Model-based Mining of Source Code Repositories

Markus Scheidgen and Joachim Fischer

Humboldt-Universität zu Berlin,
Unter den Linden 6, 10099 Berlin, Germany
{scheidge,fischer}@informatik.hu-berlin.de
http://www.hu-berlin.de

Abstract. The *Mining Software Repositories* (MSR) field analyzes the rich data available in source code repositories (SCR) to uncover interesting and actionable information about software system evolution. Major obstacles in MSR are the heterogeneity of software projects and the amount of data that is processed. Model-driven software engineering (MDSE) can deal with heterogeneity by abstraction as its core strength, but only recent efforts in adopting NoSQL-databases for persisting and processing very large models made MDSE a feasible approach for MSR. This paper is a work in progress report on *srcrepo*: a model-based MSR system. *Srcrepo* uses the NoSQL-based EMF-model persistence layer *EMF-Fragments* and Eclipse's *MoDisco* reverse engineering framework to create EMF-models of whole SCRs that comprise all code of all revisions at an abstract syntax tree (AST) level. An OCL-like language is used as an accessible way to finally gather information such as software metrics from these SCR models.

1 Introduction

Software repositories hold a wealth of information and provide a unique view of the actual evolutionary path taken to realize a software system [1]. Software engineering researchers have devised a wide spectrum of approaches to extract this information; this research is commonly subsumed under the term *Mining Software Repositories* (MSR). A specific branch of MSR uses statistical analysis of code metrics gathered for each software revision to understand the evolution of software projects [2]. Recent advances in large-scale data processing (i.e. NoSQL-databases and Map/Reduce-style processing) allowed to extend this research to *large* or even *ultra-large* scale software repositories that comprise a large number of software projects [3]. Examples for large repositories are the projects hosted under the umbrella of the *Apache Software Foundation* or the *Eclipse Foundation*, and ultra-large repository examples are web-based software project hosting services like *GitHub* (250.000+ projects) or *SourceForge* (350.000+ projects) [3]. But analyzing many heterogeneous software projects has limits. While existing approaches [4,5,6,3] manage to abstract from different *code versioning systems* (e.g. CVS, SVN, Git), different programming languages with different syntax and

semantics are still a major issue. The EU FP 7 project FLOSS [4] for example produced data sets for over 3000 libre software projects, but could only gather language independent text-based metrics, like *lines of code* (LOC). But many software evolution approaches [2,7,8,9] depend on object-oriented metrics (e.g. CK-metrics [10]) or more precise complexity-based size metrics (e.g. Halstead or McCabe) that can only be gathered by aggregating the occurences of concrete language constructs and therefore require the analysis of *abstract syntax trees* (AST). Furthermore, other MSR techniques, like *implicit dependencies* [11] or mining for common API-usage patterns [12], also require a language dependent syntax-based analysis.

We hypothesize that MDSE methods and tools like reverse engineering frameworks (e.g. *MoDisco* [13]) and the recent adoption of NoSQL-databases for persisting and processing very large models (e.g. [14,15,16]), allow us to implement a MSR-system that overcomes these issues and fulfils the following goals:

1. the potential to abstract from different programming languages and version control systems
2. syntax-based source code analysis, i.e. analysis of models for corresponding ASTs
3. high accessibility and low programming efforts through high-level languages
4. scalability through NoSQL-based model persistence that enables highly concurrent model processing

We started to implement a model-based MSR-system, coined *srcrepo*[1], in order to verify this hypothesis and research whether the stated goals are achievable.

Note that *srcrepo* only covers the analysis of *source code repositories* and does not deal with other aspects of software repositories, such as issue tracking systems, mailing-lists, Wiki-entries, etc. which are important additional data sources for many MSR techniques.

This paper is organized as follows. First, we describe the process of analyzing repositories with *srcrepo* and introduce all necessary components of our system. After that, we take a detailed look at some of these components: the used metamodel for versioned source code (section 3), our model persistence layer *EMF-Fragments* [16] (section 4), and an OCL-like DSL that can be used to calculate and aggregate software metrics (section 5). The evaluation section 6 discusses our preliminary findings. We finally present related work and conclusions.

2 *srcrepo's* Analysis Process and Components

Fig. 1 shows the basic process of a *srcrepo*-based analysis and all the entities that are involved.

The process starts with existing software projects as they are typically found in (ultra-)large scale software repositories like GitHub or SourceForge (top left). They usually entail a *source code repository* that is maintained by a *version*

[1] `http://github.com/markus1978/srcrepo`

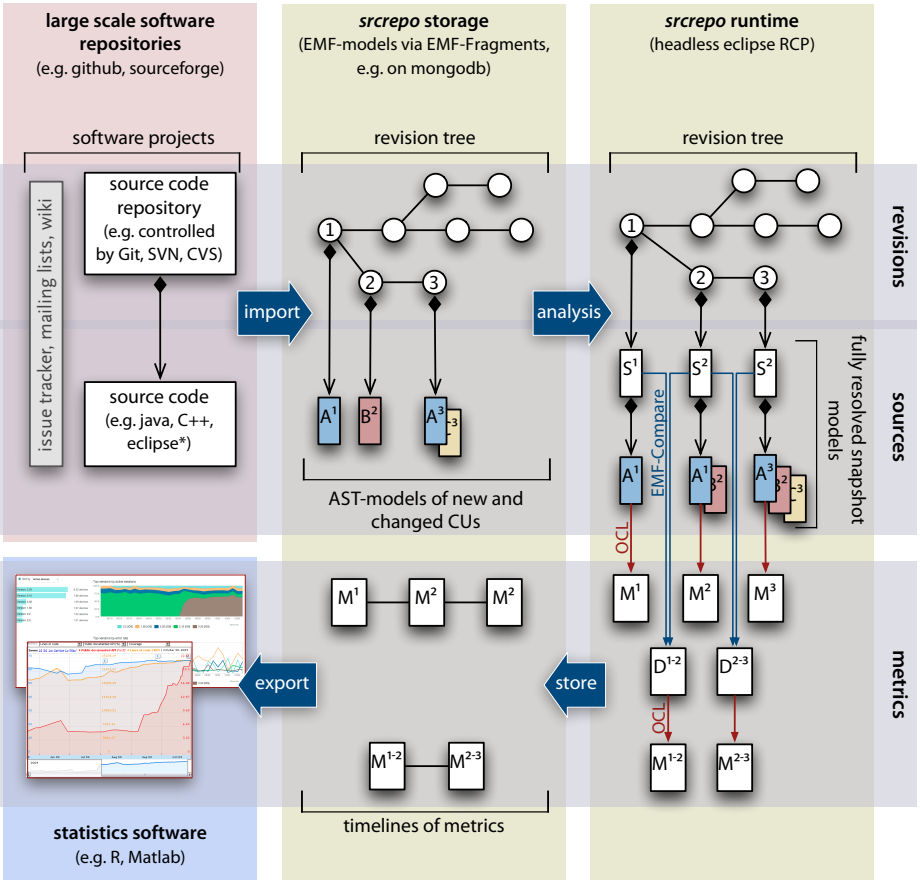

Fig. 1. Schematic visualization of the *srcrepo* analyzes process.

control system like CVS, Git, or SVN. The source code repository provides the actual *source code* organized in files. These files represent the smallest compileable units of source code and are therefore referred to as *compilation units* (*CUs*). Please not that CUs are only an organizational concept and not necessarily relate to a programming language construct, even though CUs sometimes contain a single class or module of the same name. Source code is written in a programming language, but the version control system only treats CUs as plain text files. Software projects also come with other repositories such as issue trackers, mailing lists, etc., which are not processed by *srcrepo*.

Now *srcrepo* provides the functionality to create an EMF-model from source code repositories as a single *import* step. First, *srcrepo* creates a model of the *revision tree*. The revision tree is a lattice of nodes each representing a single commit of changes to the source code repository. Each revision has a unique ID; for simplicity the figure shows revisions with numbers as IDs ($1 \ldots 3$). Each

revision relates to the files that were added, modified, or deleted within the corresponding commit. Currently *srcrepo* uses the JGit Java API to read the revision tree from Git controlled source code repositories. *Srcrepo*'s meta-model for revision trees is explained in the next section. For each file referenced by each revision, *srcrepo* creates a model of the contained source code. Currently *srcrepo* supports Java source code and uses the EMF-based reverse engineering framework MoDisco [13] to create an EMF-model for each compilation unit (depicted via capital letters $A \ldots C$ followed by superscripted revision numbers). MoDisco models are AST-level models that contain instances for all language constructs from classes to literals. Even though MoDisco collects named elements and references within the Java code, the references are not yet resolved because in the current step CUs are processed individually and references may refer to entities in other CUs. But *srcrepo* stores all paths to named elements and references as part of the source code repository model (see the meta-model in the next section). Importing repositories is a rather slow process: checking out each revision in a large repository takes a lot of time. Therefore, we persist the created models. This allows us to repeat the next steps without having to redo the import each time. But, AST-level models are rather large compared to the source code they are taken from. Our experience confirms [17]'s observation of factor 400. For example, the 53 MB Git repository of EMF (*org.eclipse.emf*) is turned into a 20 GB model (using a binary serialization, not XMI). To process such large models, we use *EMF-Fragments* [16] that automatically fragments the model into many small resources that are stored in a NoSQL-database. *EMF-Fragments* is introduced in section 4.

Based on the model created during import, we can now start the actual analysis. *Srcrepo* provides the necessary functionality to traverse the revision tree and to create *snapshots* S^x for each revision. These snapshots contain a model of all the CUs created in all revisions before, not just of the CUs changed in the current revision. Therefore, each snapshot represents the whole codebase at the current revision. *Srcrepo* uses the stored data on named elements and references to resolve all references and create a fully linked model. Of course, we do not create all snapshots at once, but only a couple at a time. This allows us to perform this step within a single runtime (i.e. JVM) without running into memory issues. But this also means that snapshots have to be processed individually. Which is fine, since all software evolution and MSR methods are based on analyzing snapshots sequentially or on analyzing the differences between two successive snapshots.

Clients should have different very accessible options to analyze these snapshots. Currently we are working on the option to use an OCL-like language to count and aggregate occurrences of language constructs (refer to section 5). This is enough to calculate most existing code metrics (depicted by M^x). The language allows clients to write OCL-like expressions that are executed for the whole revision tree. Since snapshots can be analyzed individually, *srcrepo* can run these queries concurrently on different revisions. As future work, we plan to use *EMF Compare* to analyze the differences between snapshots. This is for ex-

ample valuable to find *implicit dependencies* similar to [11], or to analyze typical change patterns/refactorings [18]. The results of *EMF Compare* D^{x-y} can also be processed via OCL. EMF-based model transformation languages are another option for analyzing snapshot models that we need to evaluate. Of course, there is always the possibility to use plain Java code, since all involved models are plain EMF-models.

The artifacts created during analysis (e.g. code metrics, metrics on differences) are also models (e.g. there is a OMG standard/meta-model for organizing software metrics[2]). These result models are also stored within the same storage that is used to persist the repository models. This allows us to maintain cross references between results and the entities that these results were created from (cross references not shown in Fig. 1). For example, we can use *srcrepo* to calculate McCabe's *cyclomatic complexity* for each method and link the resulting numbers to the corresponding methods. Thus, we calculate this metric once and can use it repeatedly in later analysis runs (e.g. use them as weights to calculate the CK-metric *Weighted Methods per Class* (WMC) [10].

As a final step, results are exported for the use in statistics software, such as R or Matlab. The statistics software can then be used to process and analyze the gathered "raw"-data into human readable charts and other forms of usable knowledge.

3 A Meta-Model for Source Code Repositories

Fig. 2 shows the meta-model that we currently use for representing source code repositories.

The top part contains the elements used to model revision trees. A `RepositoryModel` contains revisions (`Rev`) that are connected via `RevRelations` (thus forming a lattice of revisions). Relations between revisions can be navigated both ways. The relation between two revisions contains all `Diffs` between those. A `Diff` can reference a changed file. There can be a reference to a model of the file (`AbstractFileRef`).

In the middle part of the diagram, we have source code related constructs. A `CompilationUnitRef` is a concrete file reference targeting a model of a compilation unit. `PendingElements` and `Targets` are used to store references and named elements within code. We later use this data to resolve all references in snapshots models.

The lower part of the diagram shows (only) some elements from the MoDisco meta-model, which is used to represent the actual Java code. Each `CompilationUnitRef` refers to its own `Model`, i.e. we store a Java model for each CU separately. During analysis, *srcrepo* will merge the models of multiple CUs into snapshots and resolve all references stored within the individual models of the corresponding CUs.

[2] `http://www.omg.org/spec/SMM/1.0/`

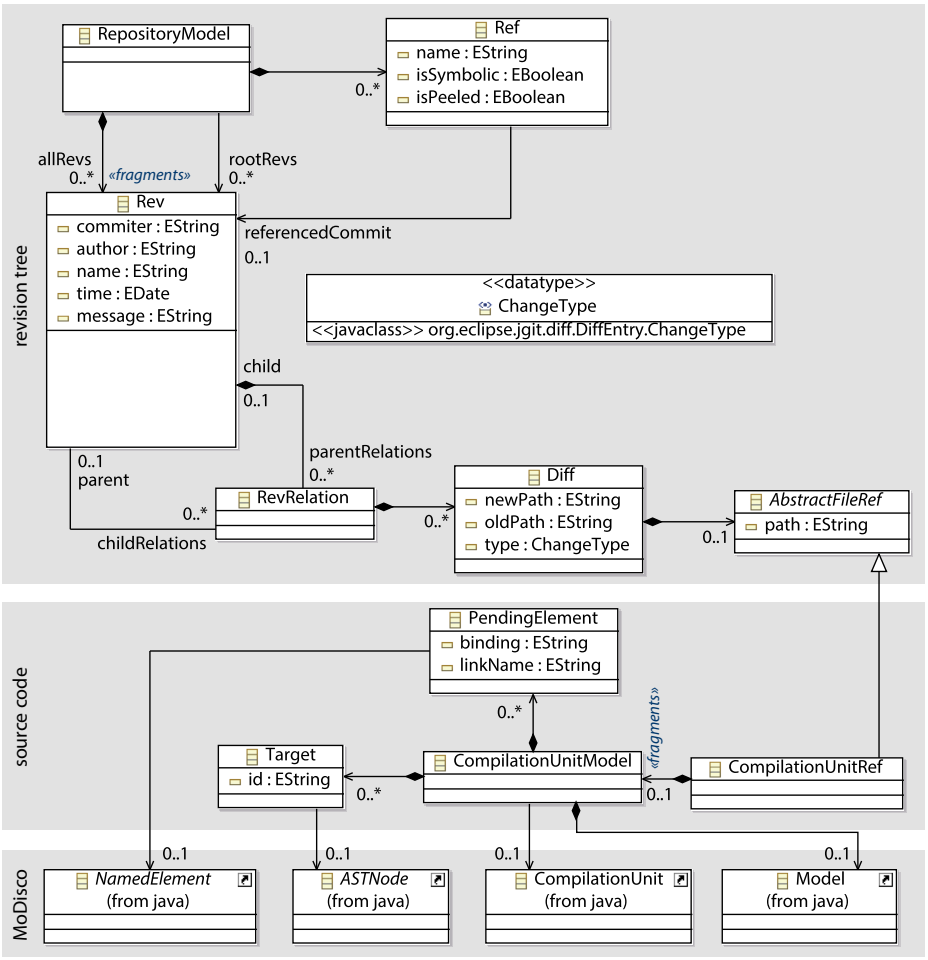

Fig. 2. Meta-model for source code repositories.

4 Model Persistence in NoSQL-Databases

We build a model persistence framework for EMF [19] called *EMF-fragments* [20]. *EMF-Fragments* is different from frameworks based on object-relational mappings (ORM) like Connected Data Objects (CDO). While ORM mappings map single objects, attributes, and references to database entries, *EMF-Fragments* maps larger chunks of a model (*fragments*) to URIs that reference these fragments. Such fragmented models can then be saved to databases that allow us to store maps between keys (URIs) and values (serialized fragments). There is a wide range of such (distributed) data-stores including (distributed) file-systems and document-databases like Hadoop's *HBase* or *mongodb*.

EMF-Fragments uses and extends the regular EMF *resource* API [19]: each fragment is a EMF resource, a fragmented model is a EMF *resource set*. Resources have URIs (key) and can be serialized (value). *EMF-Fragments* uses EMF's *URI converters* to map URIs and serialized resources to database entries. EMF already supports on-demand loading (and later unloading) of resources, and *EMF-Fragments* simply triggers this functionality to automatically and transparently create, delete, save, and unload fragments/resources. *EMF-Fragments* only holds a few fragments in main memory at the same time and therefore can process arbitrary large models with limited main memory. *EMF-Fragments* automatically unloads fragments that are no longer used (*referenced* in Java terms) by clients. Note that at least the largest fragment has to fit into main memory, since fragments have to be loaded as a whole. To fragment a model, clients have to annotate their meta-models and designate references that shall fragment corresponding models. *EMF-Fragments* listens to changes on these references and creates and deletes fragments accordingly. Fig. 4 exemplifies fragmentation on meta-model and model level.

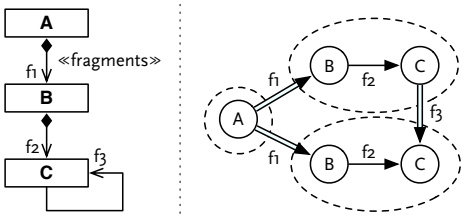

Fig. 3. Fragmentation of models

The *srcrepo* meta-model in Fig. 2 further exemplifies the use of *fragmenting* references, here annotated as UML stereotypes (i.e. with guillemets, set in italic). In consequence each revision with all its `RevRelations` and `Diff` information is stored in an individual database entry. Each `CompilationUnitModel` is consequently stored in another database entry. The assumption is that single revisions and single compilation units will always fit into main memory. On the other hand, we usually analyze whole revisions and whole CUs, and therefore we would not benefit from further fragmentation. Should we, for example, discover that we often only look at parts of a CU (e.g. are only interested in class, field, and method declarations), we could further fragment the model (e.g. mark the reference between declaration and body as fragmenting) and therefore prevent the loading of irrelevant model parts. For a detailed discussion on how to design fragmentation refer to [16].

5 An OCL-like Language for Ascertaining Software Metrics

5.1 Why OCL?

Even though OCL is called the *Object Constraint Language*, it can be used to write expressions with other return types than boolean. OCL was designed to easily navigate between model elements via their associations. To navigate multi-valued association ends comfortably, OCL provides a set of higher-order collection operations that allow to quickly collect, select, and otherwise process association ends. This makes OCL a good language to aggregate data about a model, while navigating that model. The following example OCL expression counts the classes contained in the top-level packages of a MoDisco model.

```
1 context Model:
2   self.ownedElements→collect(p | p.ownedElements)→size
```

Listing 1.1. OCL for collecting all types in all top-level packages of a MoDisco model.

5.2 Why Not OCL?

Despite its merits, OCL was not designed to write complex "programs". OCL can be used to implement functionality but not to declare it. Therefore, concepts to structure OCL code are very limited: there is no way to write parameterized functions for example with-in OCL: callable context for OCL expressions has to be provided out-side of OCL, e.g. in an UML class diagram. Further, its side-effect free design makes it impossible to store results by means of creating and modifying new model elements, e.g. creating and filling a metrics-model.

Therefore, we wanted to extend OCL, or rather create a language that contains what we like about OCL. Similar to [21], where the authors mimic the syntax of model transformation languages in a very extensible internal Scala DSL, we transfered OCL's collection operations to Scala. Filip Krikava presents a way to transfer OCL's higher-order function syntax to Scala's lambda inspired function objects[3]. Besides its flexible syntax, Scala also provides type-inference. This allows us to omit most type information while retaining full static type safety and sensible code assist, which is essential when dealing with complex meta-models such as MoDisco's Java Model. The following shows the expression of the previous example in OCL-like Scala syntax.

```
1 def firstPackageLevelTypes(self: Model):Int =
2  self.getOwnedElements().collect(p⇒p.getOwnedElements()).size()
```

Listing 1.2. Collecting all types in all top-level packages with OCL-like collections in Scala.

[3] http://www.slideshare.net/krikava/enriching-emf-models-with-scala

Instead of defining the context of the expression (line 1), we define a function with corresponding parameter and return type. The resemblance between the OCL expression body and the Scala body is apparent. We implemented these OCL-like collection operations (as declared in Listing 1.3) on top of Java `Iterables`; implicit conversions between `Iterables` and `OclCollections` provide these operations immediately to all Java and Scala collections including EMF's collections. Besides OCL's collect and select operations, we also added a few operations tailored for calculating metrics:

```
1  trait OclCollection[E] extends java.lang.Iterable[E] {
2    ...
3    def collect[R](exp:(E)⇒R):OclCollection[R]
4    def collectAll[R](exp:(E)⇒OclCollection[R]):OclCollection[R]
5    def collectNotNull[R](exp:(E)⇒R):OclCollection[R]
6    def collectClosure(exp:(E)⇒OclCollection[E]):OclCollection[E]
7
8    def select(expr:(E)⇒Bool):OclCollection[E]
9    def selectOfType[T]:OclCollection[T]
10
11   def aggregate[R,I](exp:(E)⇒I,start:()⇒R,aggr:(R, I)⇒R):R
12   def sum(exp:(E)⇒Double):Double
13   def product(exp:(E)⇒Double):Double
14   def max(exp:(E)⇒Double):Double
15   def min(exp:(E)⇒Double):Double
16   def avg(exp:(E)⇒Double):Stats[Double]
17
18   def run(runnable:(E)⇒Unit):Unit
19 }
```

Listing 1.3. Additional OCL-like collection functions defined in Scala.

`CollectAll` collects and flattens the result; `collectNotNull` behaves like collect, but omits `Null` values; `collectClosure` applies the expression recursively to the result until no more new elements are found. `SelectOfType` selects elements of a certain type and returns a collection with casted values. `aggregate` allows to easily implement aggregation. `Sum` for example is implemented as:

```
1 sum(exp:(E)⇒Double):Double =
2   aggregate[Double,Double](exp, ()⇒0, (a,b)⇒a+b)
```

5.3 Example Usage to Calculate CK-Metrics

The following demonstrates the OCL-like collections by implementing three of the CK-metrics [10]: *Weighted Methods per Class* (WMC)[4], *Coupling Between Object classes* (CBO), *Number Of Children* (NOC). To average these metrics

[4] Commonly weighted with unity or McCabe [2]. We use unity here.

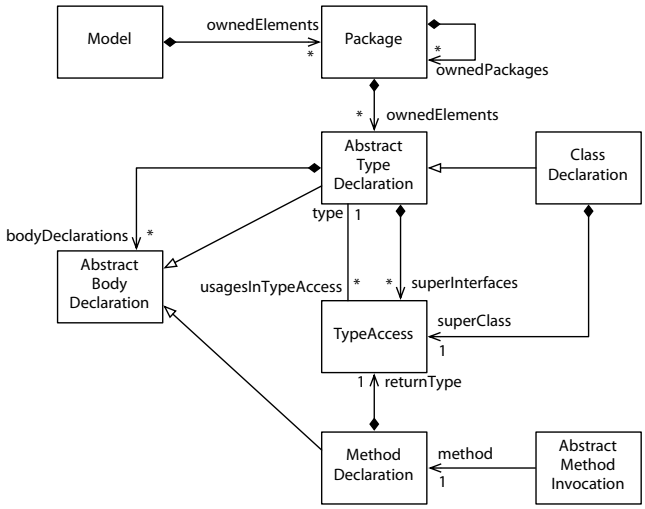

Fig. 4. Simplyfied excerpt of the MoDisco meta-model.

over all classes, we also demonstrate an operation that collects all classes in a model (line 1). Note that all required recursion (i.e. gathering all packages, in all packages, etc. and all the inner classes in all other potential inner classes, etc.) is covered through the use of `collectClosure` (lines 3 and 5). The meta-model excerpt of MoDisco in Fig. 4 explains the navigated classes and associations.

```
1  def classes(model:Model):OclCollection[ClassDeclaration] = model
2    .getOwnedElements()
3    .collectClosure(pkg⇒ p.getOwnedPackages())
4    .collectAll(pkg⇒ pkg.getOwnedElements())
5    .collectClosure(typeDcl⇒ typeDcl
6      .getBodyDeclarations()
7      .selectOfType[ClassDeclaration]
8    )
9
10 def WMC(clazz:ClassDeclaration):Int = clazz
11   .getBodyDeclarations()
12   .selectOfType[MethodDeclaration]()
13   .size()
14
15 def CBO(clazz:ClassDeclration):Int = {
16   val types=new HashSet[AbstractTypeDeclaration]()
17   clazz.eContents()
18     .closure(e⇒ e.eContents())
19     .selectOfType[AbstractMethodInvocation]()
20     .collectNotNull(inv⇒ inv.getMethod())
21     .collectNotNull(meth⇒ meth.getReturnType())
22     .collectNotNull(typeAccess⇒ typeAccess.getType())
```

```
23      .select(typeDcl⇒types.add(typeDcl)
24      .size()
25  }
26
27  def NOC(clazz:ClassDeclaration):Int = clazz
28    .getUsagesInTypeAccess()
29    .select(e⇒e.eContainer()
30      .isInstanceOf[AbstractTypeDeclaration])
31    .size()
32
33  def averageWMC(model:Model):Double =
34    classes(model).avg(clazz⇒WMC(clazz)).value
35  ...
```

Listing 1.4. Some CK-Metrics expressed in Scala with OCL-like collection operations.

5.4 Implementation

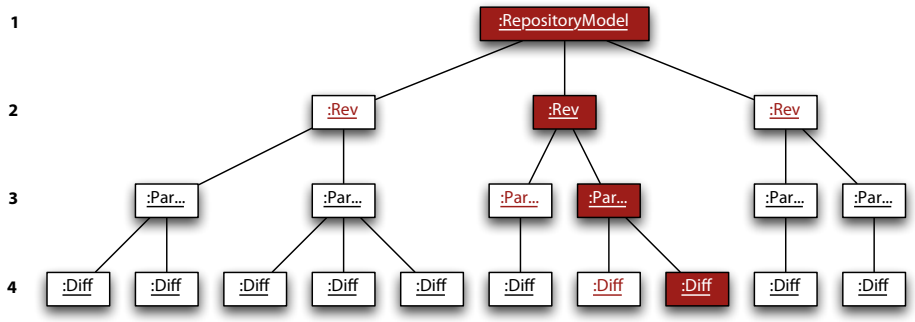

Fig. 5. Object diagram depicting the model elements involved in executing Listing 1.5.

The straight forward method of implementing these OCL-like collection operations is to create a new collection for each operation call to hold the results. This approach however, requires to keep all used collections in memory, even though you are just interested in an aggregation of these interim results. Consider counting all classes in all revisions of a source code repository for instance:

```
1  def countAllClasses(repo:RepositoryModel):Int = repo
2    .getAllRevs().
3    .collectAll(rev⇒rev.getParentRelations())
4    .collectAll(parent⇒parent.getDiffs())
5    .collect(diff⇒diff.getFileRef())
6    .selectOfType[CompilationUnitRef]
```

```
7   .collectAll(classes(cu.getModel()))
8   .size()
```

Listing 1.5. Example aggregation of a large model.

This would mean to create and keep in memory a list of all `ParentRelations`, all `Diffs`, ..., and all `ClassDeclarations`. If we wanted to count all calls of a certain method for example, we had to go even deeper and eventually hold most of the repository model in memory.

To retain scalability, we implemented the collection operations differently. Instead of creating collections that contain all the interim results, we create iterators that behave like collections containing the corresponding results. The iterators only hold references to the current element and loose these references when they move to the next element. Remember that *EMF-Fragments* can automatically unload resources that contain un-referenced model elements. This allows us to navigate the whole model and aggregate data and only have small parts of the model in memory at the same time. Fig. 5 visualizes the difference. With the straight forward approach, all model elements have to be kept in memory in order to count the elements on level 4 (the levels 1-4 represent the results create in line 1-4). With the iterator-based approach, only the red elements have to be loaded at the same time; they represent those elements that are currently collected from the respective iterator positions (white on red ground).

6 Current State, Problems, Limitations, and Future Work

Currently the presented components of *srcrepo* work as described. With respect to the defined four goals (section 1), we encountered the following problems in dealing with heterogeneous repositories (goal 1) and scalability (goal 4).

Our system *srcrepo* currently only supports Git-based source code repositories that contain Java code with Eclipse project meta-data. Most notably this meta-data contains information about which files are actual sources and how the class-path looks like. We are convinced that our revision meta-model can work as an abstraction for other version control systems as well, and we are working on support for SVN as a proof of concept. Supporting other programming languages is a different class of problem. While the use of models, in principle, proliferates abstraction, it is not obvious that a *reasonable* abstraction (i.e. an abstraction that works for MSR) exists. MoDisco for example claims to be an extensible framework, but up to this moments it only supports Java and its meta-model is just an EMF-version of Eclipse's JDT datamodel. OMG's *Knowledge Discovery Metamodel* (KDM) (which, in addition to the given Java meta-model, is also supported by MoDisco) might be such a *reasonable* abstraction, but this has to be evaluated carefully. A different solution is to simply add meta-models for other languages. But this means that all parts of an analysis that are language specific, have also to be implemented for all other programming languages and their respective meta-models. An abstraction of the results (e.g. most software

metrics can be defined for many languages) could still provide value: clients will need to deal with different languages during source code analysis, but not for studying the resulting metrics.

We also encountered performance issues that currently prevent a reasonable application of *srcrepo* on a large number of real life software repositories. These problems have three causes.

First, creating a snapshot model for each revision individually, involves a lot of repeated computation, since only a small part of the underlying *compilation unit*s (CU) actually changes from revision to revision. We are working on an incremental snapshot creation that merges differences into the snapshots of previous revisions and therefore reduces the workload drastically.

Secondly, CUs are atomic to *srcrepo*. When a CU changes, *srcrepo* will process that changed CU as a whole, even if only a small part has changed. This is fine for typically sized CUs, but in some seldom cases (especially when code generation is involved) CUs become very big. For example, code repository of *EMF* itself contains a >3 MB CU. This massive Java file with >600.000 LOC has various aspects of EMF generated into it. Not only is it very large, but it also has a lot of dependencies to other parts of the source code. Thus, it also changes very often and therefore makes the problem even bigger. Obviously we have to use a more granular unit as the smallest changeable part. Unfortunately however, CUs are the smallest common nominator between the syntax-based view and the text-file-based view that version control systems offer. CUs and files can be directly mapped onto each other: each CU corresponds to a file. Finer units like class members on one side and distinct lines of text on the other side are much harder to map to each other. Therefore, we will always have to parse the whole CU, but we do not necessarily have to convert the whole AST into a model, and we certainly don't have to store the whole CU model. We can either map text-based difference information from the version control system onto the AST to determine which elements have actually changed, or employ some form of model comparision (e.g. *EMF Compare*) on ASTs/models.

Thirdly, at first glance it should be easy to run most of an analysis in parallel, since all revisions can be analyzed individually. But, things become more complicated, if we introduce incremental snapshot creation as described as a possible solution to the first problem. We still have to implement concurrency that is sensitive to this issue.

As a general last limitation, *srcrepo* only analyses source code repositories. For many research (e.g. [12,2,7]) in MSR this has to be integrated with other systems to analyze source code repositories in unity with other parts of software repositories, e.g. issues-tracking systems, mailing lists, Wiki's, etc.

7 Related work

The field *Mining Software Repositories* is as old as software repositories; an overview of recent research can be found here [1]. A recent facet of this research

is gathering large metrics-based data-set from large and ultra-scale repositories. This is also what our framework aims at.

The Floss project (EU Framework Programme 7) [4] gathered per revision data-sets of language independent text-based metrics from more than 3000 libre software projects. Their tool *Alitheia* [6] not only gathers metrics from source code (CVS, SNV, and Git), but also data from issue tracking and mailing lists. Thereby, the project goal was not to analyze this data, but to create a comprehensive common database for other researchers. *Sourcerer* [5] is a example for a language-dependent approach. In this project, over 4000 libre software Java projects have been mined for metrics based on class, field, and method declarations [22]. But, the project only gathered data from released revisions, not for whole repositories. Similar projects and tools are *BOA* [3] and Harmony[5].

Another source for related work is the recent adoption of NoSQL-databases [23,24,25] for the persistence of large models [26]. The use of document or graph databases promises better performance and scalability than traditional *object relational mapping* (ORM)-based technologies like CDO[6] or Teneo[7]. In [26,15] the authors implement EMF-persistence for graph databases. Morsa [14] stores individual objects as JSON-records in the document-database mongodb. Our own *EMF-Fragments*[8] stores individual EMF-resources in document-databases like HBase or mongodb. These approaches use different strategies to map objects and relations to their respective database-technology [27].

There are also attempts to create version control systems for models; [28] provides an overview of recent research. The approach in [29] is (to our knowledge) the first approach that uses a NoSQL-backend.

8 Conclusions

We presented *srcrepo*, a model-based system for the analysis of source code repositories and a proof of concept for a model-based approach to *Mining Source Code Repositories*. We presented 4 goals: (1) to abstract from heterogeneous repositories, (2) achieve syntax-based and not text-based analysis, (3) high accessibility, and (4) scalability. In respect to goal (1), we started to implement support for a single type of version control system and programming language. Consequently, we could not yet prove a possible abstraction from different programming languages and version control systems. But the model-based approach still offers this potential in principle. In respect to (2) and different from comparable systems, we could realize a language dependent AST-level deep analysis. Furthermore, clients only have to write small OCL-like expressions to gather language dependent metrics from a vast amount of available software projects (goal 3). Although, all used technologies and components are prepared for concurrent ex-

[5] `http://code.google.com/p/harmony`
[6] `http://www.eclipse.org/cdo/`
[7] `http://www.eclipse.org/modeling/emft/?project=teneo`
[8] `http://github.com/markus1978/emf-fragments`

ecution, we encountered several issues that we discussed possible solutions for, and we are confident to realize goal (4) in the near future.

References

1. Kagdi, H., Collard, M.L., Maletic, J.I.: A survey and taxonomy of approaches for mining software repositories in the context of software evolution. Journal of Software Maintenance and Evolution: Research and Practice **19** (2007) 77–131
2. Gyimothy, T., Ferenc, R., Siket, I.: Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans. Softw. Eng. **31** (2005) 897–910
3. Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N.: Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: Proceedings of the 2013 International Conference on Software Engineering. ICSE '13, Piscataway, NJ, USA, IEEE Press (2013) 422–431
4. consortium, F.: Flossmetrics final report: Free/libre/open source metrics and benchmarking. Technical report, FLOSSMetrics consortium (2010)
5. Bajracharya, S., Ossher, J., Lepos, C.: Sourcerer: An internet-scale software repository. In: Proceedings of SUITE'09, an ICSE'09 Workshop, Vancouver, Canada (2009)
6. Gousios, G., Spinellis, D.: A platform for software engineering research. In Godfrey, M.W., Whitehead, J., eds.: MSR, IEEE (2009) 31–40
7. Basili, V.R., Briand, L.C., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. IEEE Trans. Softw. Eng. **22** (1996) 751–761
8. Subramanyam, R., Krishnan, M.S.: Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. IEEE Trans. Softw. Eng. **29** (2003) 297–310
9. Yu, P., Systä, T., Müller, H.A.: Predicting fault-proneness using OO metrics: An industrial case study. In: Proceedings of the 6th European Conference on Software Maintenance and Reengineering. CSMR '02, Washington, DC, USA, IEEE Computer Society (2002) 99–107
10. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20** (1994) 476–493
11. Zimmermann, T., Weißgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. IEEE Trans. Software Eng. **31** (2005) 429–445
12. Livshits, V.B., Zimmermann, T.: Dynamine: finding common error patterns by mining software revision histories. In Wermelinger, M., Gall, H., eds.: ESEC/SIGSOFT FSE, ACM (2005) 296–305
13. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: Modisco: A generic and extensible framework for model driven reverse engineering. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. ASE '10, ACM (2010) 173–174
14. Espinazo Pagán, J., Sánchez Cuadrado, J., García Molina, J.: Morsa: A scalable approach for persisting and accessing large models. In Whittle, J., Clark, T., Kühne, T., eds.: Model Driven Engineering Languages and Systems. Volume 6981 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2011) 77–92
15. Benelallam, A., Gómez, A., Sunyé, G., Tisi, M., Launay, D.: Neo4EMF, a scalable persistence layer for EMF models. In: ECMFA- European conference on Modeling Foundations and applications, York, UK, Royaume-Uni, Springer (2014)

16. Scheidgen, M., Zubow, A., Fischer, J., Kolbe, T.H.: Automated and transparent model fragmentation for persisting large models. In France, R.B., Kazmeier, J., Breu, R., Atkinson, C., eds.: Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings. Volume 7590 of Lecture Notes in Computer Science., Springer (2012) 102–118
17. Kagdi, H.H., Collard, M.L., Maletic, J.I.: Towards a taxonomy of approaches for mining of source code repositories. ACM SIGSOFT Software Engineering Notes **30** (2005) 1–5
18. Williams, C.C., Hollingsworth, J.K.: Automatic mining of source code repositories to improve bug finding techniques. IEEE Trans. Software Eng. **31** (2005) 466–480
19. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. 2nd edn. Addison-Wesley, Boston (2009)
20. Scheidgen, M.: EMFFrag – Meta-Model-based Model Fragmentation and Persistence Framework. http://github.com/markus1978/emf-fragments (2012)
21. George, L., Wider, A., Scheidgen, M.: Type-safe model transformation languages as internal DSLs in Scala. In Hu, Z., de Lara, J., eds.: ICMT. Volume 7307 of Lecture Notes in Computer Science., Springer (2012) 160–175
22. Cox, A., Clarke, C., Sim, S.: A model independent source code repository. In: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research. CASCON '99, IBM Press (1999) 1–
23. Khetrapal, A., Ganesh, V.: HBase and Hypertable for large scale distributed storage systems a performance evaluation for open source Big-table implementations. Technical report, Purdue University (2008)
24. Lakshman, A., Malik, P.: Cassandra: Structured storage system on a P2P network. In: Proceedings of the 28th ACM symposium on Principles of distributed computing. PODC '09, New York, NY, USA, ACM (2009) 5–5
25. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Proceedings of twenty-first ACM SIGOPS symposium on Operating Systems Principles. SOSP '07, New York, NY, USA, ACM (2007) 205–220
26. Barmpis, K., Kolovos, D.S.: 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 (2012)
27. Scheidgen, M.: Reference representation techniques for large models. In: Proceedings of the Workshop on Scalability in Model Driven Engineering. BigMDE '13, ACM (2013) 5:1–5:9
28. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. IJWIS **5** (2009) 271–304
29. Barmpis, K., Kolovos, D.: Hawk: Towards a scalable model indexing architecture. In: Proceedings of the Workshop on Scalability in Model Driven Engineering. BigMDE '13, New York, NY, USA, ACM (2013) 6:1–6:9