Creating and Analyzing Source Code Repository Models
A Model-based Approach to Mining Software Repositories

Markus Scheidgen, Martin Smidt and Joachim Fischer
Institut für Informatik, Humboldt-Universität zu Berlin, Unter den Linden 6, Berlin, Germany

Keywords: Reverse Engineering, Software Evolution, Metrics, Mining Software Repositories, Metamodels, OCL.

Abstract: With *mining software repositories* (MSR), we analyze the rich data created during the whole evolution of one or more software projects. One major obstacle in MSR is the heterogeneity and complexity of source code as a data source. With model-based technology in general and reverse engineering in particular, we can use abstraction to overcome this obstacle. But, this raises a new question: can we apply existing reverse engineering frameworks that were designed to create models from a single revision of a software system to analyze all revisions of such a system at once? This paper presents a framework that uses a combination of EMF, the reverse engineering framework Modisco, a NoSQL-based model persistence framework, and OCL-like expressions to create and analyze fully resolved AST-level model representations of whole source code repositories. We evaluated the feasibility of this approach with a series of experiments on the Eclipse code-base.

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 (Kagdi et al., 2007). 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 that are computed for each software revision to understand the evolution of software projects (Gyimothy et al., 2005).

Recent advances in big-data processing (i.e. Map/Reduce) allowed to extend this research to *large-* or even *ultra-large*-scale software repositories that comprise a large number of software projects (Dyer et al., 2015). Examples for large-scale repositories are the *Apache Software Foundation* or the *Eclipse Foundation*, and ultra-large-scale repository examples are software project hosting services (sometimes refered to as *open-source software forges* (Williams et al., 2014b)) like *Github* (250.000+ projects) or *Source-Forge* (350.000+ projects) (Dyer et al., 2015).

But analyzing many heterogeneous software projects has limits. While existing approaches (Bajracharya et al., 2009; Gousios and Spinellis, 2009; Dyer et al., 2015) manage to abstract from different *code versioning systems* (e.g. CVS, SVN, Git), different programming languages are still an issue. Especially, if the full AST's of the code-base is necessary: A lot of software evolution and MSR research (Gyimothy et al., 2005; Basili et al., 1996; Subramanyam and Krishnan, 2003; Yu et al., 2002) depends on object-oriented metrics (e.g. CK-metrics (Chidamber and Kemerer, 1994)) or more precise complexity-based size metrics (e.g. Halstead or McCabe) that aggregate the occurences of concrete language constructs and therefore require the analysis of *abstract syntax trees* (AST). Furthermore, other MSR techniques, like mining for common API-usage patterns (Livshits and Zimmermann, 2005) or *design structure matrices* (Milev et al., 2009) also require a language dependent syntax-based (AST-level) analysis.

Based on different goals, reverse engineering is used to find abstract representations of source code that are suited to derive knowledge from existing code-bases (Chikofsky et al., 1990). Concrete reverse engineering frameworks (e.g. *Modisco* (Bruneliere et al., 2010)) suggest a multi-stage model transformation process that goes from source code over language dependent models (e.g. via Modiscos' Java metamodel or OMG's ASTM) to language independent information representations (e.g. via the OMG metamodels KDM and SMM). We can therefore assume that reverse engineering can solve the mentioned heterogeneity to create MSR applications more

Scheidgen M., Smidt M. and Fischer J.
Creating and Analyzing Source Code Repository Models - A Model-based Approach to Mining Software Repositories.
DOI: 10.5220/0006127303290336
In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, pages 329-336
ISBN: 978-989-758-210-3
Copyright © 2017 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

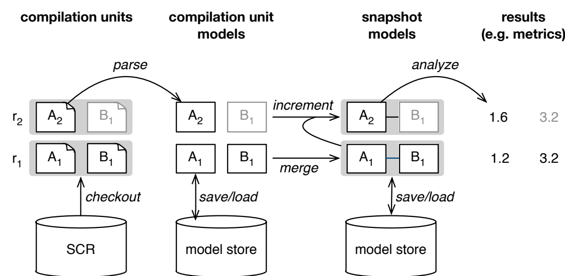

Figure 1: Different artifacts and operations involved in model-based MSR.

efficiently with model-based technology like meta-models, and query/transformation languages. But, we still need to show that model-based technology can scale from processing individual repository snapshots to complete reversion histories.

In this paper, we evaluate the scalability of model-based MSR. We build a model-based MSR framework called *Srcrepo* on top of EMF and Modisco (Bruneliere et al., 2010) as described in a previous publication (Scheidgen and Fischer, 2014). Here, we evaluate the scalability with the code-base of the Eclipse Foundation as an example for a large-scale software repository. In this evaluation, we gather datasets from a subset of the CK-metrics suite (Chidamber and Kemerer, 1994). The following section 2 gives a brief overview of our approach and framework. Section 3 evaluates the scaleability of the approach. We present related work and conclusions in sections 4 and 5.

2 A MODEL-BASED MSR FRAMEWORK

We assume that in general a MSR application needs to visit each revision of a source code repository and that it needs to analyze each revision based on a fully resolved AST-level model representation of all code in that revision (*snapshot* model). Figure 1 shows the operations and artifacts involved in model-based MSR.

Each analysis starts with a *source code repository* (SCR). Here, source code is organized in *compilation units* (CU) (e.g. Java source files) which are themselves organized in revisions. All version control systems provide a checkout operation to access individual revisions of individual CUs. After checkout, we need to transform CUs into CU models through parsing. Of course the code from different CUs is not independent and we need to merge models for individual CUs into snapshot models that represent all code in one revision. This can either be done by merging all CUs from one revision or by incrementally updating an existing older snapshot model with data from changed CUs. References between CUs have to be resolved for each snapshot. Finally, we need to actually analyze the source code models (e.g. compute metrics data). The intermediate model representations for CUs and snapshots can be stored to omit all previous steps in future analysis runs.

Srcrepo is build on top of Eclipse and the following frameworks and libraries: JGit, Modisco (Bruneliere et al., 2010), EMF, JDT, Xtend, and EMF-Fragments (Scheidgen et al., 2012).

Clients can transform existing Git repositories and the contained Java source code into EMF-models (refer to Figure 2 for the meta-model). This is a two step process. First, *Srcrepo* creates a model of the revision tree. The revision tree is a lattice of nodes. Each node represents a single commit of changes to the source code repository. Each revision relates to the files that were added, modified, or deleted within the corresponding commit. *Srcrepo* uses the JGit plumbing API to read the revision tree from Git controlled source code repositories. Secondly, *Srcrepo* performs a checkout on each revision of the tree using JGit's porcelain API. For all differences in each revision that refer to Java source files, *Srcrepo* creates a model of these *compilation units* (CU) with Modisco (Bruneliere et al., 2010). Modisco uses JDT to parse and analyze the Java code. The resulting CU models are AST-level models that contain instances for all language constructs from classes to literals. Modisco collects all named elements and references within the Java code. At this point, we only resolve local references within the same CU, because we process each CU individually. All names and the remaining unresolved references are added to the model and are later used to resolve cross-references in snapshot models. The CU models are added to the revision tree model. The whole model is persisted with EMF-Fragments (Scheidgen and Zubow, 2012) in a MongoDB.

Once this repository model is created, clients can use *Srcrepo* to analyze it repeatedly. *Srcrepo* can traverse the revision tree and can create snapshot models for each visited revision. These snapshots contain a model of all current CUs, not just of the CUs that were changed in the last revision. Therefore, each snapshot represents the whole code-base of the currently visited revision. *Srcrepo* uses the stored data on named elements and references to resolve all references and creates fully resolved snapshot models. Of course, *Srcrepo* does not create all snapshots at once, but only one 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 based on the assumption that typical MSR applications only need to analyze snapshots sequentially or only need to analyze the differences between two successive snapshots. If possible, *Srcrepo* incrementally updates snapshots. It takes the snapshot model from the previous revision and only processes the CUs that have changed in the current revision. For each changed CU, *Srcrepo* first removes the old version, unresolves all references to this CU, and then adds the new version to the snapshot and re-resolves all references.

Srcrepo offers clients different options to analyze snapshot models. First, they can use an OCL-like language to count and aggregate occurrences of language constructs. This is sufficient to calculate most existing code metrics. For metrics that only rely on containment and do not require reference data, clients can skip snapshot creation on calculate metrics on the bare CU models. The CU and snapshot models are regular EMF-models. Therefore, clients can use model comparison frameworks such as *EMF compare* to identify and analyze the matches and differences between snapshots. This is for example valuable to analyze typical change patterns/refactorings (Williams and Hollingsworth, 2005). Since matches and differences can themselves be represented as models, they could also be processed via OCL. EMF-based model transformation languages are a third option. Finally, there is always the possibility to use plain Java code to analyze all models.

The artifacts created during analysis (e.g. code metrics, difference models, metrics on differences, etc.) could also be represented as models (e.g. instances of OMG's SMM or KDM metamodels). These result models could also be stored within the same storage that is used to persist the repository models. Clients could also maintain cross-references between model elements that represent results and model elements that these results were created from. 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 following analysis runs (e.g. use them as weights to calculate the CK-metric *weighted methods per class* (WMC) (Chidamber and Kemerer, 1994), also refer to section 3).

As a final step, clients need to export the produced data (e.g. as XML, JSON, comma separated values) to load the data into 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.

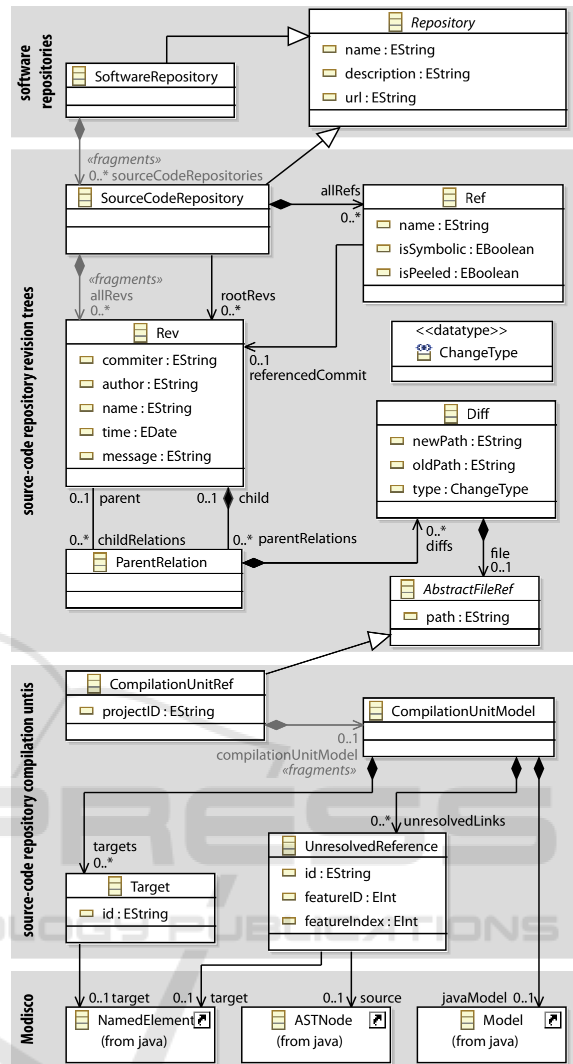

Figure 2: *Srcrepo*'s meta-model for repository and compilation unit models.

3 EXPERIMENTS

3.1 Setup

The subject for our experiments is a corpus that comprises a subset of the Eclipse Foundation's source code repositories. The Eclipse Foundation maintains code (and other artifacts, like documentation, webpages, etc.) in over 600 Git repositories. Each repository contains the code of one Eclipse project or subproject. The repositories themselves may contain the code of several Eclipse plug-ins. We took the largest

200 of those repositories that actually contained Java code. These 200 repositories contain about 600 thousand revisions with a total of over 3 million *compilation unit* (CU) revisions that contain about 400 million SLOC (lines of code without empty lines and comments). The Git repositories take 6.6 GB of disk space. The generated model representation of these repositories comprises over 4 billion objects and takes about 230 GB of diskspace in EMF-Fragment's binary serialization format. The model of all 200 repositories is a single connected 4 billion object EMF-model.

We run all experiments on a server computer with four 6-core Intel Xeon X74600 2.6 GHz processors and 128 GB of main memory. However, *Srcrepo* operates mostly in a single thread with the exception of JVM and Eclipse maintenance tasks. *Srcrepo* runs on Eclipse Mars' versions of EMF, Modisco, JDT, JGit, and Xtend in a Java 8 virtual machine. All operations were run with a maximum of 12 GB of heap memory. The database runs locally in a not distributed MongoDB 3.0.6. The database store lays on a regular local hard disk drive. All Git working copies and corresponding operations were performed on the same local hard drive. However as the data will indicate, hard disk IO is not the issue and disk activity was relatively throughout all experiments. The system runs on GNU/Linux with a 3.16 kernel. *Srcrepo* operations are long running computations over thousands of revisions and CUs and respective algorithms are invoked over and over again. As usual for such macro-benchmark measures, we are therefore ignoring JIT warm-ups and other micro-benchmark related issues (Wilson and Kesselman, 2000).

3.2 Execution Time

First, we compare the execution time for creating a repository model with the time necessary to analyze it. Can we create repository models in *reasonable* time? Does the transformation/analysis execution time difference justify the added efforts to persist the created models? Snapshot models may only be necessary for certain kinds of analyses. How much of the execution time has to be dedicated to which operation?

To answer these questions, we measure the accumulated execution time of the operations introduced in section 1 checkout, parse, save, load, merge/increment, and analysis. *Checkout* comprises all version control system operations necessary to checkout all revisions of the repository (i.e. everything Git does). *Parse* refers to all actions necessary to derive model representations of the source code (i.e. everything Modisco does). *Save/load* refers to the time for persistence operations (i.e. everything EMF-Fragments does). *Merge/increment* refers to the operations that incrementally merge snapshot models from individual CU models. Analysis refers to the *user defined function* (UDF) that constitutes the actual analysis (i.e. the client code that is executed on the provided snapshot models).

In our experiments the UDF calculates the *weighted methods per class* (WMC) metric with Halstead length as weight. The UDF also counts WMC with fixed weight 1, number of Java types, and of overall model elements. To measure the Halstead length, we count methods and method calls in addition to operators and operator usages. This computation relies on resolved method calls. Therefore, this example analysis is a representative of the class of analyses that requires fully resolved snapshot models. A simpler version counts only standard Java operators. This is an example that does not require such resolved models and can be computed from individual CU models.

Figure 3 shows the execution times for 10 large example projects accumulated over all revisions. The left bars represent the model creation operations and the right bars the analysis operations. The box plot in Figure 5 shows the same measurements for all projects as an average per revision.

The figures show that the execution times of *save*, *load*, and our example *UDF* have a lower magnitude than those of *checkout*, *parse*, and *merge/increment*. *Save*, *load*, and in the case of simple size metrics also *UDF* basically boil down to simple model containment traversals with the added efforts for saving and loading binary representations to/from the database. These are very simple operations with linear complexity in model size.

At first glance, *checkout* should be a very fast operation as well; it should be comparable to loading or saving CU models. But, Git compresses many revisions of the same file into a so called *pack* file. The whole pack file has to be decompressed in order to obtain a single file revision. Since each checkout is an individual operation, Git has to decompress pack files containing many file revisions for each individual revision again and again. This also explains the wide distribution of average *checkout* times over all repositories: pack files tend to be larger for larger repositories. Git was simply not designed to successively checkout all revisions.

Less surprising are the results for *parse*. It is a complex operation that includes refreshing the underlying Eclipse workspace, reading and parsing of Java code files, checking static semantics, creating

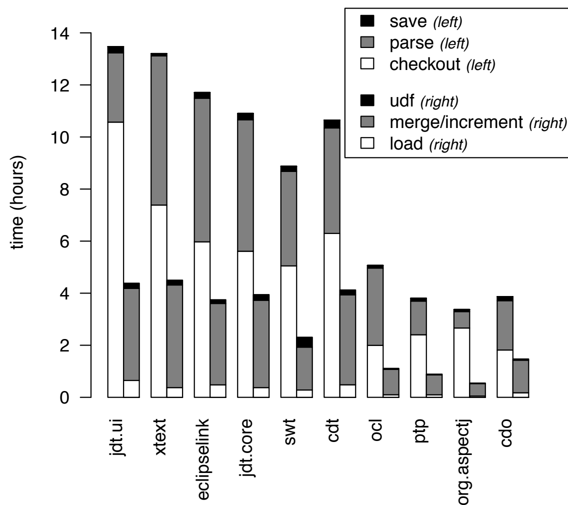

Figure 3: Execution times for model creation and analysis related operation for 10 example projects accumulated for all revisions.

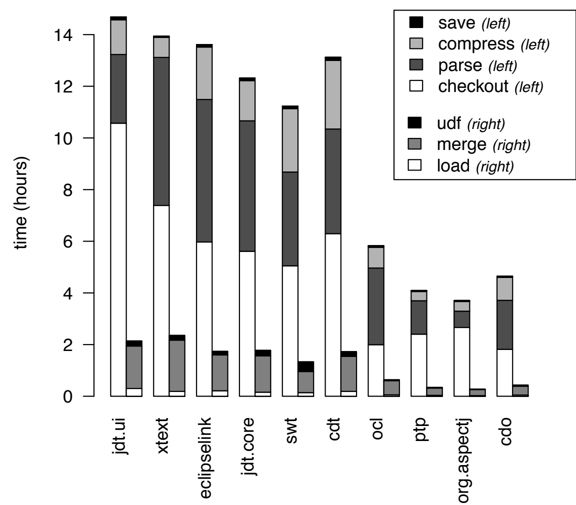

Figure 4: Hypothetical execution times for model creation and analysis with delta-compression based on meta-class-based matching.

EMF-models from ASTs, providing name-tables, and resolving all local references. *Merge/increment* has similar execution times. This is also not surprising, since it processes similar data as *parse*. To update a snapshot model, this operation has to remove changed CU data from the snapshot, replace it with the data from the new revision of the same CU, and re-resolve references. Both of these operations have to process CUs as a whole, even though only a small part of the CU might have changed between revisions. Different dimensions of average CU sizes (especially for projects that incorporate generated or boilerplate heavy code as part of their code-base) explain the wide distribution of execution times between different repositories. Larger CUs tend to have more dependencies between each other, which leads to larger and more often changed CUs, which leads to more changes between revisions and therefore longer execution times for *parse* and *merge/increment*.

For an analysis based on snapshot models, the differences between model creation and analysis execution times are apparent, but they have similar magnitudes. Therefore, we do not gain much from existing models for successive analysis runs. For an analysis where we can omit merge/increment and can operate on individual CU models, the situation is different. Model creation takes hours, analyses run in minutes.

It is hard to compare our system to other MSR tools, because most existing tools (section 4) either do not support a AST-level analysis or analyze only parts of the code-base. Different MSR applications also have different performance characteristics, since they require vastly different functionality. As a rough comparison, Livshits and Zimmermann describe a similar AST-level analysis performed on a smaller part of the Eclipse code-base of about 4 million SLOC that they claim completed in about 24h on a slightly slower system. Similar to our results, their UDF (API pattern mining) required only very low computing effort once the AST-data was created. In general, MSR is often (especially for large-scale repositories) done in distributed computing environments and hours to days of computing seems to be accepted as a *reasonable* time frame (Gousios and Spinellis, 2009; Bajracharya et al., 2009; Livshits and Zimmermann, 2005; Dyer et al., 2015).

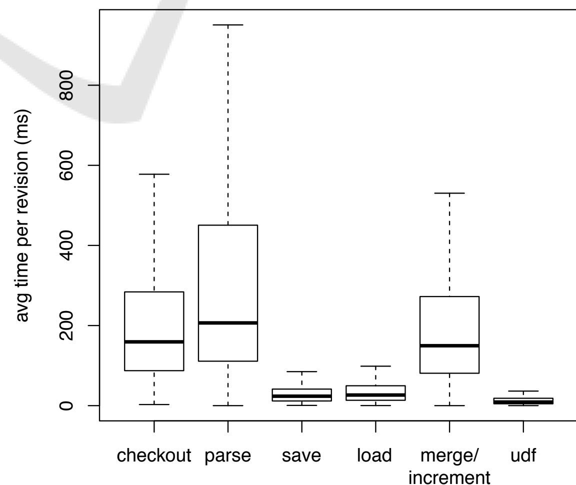

Figure 5: Execution times for model creation and analysis related operations for all measured projects as averages per revision.

3.3 CU Comparison and Compression

Usually only a small part of a CU changes between revisions, and most of the *increment* time in the previous experiment is wasted on those parts in a CU that do not change between revisions. If there was a way to identify matches and differences between two revisions of the same CU, we might reduce the *increment* execution time significantly.

Existing frameworks for model comparison (Kolovos et al., 2009) use two steps. First, model elements are matched, i.e. pairs of elements in both model revisions that represent the same element are identified. In a second step, framework find the differences between matching elements. Here, frameworks apply the same *minimum editing distance* algorithms as regular line-by-line *diff* programs. However, while matching lines of code is pretty straight forward, matching model elements is not. Existing model comparison frameworks employ rather complex heuristics (based on names, size, place, structure, etc.) to establish matching elements. Their goal is to find the smallest possible set of differences. But, the comparison quality has to be traded for comparison execution time.

We therefore started to experiment with two simpler, less time consuming matching algorithms. The first algorithm matches elements only by names (and signatures, i.e. for methods): two named elements match if they have the same name and do not match otherwise. All elements without names (e.g. the contents of code-blocks) only match if they are equal. A second algorithm also matches named elements by name, but all other elements are matched if they have the same meta-class. We compare the results of these algorithms with traditional line-based comparison on the original Java files.

All three provide vastly different granularity for comparison. With named element matching, a method is completely replaced even though only a small part of it has changed. With meta-class-based matching, all elements are considered for matching, but their meta-class is only a very weak measure of similarity. The basic rationale behind this is that in a code block consecutive statements have mostly different meta-classes. Line-based comparison is somewhere in between, with the added benefit that comparing two lines is very robust and fast. Please note that line-based matching is only added here to have a base-line to compare the other algorithms to. Line-based comparison itself does not help our problem, because we cannot easily map line matches and differences to matches and differences in models.

Using comparison to process differences between models is a form of delta compression. The result of matching and finding differences is a difference- or delta-model. With such a delta-model and one of the compared original models, we can later reconstruct (de-compress) the other original model. Of course, we can use sequences of delta-models to delta-compress successive revisions.

We applied this form of delta-compression to the compilation units in the created repository models. Figure 6 shows the results. The top row of bar plots compares the size of the original repositories (left bars) with the size of compressed models (right bars) for some example projects. For model-based comparison, size is plotted in terms of serialized model size in bytes; for line-based comparison we user number of lines. The compressed size is split into uncompressed initial CU revisions and compressed CU revisions (i.e. delta-models). The box plot on the lower left shows the compression data for all projects relative the original uncompressed size.

Apparently, the simple named elements-based matching performs badly, but the still simple meta-class-based matching get close to line-based matching. If we compare the execution time for named element- and meta-class-based matching in terms of time needed to find the differences and time needed to reproduce models from delta-modes (lower middle and right), named elements-based matching also performs much worse. Even though the matching itself is simpler with named elements-based matching, it requires much more consequential comparisons of contained model elements than the meta-class matching algorithm. Compared to the time necessary to parse all CUs, compression only requires a reasonable addition in execution time.

We did not yet implement compression-based snapshot increments, but we can extrapolate the execution times for model creation and analysis with compression. We assume that *save*, *load*, and *increment* execution times scale linearly with the amount of compression and that the execution time for compressing needs to be added to model creation. Figure 4 shows the hypothetical results. Now, even a snapshot-based analysis runs a magnitude faster than model creation and this strategy promises significantly performance gains for sequential analyses.

4 RELATED WORK

The field *Mining Software Repositories* is as old as software repositories; an overview of recent research can be found here (Kagdi et al., 2007). One facet of this research is gathering large metrics-based data-set

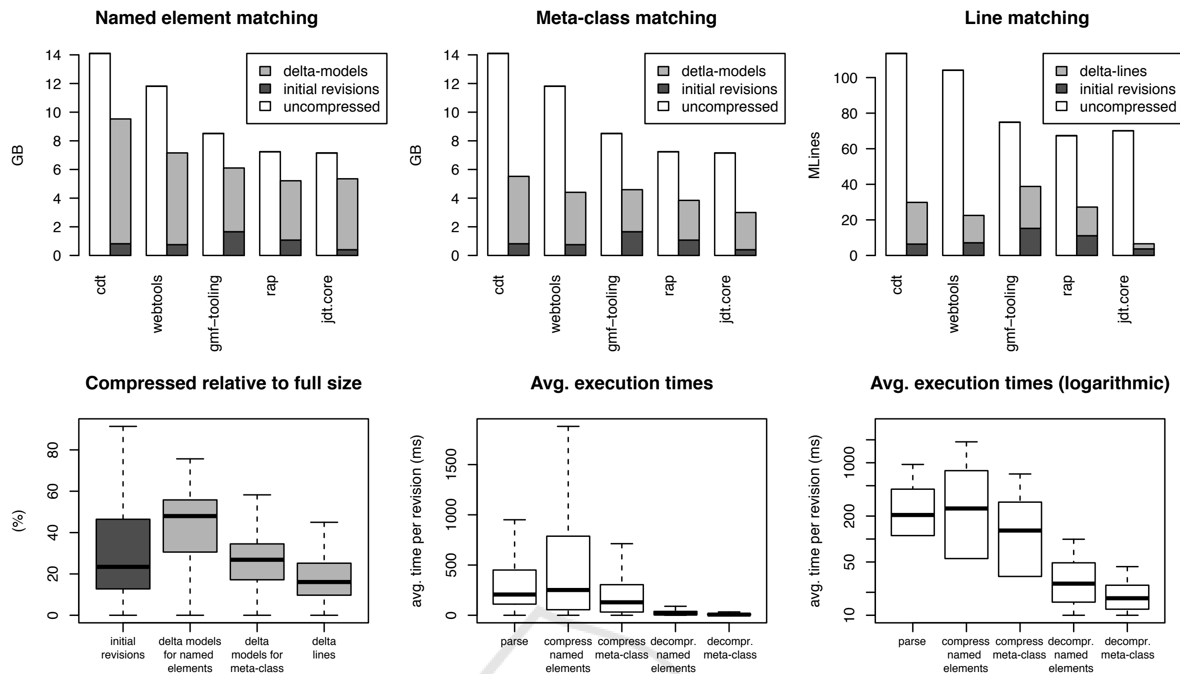

Figure 6: Compression related measurements.

from large and ultra-scale repositories. This is also what our framework aims at.

A lot of projects that create MSR-datasets with traditional text- or grammar-based technology exist (Gousios and Spinellis, 2009; Bajracharya et al., 2009; Falleri et al., 2013). *BOA* (Dyer et al., 2015) even produces AST's of complete revision histories and a query-language for analysis of Java repositories. While these approaches, specifically design from scratch for MSR, exceed our work in performance, they provide little means for possible future abstraction and multi-language support.

The OSS-Meter project uses model-based technology to measure and analyze source code repositories, communication channels, and bug-tracking systems, and other relevant meta-data to measure the quality and activity of open-source software projects. This includes efforts to abstract from different (ultra-)large-scale software repositories (a.k.a. open-source software forges) with a common metamodel (Williams et al., 2014b).

MSR should not be limited to source code written in general purpose programming languages. It can also be applied to repositories containing models and metamodels as Williams et al. and DiRocco et al. suggest (Williams et al., 2014a; Di Rocco et al., 2014). There are also attempts to create version control systems for models; (Altmanninger et al., 2009) provides an overview of recent research. The approach in (Barmpis and Kolovos, 2013) is (to our knowledge) the first approach that uses a NoSQL-backend.

5 CONCLUSIONS

We presented *Srcrepo*, a model-based framework for the creation and analysis of source code repository models as a model-based approach to mining software repositories. We started with support for a single type of version control system and one programming language. Of course, this does not prove a possible abstraction from different programming languages and version control systems yet, but we could show that reverse engineering, which promises the necessary abstraction, can be practically applied to all revision in large-scale software repositories.

Our experiments have shown that we can analyze a large set of source code repositories in a *reasonable* time frame. We can persist repository models to reuse them and reduce execution time in sequential analyses. Furthermore, analyses of independent software projects can be run in parallel. In summary, we can be confident that model-based MSR can be run on large-scale software repositories. Independent of MSR and from a scalable modeling technology perspective, we could show that EMF can practically be scaled to models of several billion model elements. The presented experiments only showed the principle scalability of model-based technology for the use in MSR. In the future, frameworks like MoDisco have

to adopt support for multiple-languages to actually facilitate the use of abstractions and allow us to harness the heterogeneity of large scale software repositories.

REFERENCES

Altmanninger, K., Seidl, M., and Wimmer, M. (2009). A survey on model versioning approaches. *IJWIS*, 5(3):271–304.

Bajracharya, S., Ossher, J., and Lepos, C. (2009). Sourcerer: An internet-scale software repository. In *Proceedings of SUITE'09, an ICSE'09 Workshop*, Vancouver, Canada.

Barmpis, K. and Kolovos, D. (2013). Hawk: Towards a scalable model indexing architecture. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pages 6:1–6:9, New York, NY, USA. ACM.

Basili, V. R., Briand, L. C., and Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761.

Bruneliere, H., Cabot, J., Jouault, F., and Madiot, F. (2010). 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, pages 173–174. ACM.

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493.

Chikofsky, E. J., Cross, J. H., et al. (1990). Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, 7(1):13–17.

Di Rocco, J., Di Ruscio, D., Iovino, L., and Pierantonio, A. (2014). Mining metrics for understanding metamodel characteristics. In *Proceedings of the 6th International Workshop on Modeling in Software Engineering (MiSE)*, pages 55–60.

Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2015). Boa: Ultra-large-scale software repository and source-code mining. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1):7.

Falleri, J.-R., Teyton, C., Foucault, M., Palyart, M., Morandat, F., and Blanc, X. (2013). The harmony platform. *CoRR*, abs/1309.0456.

Gousios, G. and Spinellis, D. (2009). A platform for software engineering research. In Godfrey, M. W. and Whitehead, J., editors, *MSR*, pages 31–40. IEEE.

Gyimothy, T., Ferenc, R., and Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Softw. Eng.*, 31(10):897–910.

Kagdi, H., Collard, M. L., and Maletic, J. I. (2007). A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131.

Kolovos, D. S., Di Ruscio, D., Pierantonio, A., and Paige, R. F. (2009). Different models for model matching: An analysis of approaches to support model differencing. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, pages 1–6. IEEE Computer Society.

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

Milev, R., Muegge, S., and Weiss, M. (2009). Design Evolution of an Open Source Project Using an Improved Modularity Metric. *Open Source Ecosystems: Diverse Communities Interacting*, 299:20–33.

Scheidgen, M. and Fischer, J. (2014). Model-based mining of source code repositories. In Amyot, D., Fonseca i Casas, P., and Mussbacher, G., editors, *System Analysis and Modeling: Models and Reusability*, volume 8769 of *Lecture Notes in Computer Science*, pages 239–254. Springer International Publishing.

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

Scheidgen, M., Zubow, A., Fischer, J., and Kolbe, T. H. (2012). Automated and transparent model fragmentation for persisting large models. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, volume 7590 of *LNCS*, pages 102–118, Innsbruck, Austria. Springer.

Subramanyam, R. and Krishnan, M. S. (2003). Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Softw. Eng.*, 29(4):297–310.

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

Williams, J., Matragkas, N., Kolovos, D., Korkontzelos, I., Ananiadou, S., and Paige, R. (2014a). Software analytics for MDE communities. *CEUR Workshop Proceedings*, 1290:53–63.

Williams, J. R., Ruscio, D. D., Matragkas, N., Rocco, J. D., and Kolovos, D. S. (2014b). Models of OSS project meta-information: a dataset of three forges. *Proceedings of the 11th Working Conference on Mining Software Repositories*, undefined(undefined):408–411.

Wilson, S. and Kesselman, J. (2000). *Java Platform Performance: Strategies and Tactics*. Addison-Wesley, Boston, MA.

Yu, P., Systä, T., and Müller, H. A. (2002). Predicting fault-proneness using OO metrics: An industrial case study. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering*, CSMR '02, pages 99–107, Washington, DC, USA. IEEE Computer Society.