is commonplace for handling development and production data in large organizations today. While automated configuration and build mechanisms have made the implementation data more integrated, on higher levels of abstraction data is still fragmented into silos such as files or isolated databases. In the recent years, many companies, including a large number of automotive corporations, have realized the potential of integrating high-level data when it comes to saving cost, lowering time to market, increasing the efficiency of the processes, and increasing the quality of products. To achieve the integration, many initiatives on tool provider level and on standardization level have started. OSLC is one example of underlying architecture to enable integration of data that is stored in a distributed fashion. Systemite has more than 15 years of experience in central integrated data management, largely in the automotive industry. In this paper, we discuss our experience on some of the challenges that we have faced during the past years when it comes to integrating model-based and fine-grained data. Finally, we provide two approaches to move forward towards open ecosystems for tool interoperability.

Keywords: Model-based information management, tool interoperability, integrated information management, OSLC, meta modeling

1 Introduction

In most industries the predominant way of managing the complexity of product and process data during development is still the use of a file based approach, where data is fragmented into multitudes of computer files. As the amount of data increases and the nature of the products and processes become more complex, a new and separate type of complexity emerges from this file based approach that is not related to the complexity of the product and the organization. This complexity is a result of the scattered data. Finding the correct and up to date data in a large organization with thousands of documents can be a cumbersome task that can consume 14-30% of an engineer's time [1, 2, 3]. Finding the right version of the right data with a high confidence in the sea of data produced by parallel projects and historic data is often an impossible task. At the same time, using incorrect and obsolete data will produce inconsistencies that decrease the quality of the product or raise the verification costs.

As the products, processes, and development and manufacturing data becomes more complex, the cost of fragmenting data into files or other silos increases. This cost will rise as a result of difficulties to keep the data up to date in several files or several databases. At the same time organizations need to perform increasingly advanced analysis over the silo border, not the least as a result of external requirements such as safety standards or legal requirements. Another need from a process perspective is to have a realistic and correct view of the progress of large projects that is based on more reliable data than estimations. The need and the trend is clear that organizations are moving more towards generation of reports, analysis, visualization, and views of data instead of manually creating them.

An example that stresses the need for efficient data management is testing of large systems. The challenge here is that the results of several processes meet and touch in the testing activities; the test cases, the specifications that are used as references for the test cases, the artifacts comprising the System Under Test, the test environment and equipment to mention the most important. The realization of individual artifacts is typically uncorrelated, on the scale of time relevant for testing activities: new artifacts arrive every day, requirements and test cases constantly change, and regression tests have to be performed daily. The rate of change means that formal waterfall, baseline based configuration management is not effective nor efficient, since there will be many changes included within each iteration, between each formal baseline. This rate of change means that the test data representing the developed system will change frequently. This also means that defining configurations based on labels and performing check-out, check-in, and merge operations required for file based configuration management is not adequate.

These are just a few of the challenges that organizations are facing and all of these challenges point to the need for traversing the data not only inside one information silo, but over the borders of the silos (tools, files, databases). Therefore, there are many industrial initiatives to facilitate the interoperability between tools. The CRYSTAL project[1] is one of these initiatives. Crystal aims at standardizing interoperability between tools and integration of product and process information relying on the architecture of Open Services for Lifecycle Collaboration (OSLC)[2]. OSLC is an open community that exists to define specifications and mechanisms to integrate disjoint tools and workflows. The purpose of the integration is to save time and money through integrating data in different tools with the aim to keep the data more consistent and transparent over the borders of different tools. The underlying mechanism in OSLC is inspired by the web architecture in the sense that it is distributed and link based.

Systemite is a company with more than 15 years of experience in integrated data management. The company has an extensive experience in managing development data in the embedded and especially the automotive industry. Systemite provides a product called SystemWeaver is a model-based data management tool where all data that is stored or linked exists in a context and is integrated with its surrounding. The

---

1 http://www.crystal-artemis.eu/

2 http://open-services.net/

cornerstone and basic philosophy of SystemWeaver is to keep data integrated, traceable, and consistent while it provides a real-time and high performance platform for engineers and developers to enter and view data in its context in contrast to fragmenting data in different databases based on the type of data (for instance requirements and tests). At the same time the collaborative, high performance and scalability aspect are essential parts of the SystemWeaver platform. In our opinion, neglecting any of these aspects renders an industrial solution a liability rather than a resource as the quantity and complexity of data and organizations increase.

As the industry moves towards more integrated strategies for data management, we want to discuss some of the challenges in the area of data and share our experience on some of the obstacles that the industry needs to overcome to unleash more of the potentials of integrated data management.

2 Structured Data Management

The idea to manage system and product models in structured databases is not new. In the early 80's there were the CASE tools that relied on a centralized data dictionary or database server to manage and give access to the model data, which was typically developed according to some structured analysis and design methodology. (Example: Teamwork3). These tools were typically expensive and required the use of networked so called workstations, the high spec computers of the time. The high cost limited the use of the tools to specific, high margin industries such as military or aerospace. The arrival of PCs that offered low cost computing power coincided with the development of object oriented (OO) methods and tools such as Rational ROSE4. Frameworks of the 90s that relied on centralized data dictionaries, like the AD/Cycle of IBM or the open standard PCTE5 (Portable Common Tools Environment) were never used widely and were all ousted by the new object oriented tools running on the inexpensive PCs. A common feature of most of this type of tools, apart from being OO, was that they relied on storing and managing the model in computer files. This made the tools accessible when used in small projects, but the use of computer files also introduces the complications described in this paper. Throughout this period software development has remained a file based affair. Even modern ALM (Application Lifecycle Management) solutions manage the program source code as computer files, although information like change requests and configurations are managed with a database approach.

A common industrial method to deal with complexity is to minimize dependencies and interactions between subsystems, so that these can be largely developed independently. This practice of development on the sub system level leads to problems like sub-optimization, since analysis on the system level becomes difficult when all data is hidden on the sub system level. Any changes that need to break the subsystem barrier also tend to be very difficult to handle, calling for negotiation between sepa-

[3] Keysight Technologies
[4] IBM
[5] ISO/IEC 13719-1

rate development teams. Integration is not done until the subsystems have been developed, leading to late discovery of integration problems.

A special characteristic of configuration management of software is that the focus of the system generation (build) process is to compile and link single executables. It is indeed possible to extend the generation process to higher levels to collections of executables. However, since these higher level configurations do not have any specific semantics, from the perspective of the single executable and its software, they are rarely, if ever, used. Even when the development is according to concepts that support system level description and composability aspects, like AUTOSAR[6], the detailed development is today done on the subsystem (Electronic Control Unit) level. A reason for this is the lack of tool support for collaboration, configuration management and integration of data on the system level. Data is instead managed in multitudes of computer files, according to state-of-the-art software development practice. Another reason for the late integration is that the organizations responsible for the development of systems (and subsystems) are themselves organized in a way similar to the structure of the fragmented data. If you organize your system according to the technology required for the different parts, you probably have a development organization made up in the same way. This can even mean that there is actually no one in the organization responsible for the system level aspects. If the development of the subsystems is done by separate corporations, which is common in the automotive world, this situation gets even more accentuated.

3 Fragmented Data Management

According to our experience, file based data management is by far the most common way of handling data in the industry. This also applies to industries such as automotive that are more mature in data management. One reason for the spread of the file based approach is that many development tools still store their data in files, and the scale of individual development activities is small enough to be carried out in the traditional approach. However, the growing scale of systems like automotive electric/electronic systems combined with tight schedules in large development projects, and the need for integration and collaboration between different development activities has made the traditional approaches less feasible.

In traditional file based development individual artifacts are defined in separate files. File based versioning can keep track of changes and versions of such files, while also offering basic configuration support, usually using some label mechanism, where different files in a configuration are tagged with a label representing the configuration. A limitation with this approach is that the actual system configuration has to be built outside the versioning system, using some build script/mechanism, like the traditional 'make' command for software. This approach works well for development of software where an individual developer can develop a module of the software contained in a file, for a period of hours or days. For a complex system, or system-of-

[6] http://www.autosar.org/

system context, this level of granularity and level of collaboration becomes a bottleneck.

Many companies use tools for managing specific types of data such as requirements, tests, designs, or change requests. These tools are another source of fragmentation as they operate under the paradigm of information silos. Information silos have no or limited connection to their surrounding and are not aware of their organizational context. We call this approach silo-based information management tools. The data in these tools is stored on a finer granularity level than a file, which enables configuration and version management on a more detailed level than a file. Organizations use these tools increasingly to manage their data.

4 Integrated Data Management

As systems and systems of systems become larger and more complex, organizations realize the significance and difficulty of keeping data consistent. Data is produced and developed in teams that are spread geographically, and work across different disciplines and domains. These teams still need to have a shared understanding of the product and the organization. At the same time the data changes increasingly rapidly. In this reality, duplicating data creates inconsistencies unless advanced mechanisms are put in place to keep all copies of the data in sync. Also, data does not exist in isolation. Much of the value of the data comes from its context and how it is connected to other data. Hence, linking data sources instead of copying them is a natural solution, and there are clear trends, for instance the OSLC initiative, suggesting that industrial tool vendors are moving in this direction. IBM's Jazz platform and PTC Integrity are examples of this trend.

A few of the advantages of contextual and longitudinal traceability of data is to:

- trace the data back to the rationale and why it was created. This is important to assess the validity of data in a new context,
- see how a component, product, or process has changed over time. Organizing historic data is not only important in order to improve the future work, but it is in many cases valid since that data describes products that are still operating and being used in the market,
- see how a component is connected in a specific context,
- analyze the different uses of a component in different products,
- analyze the impact of the change to a component or requirements or any data in the whole organization

On a higher abstraction level one common solution to hold the data together and linked is to use a Product Lifecycle Management (PLM) approach where data is managed in a coherent framework. In this approach the actual system configuration, for example as defined by requirements, test cases and test results, is explicit in the PLM system, with no need for a separate build process like the one used in software configuration management. This means that data produced by test activities can update the configuration in real time, by direct access to the representation of the developed

system in the PLM system. This kind of real-time access and collaboration is not limited to human intellectual interaction, but can also be used for automated processes like regression tests. PLM systems typically do not manage development data such as software development data and test automation for complex embedded systems today.

As the complexity of systems increases, the industry has no choice but to improve their handling of low level data and also improve the handling of high level Systems Engineering data in their PLM systems. In this reality, challenges arise to connect these two worlds into a holistic data management ecosystem that works on an industrial level; solutions that not only can describe an organization, its information, and how the information is connected in one project or one product, but also over time.

4.1 Tool Interoperability & Data exchange

Intra-organizational communication across different groups, domains, and disciplines is a major challenge for many organizations. While fragmented data management strategies have tried to address this challenge, they often give rise to new types of complexity in form of consistency problems and major rework instead of reuse, by dividing the data into silos; no matter if the silo is a file or a database.

Inter-organizational communication such as supplier management and customer management is another important aspect of most development and manufacturing processes. The medium for this communication is often files. In this reality, data management tools pack the relevant data into collections of files such as requirements or test specifications. These files are sent to the supplier and are either processed in their existing form or unpacked into receiving tools on the supplier side. Depending on the tools involved and the nature of the data, these supplier interaction files have different levels of formalism. Packing and unpacking data creates challenges to keep the data correct and consistent during round-trips.

An alternative to packing data to files and unpacking on the receiver's side is that customers and suppliers use the same tool or use tools that use the same formalism for their data (for example EAST-ADL[7] or AUTOSAR) or even tools with formalisms that are transformable to each other. In this case the data can theoretically be exchanged with finer granularity than file. However, even this approach results in inconsistency in the data, as the root problem is duplication of data on a storage level and not the format or type of data. Duplication of data cuts the relationship between the data and the context that gives rise to it and the context in which it was created. The disconnection limits the ability to keep the data consistent and coherent and it makes the data unanalyzable for any foreseeable future until big data analysis methods become advanced enough.

One of the characteristics of complex systems, like embedded systems, is the multitude of aspects that need concern during development. The classical (minimal) Product Data management (PDM) approach is to manage the main product structure of the system, where detailed data is kept as proprietary, black box representations for each block in the structure. This solution is established and functioning for CAD data

[7] http://www.east-adl.info/

of mechanical systems. While, for complex multi-disciplinary systems the product or module structure is one of the least important structure to be managed. Other more or equally important structures or viewpoints are, to mention a few:

- Connectivity – interfaces between components defining responsibilities between them
- Interaction – how the component interact through the interfaces in order to achieve desired functional properties.
- Allocation – how functionality or responsibility is allocated to the physical structure
- Dependability – how the system components and requirements fulfill the safety goals of the system
- Requirements – how the properties of the system components and derived requirements fulfill product requirements
- Test and verification – the implementation status of the system as proven by test

To enable interoperability of tools involved in the development process all the above structures must be open. Open means that they must be visible and accessible. Moreover their semantics must be defined. One efficient approach to achieve this is to share a common meta model, accepted by the product domain. The level of required interoperability, from the needed viewpoints (listed above), defines the level of granularity of this openness, and a coherent domain specific definition of this openness for embedded (automotive) systems has been defined in the EAST-ADL meta model. The common formalism is also the cornerstone of the OSLC architecture.

4.2 Challenges in connecting tools and silos

As already discussed, linking distributed data in a way that is useful in an industrial context is a major challenge. Service-oriented architectures such as OSLC provide a possible first step of integration. In this step, loose links between two silos are created. The source tool (Consumer) can retrieve specific data from an OSLC service connected to a target tool (Provider) by providing the address to the desired data. This link can then be stored on the consumer side to establish a standing relationship with the provider.

Shallow links and deeper contexts. These links can only be used for navigation. For instance, while it is useful to see what requirement a test comes from, it is not possible to see the connection between a requirement and design or implementation items while standing in a test tool. In other words, a limitation with these links between silos is that an item can only be aware of the links it directly owns and therefore the depth of the visible links from the perspective of each item and tool is one (connecting only two silos at a time). These links per se can therefore not be used to generate reports or analysis on deeper contexts or structures.

Consistency and up-to-date data. In a distributed linking architecture, links are directed. Information about the target item and about the link are stored in the consumer tool. In this situation if the target item is deleted the link will be invalid.

Furthermore, there are no standard versioning concepts across tools. Many tools lack any versioning and configuration concepts at all. The rest of the tools handle these concepts in a heterogeneous way. For example, if there is a new version available of the target item, the consumer should be notified of the change.

Following the same train of thought it is clear that while the concept of weak links to connect fragmented data is useful from a user's perspective and enables the user to navigate quicker between two tools, there are many challenges to keep the links consistent between two silos. If this aspect is not considered creating and maintaining OSLC-like links can be a costly and inefficient concept in an industrial setting. Links to growing and live data need frequent updating either by use of bidirectional links with a notification mechanism or by a centralized data hub that keeps track of all links between silos.

Semantics and specifications. As mentioned earlier, predefined semantics and meta models is one solution to agree upon exchange and linking formats. However, in practice we see the difficulty to agree upon these semantics. Different organizations have different products and different processes, which leads to the need for different semantics even inside the same domain. AUTOSAR is a good example of this phenomenon. Although AUTOSAR is a widely spread standard, different organizations have different versions and interpretations of AUTOSAR. Organizations want the flexibility to decide their own processes and formalisms. Meanwhile, too much flexibility makes the tool integration difficult if not impossible.

SystemWeaver has a programmable meta model, which enables us to extend and integrate specific meta models in a single platform, rather than integrating different tools. Historically, we have built the interoperability in SystemWeaver, for tool and process integration, by defining a meta model to be used within SystemWeaver.

By the use of industrially accepted meta models e.g. EAST-ADL and means of technical integration as offered by OSCL this level of integration can be achieved also between different tools. Note that provisions for technical integration are not enough since there must be a shared definition of the semantics of the shared data.

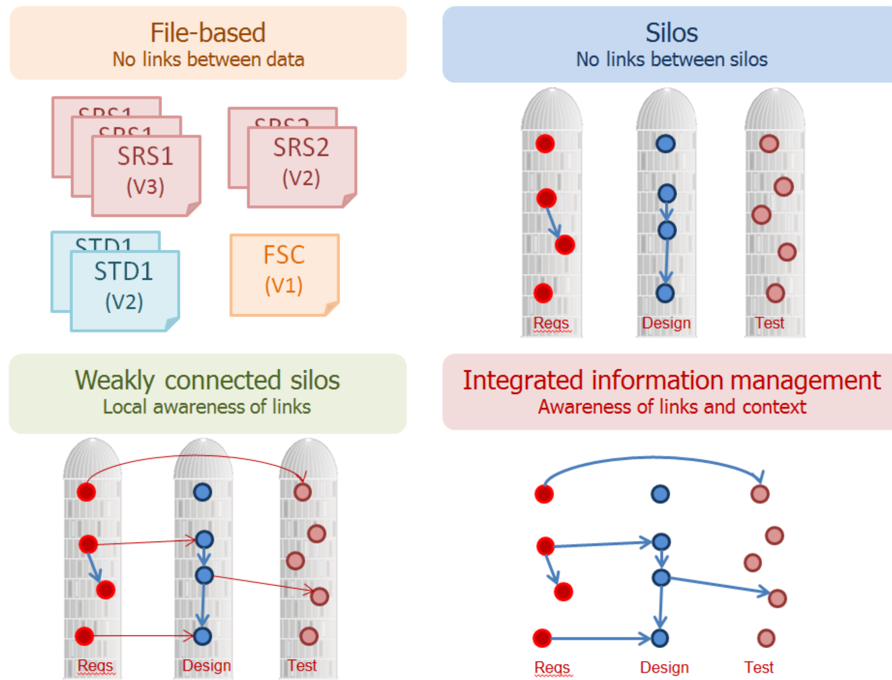
5 Discussion and Conclusion

A common way to solve a big problem is through the reductionist approach of breaking the problem down into smaller problems and dealing with them in isolation. Engineers define responsibilities and interfaces of a system early in the project on a high level in systems engineering. Early definition of interfaces requires simple and manageable interactions between subsystems. This leads to a development process with less need for integration of development data. Also, often we see that a problem is never really recognized as a problem until there is first a solution available. We think there is increasing awareness of the problems associated with fragmentation as dis-

cussed in the paper. A main driver for this awareness is the need for more cost efficient systems with higher performance, systems that require a higher degree of integration, and optimization of system properties.

In this paper we discussed four main approaches to data management. These four approaches are presented in **Fig. 1**. The file based and information silos that lead to fragmented data are the most commonly used data management paradigms.

Fig. 1. Four approaches to data management



As discussed, there are initiatives and a sense of urgency in the tool provider community to go beyond fragmented data. This need and urgency is a strong pull from the customers since the cost and complexity of fragmenting data is becoming clearer as systems become more complex. One approach to remove the fragmentation is to store and link all the data in one tool and platform and another is to create an ecosystem where different tools can coexist and cooperate, similar to the development in the smart phone app industry. Although the latter alternative seems more feasible and desirable, an open and distributed approach to tool integration is a large endeavor that requires many years of research and development. Two research questions that need to be answered are:

- How to create data links across tools to be able to traverse them in order to generate specifications and reports, export structured data to other tools, perform differ-

ent types of analysis, and etc. The traversal requires awareness of data links external to a single tool.

- Harmonized formalism between tools in order for the links to have semantics and in order to be able to define structures for the generated specifications, reports and analysis.

Tool interoperability has all the problems and challenges of classical data round-tripping (state of the information between export and import). These challenges manifest clearly in a weakly connected silo solution unless elaborate mechanisms are deployed. We can see two main solutions to address these challenges in order to create an industrial tooling ecosystem:

1. Bidirectional links with notification mechanism to keep the links between two silos consistent. The reason for bidirectionality is for the target item to be aware of the link in order to notify the source tool about potential changes.
2. Centralized data hub where there is a central/federal source that is aware of inter-tool links in an organization. In this case all tools need to notify the central hub about the relevant data changes and the central hub is in charge of keeping a consistent and up-to-date view of the links.

As discussed in the paper, there are many challenges such as versioning, configuration management, and data consistency that need to be addressed for both of these solutions to work. For example, when it comes to versioning, a decision is whether to point to the new version of the item or the old one. There are different types of links from a versioning perspective. A link that always refers to the latest version of an item, called floating link in SystemWeaver, or a link that points to a specific version of an item, called a stable Configuration Management (CM) controlled link.
When these mechanism are in place and the organizations can be confident about the validity of the links the next step would be to use the created link infrastructure to do analysis and generate reports, visualizations, and synthesize data. In case of solution 1, there is a need for peer-to-peer mechanism for generating reports that have data traces deeper than one link. According to our experience at Systemite, the ability to synthesis data that traverse deep structures is an important enabler for having fully generated reports and analysis and being confident about the correctness, completeness, and consistency of the generated reports and analysis.

References

1. Robinson, Mark A. "An empirical analysis of engineers' information behaviors. "Journal of the American Society for Information Science and Technology 61.4 (2010): 640-658.
2. Culley, S. J. "Court, AW, and McMahon, CA, 1992," The information requirements of engineering designers." Engineering Designer 18.3: 21-23.
3. Puttré, M. (1991, October). Product data management. Mechanical Engineering, 113(10), 81–83.