Reference Representation Techniques for Large Models

Markus Scheidgen
Humboldt Universität zu Berlin
Unter den Linden 6
Berlin, Germany
scheidge@informatik.hu-berlin.de

ABSTRACT

If models consist of more and more objects, time and space required to process these models becomes an issue. To solve this we can employ different existing frameworks that use different model representations (e.g. trees in XMI or relational data with CDO). Based on the observation that these frameworks reach different performance measures for different operations and different model characteristics, we rise the question if and how different model representations can be combined to mitigate performance issues of individual representations.

In this paper, we analyze different techniques to represent references, which are one important aspect to process large models efficiently. We present the persistence framework EMF-Fragments, which combines the representation of references as source-object contained sets of target-objects (e.g. in XMI) within the representation as relations similar to those in relational databases (e.g. with CDO). We also present a performance evaluation for both representations and discuss the use of both representations in three applications: models for source-code repositories, scientific data, and geo-spatial data.

Keywords

EMF, Meta-Modeling, Model Persistence, Big Data, Mining Software Repositories

1. INTRODUCTION

In *regular* MDE, where models are small, the performance (time and space consumption) of most operations is neglectable. In *BigMDE*, where models can grow arbitrarily, performance becomes more and more important. To build *BigMDE* applications that scale with the size of large meta-model based models (or meta-model based data-sets in general), we have to understand how models are represented and processed by the frameworks we use to implement these applications.

BigMDE '13, June 17, 2013 Budapest, Hungary

Meta-modeling (i.e. the use of meta-models and their instances) is always abstract and never real. To actually work with models, we always have to use other *technological spaces*[1]. To work with a model in a computer program, we map meta-models to Java classes and interfaces (programming); to exchange models we map meta-models to XML-schemas (XML); to persist large models, we map meta-models onto relations in an SQL-database (relational databases). While all technological spaces basically allow us the same operations, they do not guarantee the same performance for the same operation. To realize efficient processing of large models it is important to understand how models are represented with the respective technological spaces and how these different representations effect the performance of certain operations. In this paper, we focus on three technological spaces that are relevant for persisting, accessing, and manipulating large models. These are: (1) *databases* that we use to persist large amounts of structured data, (2) *programming* that we use to process models within computer programs, and (3) *XML* for serializing models or parts of models.

We can distinguish different modeling aspects that can be represented in different ways. There are at least two aspects with a heavy influence on the performance of processing large models: *object organization* and *reference representation*.

Organization of objects: Meta-model based models comprise objects connected through references[2]. Objects can be organized either as larger model *fragments* (e.g. like XMI files or EMF resources) or they are organized as *individual objects* that can be accessed and stored independently of other objects (e.g. in relational databases via CDO).

Reference representation: References can either be repre-

[1]The term *technological space* refers to a set of methods, languages, and tools based on a single theoretic background that distinguishes it from other technological spaces. The initially most discussed technological spaces are probably *model-ware* (i.e. meta-modeling) and *grammar-ware* [33]. The former based on object-oriented data models and their instance semantics, the later based on grammars as in formal language theory. In this paper, we use the following terms to refer to respective technological spaces: *meta-modeling*, *XML*, *databases* (with specializations *relational databases* and *key-value stores*), *programming*, and *grammar-ware*.

[2]Please do not confuse this with the EMF Ecore meta-modeling concept EReference which is used to model references (i.e. one end of an association). Thus an EReference defines a set of possible references, like each EClass defines a set of possible objects. In some publications references are referred to as links. But this term lacks the inherent direction of references.

sented as sets of target objects that are maintained as *part of the source object* (e.g. in XMI or in EMF's in memory representation) or as independent *relations* (e.g. in relational databases via CDO). As we will see later, both *fragments/individual objects* and *part of source/relations* can have contradictory performance properties.

In [25], we analyzed the different realizations of the first aspect *object organization*. We compared *individual object* organization in relational databases (CDO [1]) and document databases (Morsa [19]) with the organization as fragments (our framework EMF-Fragments[3]) We could verify our assumption that *fragments* perform well for traversing models (i.e. the whole model or at least large parts of the containment hierarchy have to be loaded eventually), and that *individual objects* work better for executing queries (i.e. accessing small specific parts of the model).

In this paper, we want to focus on the second aspect *reference representation*. In [25], we observed that *part of source* does not scale with the number of referenced objects, because all references are stored (and have to be completely loaded) as part of the source object. Others [19, 4] have observed that *relations* on the other hand require complex and expensive database operations (e.g. joins) when models are accessed.

Our hypotheses is that we can combine both representations to allow clients to choose different representations for different associations. Ideally, we only have to bear the performance burden of *relations*, if the scalability issues of *part of source* references demand it. We present our framework EMF-Fragments that has been extended to support *part of source* references and *relations* similar to those in relational databases. We show some measurement results and discuss the possible influence based on three example applications.

We only address the possibility to make static choices based on the expected nature of associations (meta-model level). We do not yet consider the possibility to use different representations for the same association based on the actual number of references that are required with-in concrete models (model level). Also we only address performance and scalability issues in this paper. All the presented representations can also imply other important properties (e.g. the ability for safe transactions) that have to be considered for real applications.

The paper is organized as follows: In the next section 2, we introduce the two mentioned reference representations in more detail. The following section 3 will describe our framework. Section 4 provides the performance measures. This is followed by related work in 5 and the discussion of applications in 6. We conclude the paper with summary and suggestions for future work in 7.

2. TWO REPRESENTATIONS OF REFERENCES

In this section, we explain the two introduced reference representations in more detail. First, we introduce the notion of a *values-set*. As a common abstraction, we denote the set of all target objects that are referenced from a single source object with respect to the same association end (e.g. EResource in EMF) a *value-set*. We discuss the performance of both representations with a focus on scalability (small vs. large value-sets), navigating individual references (accessing a value-set), and scanning through large sets of references (iterate value-sets). We consider both model query and traversal operations, where a query is used to access individual model elements, and traversal means to access all model elements eventually. Even though many factors influence the performance of these operations, we roughly assume that scalability is essential for both query and traversal operations, while fast navigation of individual references are important to queries and iterating value-sets is required for fast traversal.

2.1 Part of source

Both EMF's in memory Java object based representation of models and XMI's tree-based representation are based on *part of source* references. Java objects that represent model objects refer to associated objects via different list implementations that represent corresponding value-sets. Accessing the next object while iterating a value-sets has constant complexity. Accessing individual objects in a value-set in this representation, requires to iterate through the whole value-set and to evaluate a predicate on each entry. Accessing individual objects therefore has linear complexity depending on value-set size. Accessing the first object or an object at a specific position (if value-sets are realized as lists) has constant complexity. In XMI, objects in value-sets are either represented as child-nodes of the source object (containment references) or as a list of URIs that refer to the corresponding target object. To manipulate a value-set the whole value-set has to be loaded (and stored afterwards). Loading and saving has at least linear complexity in value-set size.

Access, load, and save perform linear and can present serious performance issues if value-sets are large. Since objects can only be loaded and represented in computer memory as a whole (i.e. including value-sets), large value-sets also present a threat to the limited heap memory. In summary *part of source* fails for very large value-sets, but has good performance for small value-sets, especially when iterating value-sets (model traversal).

2.2 Relations

In relational databases, entries of different tables (e.g. representation of model objects) can be related to each other, via additional tables where each entry represents a reference between a source object and a target object. Source and target objects are usually identified via unique IDs. In theory this table oriented style of data organization is based on relational algebra. Therefore, we refer to tables (slightly incorrectly[4]) also as relations. Object relational mappings map classes to relations and associations to more relations. Relations can be accessed via target ID or manual implemented predicates: usually frameworks like CDO allow this via custom SQL-queries, which requires an in-depth understanding of the ORM and requires to circumvent its transparency. Implementing predicates purely with the EMF-API usually ends in iterating the value-sets. Relations can also be resembled with similar properties by non relational databases

[3]EMF-Fragments transparently fragments models along their containment-tree based on client annotations in the meta-model.

[4]These tables can conceptually also be seen as persisted hash-maps. But keep in mind that most databases organize these tables in some form of balanced tree that usually only provides logarithmic (and not constant) access times.

(see 5 and 3).

To navigate individual references or even all references complex join operations have to be performed. The actual complexity depends on concrete database design and ORM-mapping (e.g. use of indexes). But we can assume it is above constant and ideally less than linear. Therefore, simple queries in large *relations* can be fast compared to queries in *part of source* representations. But this might require to leave EMF's API and use SQL on the mapped model data. Iterating value-sets is comparatively slow. There are no scalability issues for large value-sets, even though we have to assume at least logarithmic complexity for accessing individual references.

3. EMF-FRAGMENTS

In this section, we present EMF-Fragments; a framework that implements both introduced reference representations. We originally developed EMF-Fragments as a framework that realizes *fragments* for large models in contrast to the other existing persisting frameworks that to our knowledge exclusively rely on *individual objects*. Since *fragments* allow us to organize the same model in different ways, we choose to allow clients to describe the desired fragmentation on a meta-level to control performance (and other) characteristics on the model-level. EMF-Fragments uses regular EMF resources to (de-)serialize fragments as XMI (or in EMF's less verbose binary format). Therefore, the original version of EMF-Fragments did only provide *part of source* reference representation with all the described limitations.

We first describe the general architecture, the data stores EMF-Fragments can use, and how it realizes *fragments* before we finally explain *relations* and how it can be combined with *part of source*. We close this section discussing the limits of EMF-Fragments.

3.1 Architecture

Figure 2 shows the general architecture of our framework. From the client's perspective models are maintained via the regular EMF-API that comprises `ResourceSet`, `Resource`, and `EOBject`. Here, the whole model is logically contained in a single resource. Clients only access the model via proxy objects that delegate all functionality through an `EStore`. The original `EStore`-API intent is to separate EMF's generated API from the actual model representation (e.g. in a database). Instead of hiding a completely different model representation behind the `EStore`, we hide an internal also EMF-based representation of the model consisting of `ResourceSet`,`Resources` and `EObjects`. But here, one model is organized as a set of several resources that we call *fragments*. This design allows us to transparently realize fragmentation without any efforts on client side.

3.2 Data stores

EMF-Fragments relies on a key-value store. The term key-value store denotes an abstraction for data stores that allow logarithmic access to a large table of byte-array keys and values. Additionally key-value stores might be sorted, which yields constant scan complexity (accessing the next entry). Typical implementations are file-systems (file URI and content, not sorted), big-table databases (e.g. Apaches HBase, sorted), or document oriented databases (e.g. MongoDB, can be sorted via indexes [21]). EMF-Fragments uses these stores in different ways. Foremost to store fragments:

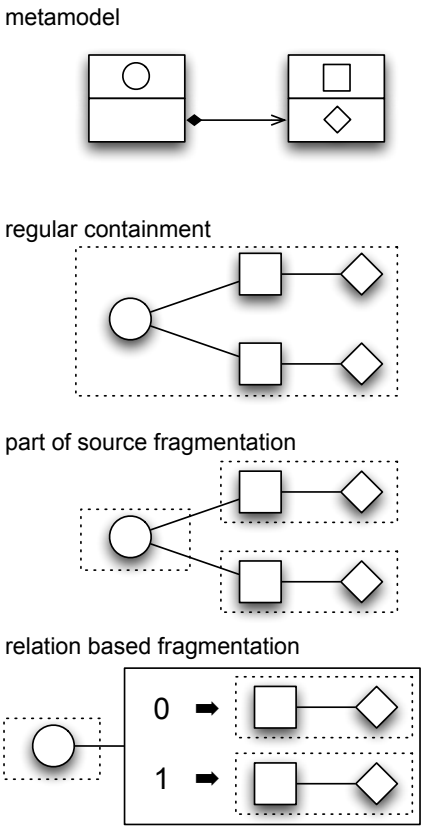

Figure 1: Schematic visualization of regular containment, fragments, and relations.

where URI's are used as keys and XMI or binary serialization as values (see 3.3). But composing keys from several parts also allows us to put several key-value tables into one store. We use this to realize relations (see 3.4) and to maintain dynamic references (see 3.5).

3.3 Fragments

Fragmentation is controlled by clients through annotations in the meta-model. Clients can mark containment references as *fragmenting*. This information is used by EMF-Fragments to automatically fragment a model along its containment hierarchy. Our `EStore` implementation intercepts client initiated changes to value-sets and thus can create and remove fragments when the value-set of an fragmenting reference is changed. Each value in such a value-set becomes the root object (with respect to the containment hierarchy) of its own fragment. Each fragment is issued an unique ID that identifies it within a data store. The fragments URI is a composition of the model URI and the fragment ID. Objects within a fragment can be identified via EMF's URI fragments (also called intrinsic IDs [29]). The middle of Figure 1 depicts fragments as boxes with dotted lines.

3.4 Relations

EMF-Fragments as described before only provides *part of source* references. To deal with large value-sets, clients can mark references as *relations*. EMF-Fragments then replaces EMF's build-in value-set implementations (different descendants of `EList`) with an implementation that refers to a

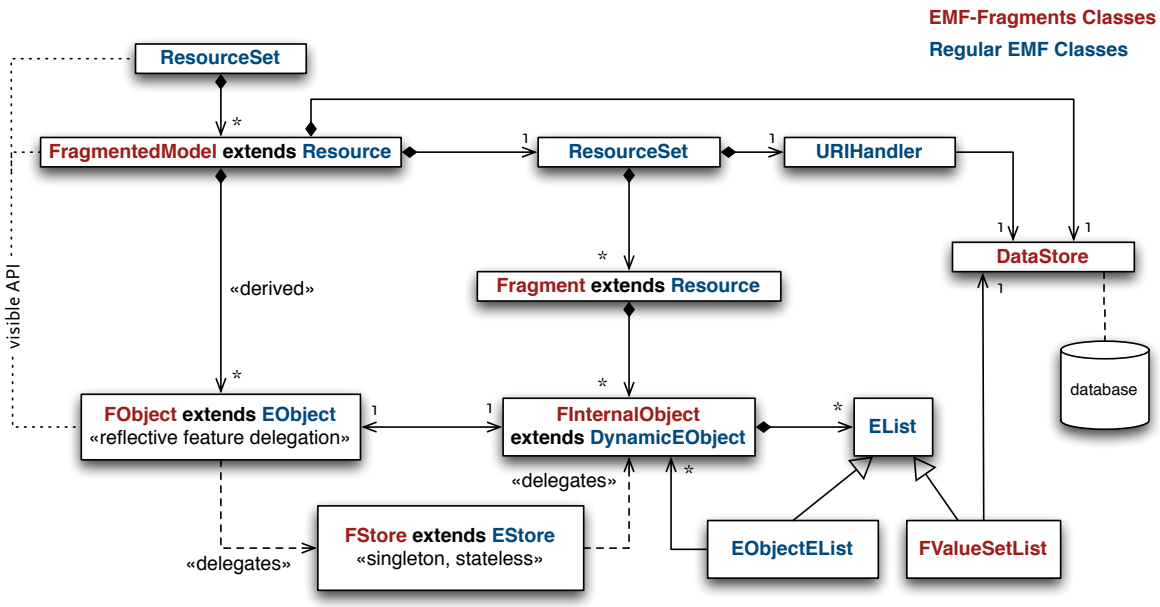

Figure 2: Basic EMF-Fragments architecture.

relation in the data store instead of holding values itself. Each relation is a sub-table of the model's data store table (see 3.2). Relations are identified by the owning object's URI and the corresponding meta-model feature. Internally all EMF value-sets are lists (i.e. *EList*). We use an objects position in these lists as keys for objects in relations. Values are either URIs to the targeted object (for non containment references) or the object itself (for containment references). The bottom of Figure 1 shows a *relation* for containment references.

3.5 Issues and Limitations

There are three conceptual issues and performance limitations that we want to discuss. First, EMF-Fragments' relations are not really relations as in relational databases. In relational databases source and target objects are related more directly. We, on the other hand, only relate a single source object with a indexed set of target-objects. But the point is that our realization of relations probably provides similar performance properties, especially when compared to *part of source* references (compare with section 4).

Secondly, non-containment references are stored as URI's that refer to the referenced object. Since URI's that use intrinsic IDs depend on an objects position in the containment-hierarchy, the URI for each incoming reference has to be changed when an object is moved. To avoid this, we use unique IDs to identify referenced objects. These IDs are maintained using the model's data store table. This bloats the table considerably and slows access times.

Thirdly, one has to be careful when using operations on value-sets within EMF. Depending on the properties defined in the corresponding `EReference`, EMF runs several implicit operations on value-sets. For example, if we add a value to a unique value-set (EMF's default on `EReferences`), EMF first checks if the value is already contained in that set. This means EMF iterates through the whole value-set before adding the value. This can be devastating for very large value-sets in general, and with *relation* backed value-sets in particular (compare with 4).

4. EVALUATION

In this section, we want to compare the introduced reference representations based on performance. Since, we do this to provide a clear basis for decision making, all measurements where performed on synthetic models with clear properties and not on actual case study data. A comparison of our framework with CDO and Morsa (see also section 5) based on actual models can be found here [25]. Since, we implemented all representations within the same framework, we get comparable results that are not distorted due to unrelated properties of individual frameworks (e.g. CDO executes everything within transactions, EMF-fragments does not). All measurements were performed with EMF-Fragments, MongoDB 2.2.3, Eclipse 3.7, on a 8-core Intel(R) Xeon(R) 3.30 GHz computer with 8 GB of RAM running 64-bit Ubuntu Linux 12.04.1 LTS. All measures were repeated 23 times: the displayed results are averages. All measurements were done with a synthetic model consisting of only two objects that play the role of source and target object. Here, the only target object is referred to multiple times to realize arbitrary large value-sets (corresponding to a single non-unique Ecore reference[5]).

We conducted two measurements for three different implementations of references for value-sets of increasing size (1 to 10^6). The results are shown in Figure 3. First, we measured the execution time of accessing a single value from a value-set (squares). Secondly, we measured the execution time of iterating through the whole value-set (triangles). The first implementation is the default EMF *part of source* realization for references. The execution includes loading the model from an EMF binary resource into memory and the actual value-set access/iteration. The second

[5]The value-set becomes a multi-set that can contain the same value several times.

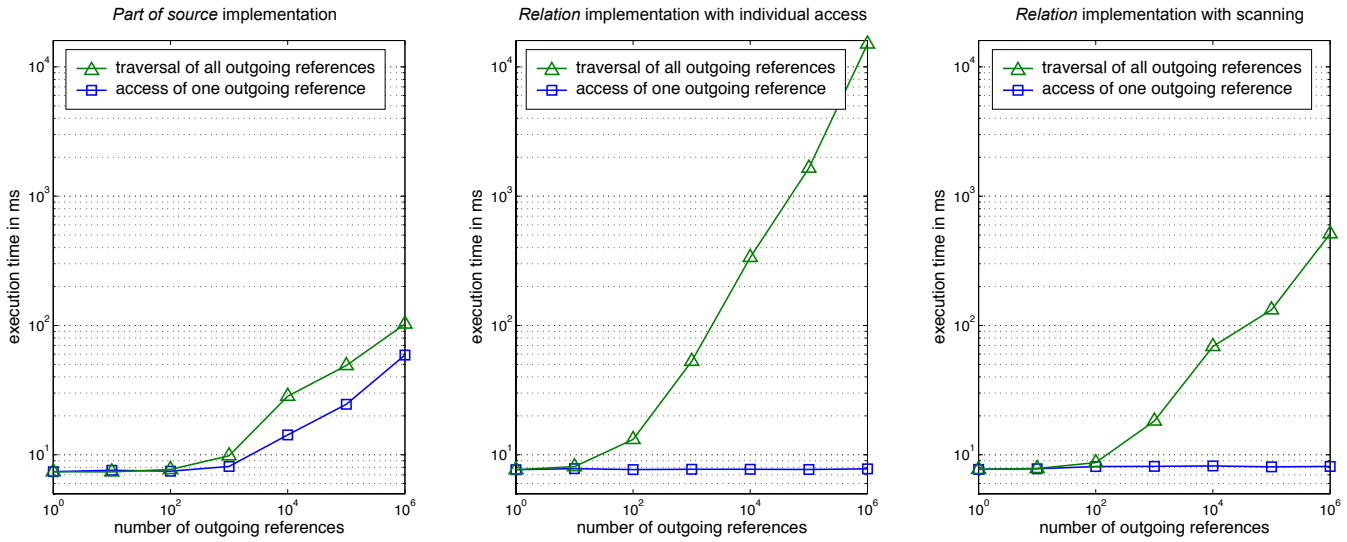

Figure 3: Execution times of traversal and access of value-sets with different sizes using *part of source*, *relations*, and *relations with scans* representation.

implementation, is our *relation* based representation of references (see 3.4), where we access the corresponding database table (MongoDB collection with index) once for each value. The third implementation also realizes *relations* , but here we scan the corresponding database table during iteration of the value-set.

We can observe three things from the results. First, with *part of source* even accessing a single value requires more time when the value-set grows. Secondly, *relations* are slow to iterate when values are accessed individually. Thirdly, the execution time of scanning a *relation* based value-set has the same magnitude as iterating a *part of source* relation, but it is still considerably (a factor of 5) slower.

We can conclude two things. First, one has to choose very carefully, because the possible performance gains and losses are large when value-sets are large. Secondly, for *part of source* value-sets with about 100 to 1000 objects the difference between accessing a single element or iterating the whole value-set is not that big. In other words, significant performance gains can only be expected for value-sets with more than these 100 to 1000 objects. We originally expected that for much smaller value-sets.

5. RELATED WORK

There are several frameworks for persisting large models based on different data stores. The most mature frameworks are based on relational databases and object relational mappings (ORM). These frameworks realize *individual object* organization and represent references with *relations*. Both CDO [1] and Teneo [2] are established parts of the EMF ecosystem. ORM based persistence has the described performance issues, but provides safe transactions (ACID properties) and arbitrary large value-sets.

With NoSql becoming more and more popular in the last years, the first persistence techniques for non relational databases are researched. Morsa [19, 20] uses the document oriented database MongoDB [21] to persist *individual objects* as documents. Objects are serialized as JSON strings. References are therefore represented as *part of source*. *Individual objects* storage within Morsa imposes performance issues similar to ORM based systems, in addition to the scalability issues implied by *part of source* reference representation. With multiple caching mechanisms however, Morsa provides better performance than ORM systems for some applications, especially when value-sets are small.

In [4] persistence of models in graph-databases is evaluated. Graph databases are based on the paradigm of *index-free adjacency*. Therefore do not require expensive join operations when models are navigated. Otherwise, models are mapped onto graph-databases with *individual objects* organization and a native (*index-free*) representation of references. Unfortunately no strong conclusions can be drawn from the measurement results in [4], because only measurements for models with arbitrary characteristics (the Grabats 2009 contest [3]) are provided.[6]

Within the *big data* community, lots of research (e.g. [30, 6, 11, 5]) is done to transfer relational database/SQL concepts to the popular peer-to-peer based big data processing systems (e.g. hadoop [31]). The use of SQL like languages and relational data representation could allow us to also facilitate ORM mappings on these kinds of database systems (e.g. use Hive [30]) too. But there is no work to our knowledge yet.

Most key-value stores or similar databases (multi row/-column, document-oriented, big-table, etc.) provide slightly more structured organization of data than a plain key-value

[6]The Grabats 2009 contest [3] models and tasks (used to evaluate Morsa, graph databases, and EMF-Fragments in [19, 4, 25]) as a benchmark only allows us to see which framework is faster, but we cannot see why. All the used models have different (quasi random) internal structure. Even though they have different sizes they do not provide linear ramp-ups for characteristics like model-size, connectivity, containment-hierarchy depth, etc. Therefore, we cannot deduct whether a persistence framework runs certain operations in logarithmic, linear, or exponential time/space depending on what model characteristics.

table. Databases like HBase, Cassandra, Amazons Dynamo, etc [16, 17, 9] allow us to distribute values into multiple cells and provide secondary indexes (indexes on cell values) and respective access functionality. These systems could be used to provide more efficient representation of references as *relations*.

Unrelated to persistence, but to human comprehension of complex models, techniques have been developed to extract sub-models from large models that cover individual semantic aspects of that large model [7, 15].

6. APPLICATIONS

In this section, we briefly introduce three applications for large meta-model based data-sets and discuss model representations with respect to each application. One is a typical software engineering application, the other two applications examplify the use of MDE methods for other domains. In the introduction, we used the notion of connecting different technological spaces to provide methods from other spaces to software modeling. E.g. we use the data persistence capabilities of relational databases to store large software models. On the other side, we can also connect technical spaces to bring the advantages of software modeling to other domains. E.g. provide textual and graphical notations or expressive transformation languages (software modeling) to structured data processing (databases) in general and to the following use-cases in particular.

6.1 Model-based Mining Source Repositories

The first application that we look at, applies software-modeling methods (meta-modeling, model-transformations) to the mining of source repositories (MSR) [14]. Here we construct a large model that comprises a detailed AST-level model of all revisions of a software code repository. Based on such a model, one can employ model transformations and model query techniques to achieve MSR goals. Typical MSR goals are (1) identification and prediction of bugs [32], (2) finding API usage patterns [18, 8], or (3) visualization of implicit code dependencies [34][7].

We build a small framework [23] based on JGit (a Java API for using Git repositories), MoDisco (an EMF-based framework for reverse engineering Java code), and our own EMF-Fragments to create repository models. In the future we want to use EMF-compare (or comparable frameworks) to analyze differences between revisions. The top of Figure 4 shows parts of the used meta-model. It combines concepts for Git repositories with Java AST classes. The bottom part of Figure 4 depicts an example repository with two branches (white and red) and respective revisions (boxes with round corners), a Java package (octagon), and several Java classes (A-D) in different revisions. In our current implementation, the model comprises all changed compilation units for each revision. References within the Java model (e.g. declaration-usage) reference objects within the the same revision (if the referenced object is part of a compilation unit that was changed in the current revision) or to object from an older revision (the revision where the containing compilation unit was updated last).

[7]Pieces of code that have been added or changed at the same time are likely to depend on each other. This allows to find dependencies that are invisible to static analysis methods. Also known as *market basket analysis* [14], e.g. *people who bought ... also bought ....*

The resulting models of source repositories become very large, compared to the size of the original Git repositories. The two reasons are: (1) the model contains whole compilation units and not just the changed parts and (2) ASTs are a far less space efficient representation of code. Models are about 400 times as big as the original code repositories (measured in bytes required to store both representations). For example, the Git repository for EMF itself is about 50 MB. The model of this repository requires about 20 GB (using MongoDB with serialized binary resources). This increase of *a couple of hundred times* fits the prediction in [14].

Among others the source-repository meta-model contains two (kinds of) associations that potentially produce very large value-sets. The first association is those between an repository and each revision. A repository contains many revisions that users might want to access individually (e.g. over 8.500 in EMF's Git repository). The second kind of association is those between declarations and entities that use declarations, e.g. packages and imports or methods and method call, etc. The later kind of association can produce value-sets of vastly different and hardly predictable sizes. A local method might only be accessed a couple of times, `java.lang.Object#toString()` is accessed a lot, and since packages are hold on a per repository and not on a per revision basis, a single package might be imported from compilation units in all revisions. If we combine multiple repositories that use the same APIs, value-sets might get even larger. The majority of associations however will only produce small value-sets. Child/parent relations between revisions or the containment of statements in blocks are examples.

In Figure 4, we use stereotype notation to mark those association ends that we want to cause *fragmentation* and that we want to be realized with *relations*. The default behavior is regular containment and *part of source* references. This meta-model shows that their might by associations within the same meta-model that should be realized differently. It also shows that it might be very difficult to decide. Especially, when we cannot predict how large all the value-sets for an associations will become, or we already know that one association will cause both small and large value-sets. This also shows, that switching representations at *runtime* might be worth investigating. In the future, we want to use this application as a vehicle to evaluate the influence of different model representations on the performance of model transformations that realize complex aggregations and queries within large models (e.g. realize certain MSR algorithms).

6.2 Sensor data

In [26], we introduced the framework Clickwatch that can obtain and manage scientific data from wireless sensor networks (WSN). Clickwatch represents WSNs, as hierarchical EMF models comprising networks, sensor nodes, and individual sensors. In these models sensors are associated with large sets of sensor readings. Depending on sample rate, a sensor collects several 100 readings per seconds. Considering that data is obtained within experiments that last for days, weeks, or even months, value-sets with more than 10^6 are very common [22]. We use WSNs for a series of applications reaching from earthquake early warning [10] to traffic surveillance [24].

We currently use this as a case study to expand our *relation* based representation of references from value-sets to

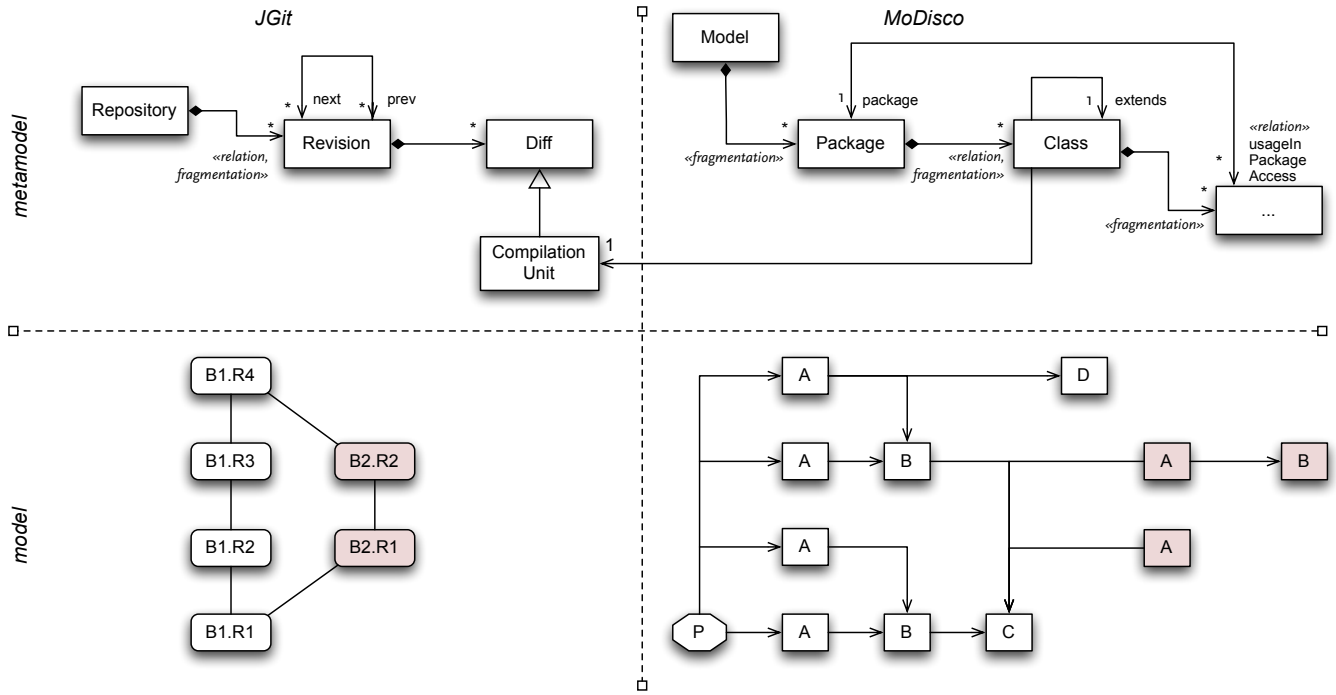

Figure 4: Excepts from metamodels and example models for source code repositories.

value-maps to associate source objects (e.g. sensors) with target objects (e.g. sensor readings) based on a key (e.g. timestamps) that can be used to access individual values or value ranges. In [27], we compared the performance influence of *individual object* and *fragment* organization for this specific use-case.

6.3 CityGML models

CityGML [12, 28] is an XML-based standard for 3D geo-spatial models of cities. These models can become particularly large, when they comprise the data of large cities [22]. These models present an interesting case-study for two reasons. First, they are an example for a large collection of XML-based standards and data. What we can show for CityGML should hold for a many other XML-based datasets. Secondly, CityGML models contain geo-spatial data. Respective applications can only be realized efficiently, if objects can be accessed efficiently based on their geo-spatial location. This requires the use of proprietary indexes (e.g. R-trees [13]). If we want to use meta-modeling (e.g. EMF) beyond software modeling, we need to show that we can cover such specialties within an abstraction like Ecore. Therefore, we want to use this to show that we can also incorporate proprietary indexes to realize associations.

7. CONCLUSIONS

In this paper, we examined two different representations of references (*part of source* and *relations*) and how they can influence the performance of applications that involve large models. From a measurement based evaluation, we can conclude that choosing the one or the other representation can have a large influence depending on the number of referenced target-objects. However, we must also conclude that we cannot always decide at a meta-model level (i.e. at development time), how large the these sets of referenced objects will be. Furthermore, there are cases, where a single association cause small and large sets of target-objects for different source-objects. We also saw, that the size of those value-sets where the representation starts to have a significant influence is about 100 to 1000 objects. Thus, the choice might not be that important for some *BigMDE* applications.

In the future, we want to explore the possibility to change the representation of a model at runtime. Thus switches between *individual object* and *fragments* organization and *part of source* and *relation* based references at runtime. This also has to include switching between fragment sizes at runtime and allowing double representation of references (i.e. maintain both *part of source* value-sets and *relations*) where appropriate. Here model dynamics become more important, decisions have to be made based on usage patterns, and costs for reorganizing model representations have to be considered.

In general, we think that the *BigMDE* community would gain from better benchmarks. We made the observation that many publications in this new field use models from the Grabats 2009 contest [3]. It is good to have this possibility to compare different approaches based on *real* models. But we also need a set of *synthetic* models and specific task that allow us to separate different concerns. We can only make specific predictions, if we can recreate the specific scenarios that might arise in a *BigMDE* application.

8. REFERENCES

[1] Connected Data Objects (CDO). http://www.eclipse.org/cdo/.
[2] Teneo, eclipse modeling framework technology (emft). http://www.eclipse.org/modeling/emft.
[3] Grabats 2009, 5th International Workshop on

Graph-based Tools: A Reverse Engineering Case Study. http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/, Jul 2009.

[4] K. Barmpis and D. S. Kolovos. Comparative analysis of data persistence technologies for large-scale models. In *Extreme Modeling workshop, XM 2012 at ACM/IEEE 15th International Conference on Model Driven Engineering Languages & Systems*. ACM Digital Library, October 2012.

[5] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, editors, *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 119–130. ACM, 2010.

[6] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C.-C. Kanne, F. Özcan, and E. J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12):1272–1283, 2011.

[7] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Modeling model slicers. In J. Whittle, T. Clark, and T. Kühne, editors, *MoDELS*, volume 6981 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2011.

[8] S. Breu, T. Zimmermann, and C. Lindig. Mining eclipse for cross-cutting concerns. In S. Diehl, H. Gall, and A. E. Hassan, editors, *MSR*, pages 94–97. ACM, 2006.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[10] J. Fischer, M. Günes, K. Nagel, T. H. Kolbe, M. Scheidgen, and et al. From earthquake detection to traffic surveillance: About information and communication infrastructures for smart cities. In Ø. Haugen, editor, *Proceedings of the 7th System Analysis and Modeling Workshop*, LNCS. Springer, 2012. to appear.

[11] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a highlevel dataflow system on top of mapreduce: The pig experience. *PVLDB*, 2(2):1414–1425, 2009.

[12] G. Gröger, T. H. Kolbe, A. Czerwinski, and C. Nagel. OpenGIS City Geography Markup Language (CityGML) Encoding Standard, Version 1.0.0. Technical Report Doc. No. 08-007r1, OGC, Wayland (MA), USA, 2008.

[13] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[14] H. H. Kagdi, M. L. Collard, and J. I. Maletic. Towards a taxonomy of approaches for mining of source code repositories. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[15] P. Kelsen, Q. Ma, and C. Glodt. Models within models: Taming model complexity using the sub-model lattice. In D. Giannakopoulou and F. Orejas, editors, *FASE*, volume 6603 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2011.

[16] A. Khetrapal and V. Ganesh. HBase and Hypertable for Large Scale Distributed Storage Systems A Performance evaluation for Open Source BigTable Implementations. Technical report, Purdue University, 2008.

[17] A. Lakshman and P. Malik. Cassandra: Structured Storage System on a P2P Network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 5–5, New York, NY, USA, 2009. ACM.

[18] V. B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In M. Wermelinger and H. Gall, editors, *ESEC/SIGSOFT FSE*, pages 296–305. ACM, 2005.

[19] J. E. Pagán, J. S. Cuadrado, and J. G. Molina. Morsa: a scalable approach for persisting and accessing large models. In *Proceedings of the 14th international conference on Model driven engineering languages and systems*, pages 77–92. Springer-Verlag, 2011.

[20] J. E. Pagán, J. S. Cuadrado, and J. G. Molina. A repository for scalable model management. In *Software and Systems Modeling*. Springer, March 2013.

[21] E. Plugge, T. Hawkins, and P. Membrey. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Apress, Berkely, CA, USA, 1st edition, 2010.

[22] M. Scheidgen. How big are models – an estimation. Technical report, Humboldt-Universität zu Berlin, 2 2012.

[23] M. Scheidgen. Source Repository. http://github.com/markus1978/srcrepo, 2013.

[24] M. Scheidgen and A. Zubow. Emf modeling in traffic surveillance experiments. In K. Duddy, J. Steel, and K. Raymond, editors, *Modeling of the Real World*. ACM Digital Library, October 2012. to appear.

[25] M. Scheidgen, A. Zubow, J. Fischer, and T. H. Kolbe. Automatic and transparent model fragmentation for persisting large models. In R. France, J. Kazmeier, C. Atkinson, and R. Breu, editors, *Models 2012*, volume to appear. Springer Verlag Berlin/Heidelberg, 2012.

[26] M. Scheidgen, A. Zubow, and R. Sombrutzki. Clickwatch - an experimentation framework for communication network test-beds. In *IEEE Wireless Communications and Networking Conference, WCNC 2012, Paris, France, April 1-4, 2012*, pages 3296–3301. IEEE, 2012.

[27] M. Scheidgen, A. Zubow, and R. Sombrutzki. HWL – a high performance wireless research network. In *The Ninth International Conference on Networked Sensing Systems (INSS 2012)*, Antwerp, Belgium, In Press 2012. IEEE, IEEE.

[28] A. Stadler. Making interoperability persistent: A 3D geo database based on CityGML. In J. Lee and S. Zlatanova, editors, *Proceedings of the 3rd International Workshop on 3D Geo-Information*, pages 175–192. Springer Verlag, Seoul, Korea, 2008.

[29] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, 2nd edition, 2009.

[30] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive-a warehousing solution over a map-reduce framework. In *IN VLDB '09: PROCEEDINGS OF THE VLDB ENDOWMENT*, pages 1626–1629, 2009.

[31] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[32] C. C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Trans. Software Eng.*, 31(6):466–480, 2005.

[33] M. Wimmer and G. Kramler. Bridging grammarware and modelware. In *Proceedings of the 2005 international conference on Satellite Events at the MoDELS*, MoDELS'05, pages 159–168, Berlin, Heidelberg, 2006. Springer-Verlag.

[34] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Trans. Software Eng.*, 31(6):429–445, 2005.