University of Zurich[UZH]

Zurich Open Repository and Archive
University of Zurich
University Library
Strickhofstrasse 39
CH-8057 Zurich
www.zora.uzh.ch

Year: 2015

Rapid Multi-Purpose, Multi-Commit Code Analysis

Alexandru, Carol V ; Gall, Harald C

DOI: https://doi.org/10.1109/ICSE.2015.211

Posted at the Zurich Open Repository and Archive, University of Zurich
ZORA URL: https://doi.org/10.5167/uzh-110155
Conference or Workshop Item
Accepted Version

Originally published at:
Alexandru, Carol V; Gall, Harald C (2015). Rapid Multi-Purpose, Multi-Commit Code Analysis. In: 37th International Conference on Software Engineering (ICSE), New Ideas and Emerging Results (NIER), Florence, Italy, 16 May 2015 - 24 May 2015, IEEE.
DOI: https://doi.org/10.1109/ICSE.2015.211

Rapid Multi-Purpose, Multi-Commit Code Analysis

Carol V. Alexandru and Harald C. Gall
s.e.a.l. - software evolution and architecture lab
Department of Informatics, University of Zurich, Switzerland
{alexandru,gall}@ifi.uzh.ch

***Abstract*—Existing code- and software evolution studies typically operate on the scale of a few revisions of a small number of projects, mostly because existing tools are unsuited for performing large-scale studies. We present a novel approach, which can be used to analyze an arbitrary number of revisions of a software project simultaneously and which can be adapted for the analysis of mixed-language projects. It lays the foundation for building high-performance code analyzers for a variety of scenarios. We show that for one particular scenario, namely code metric computation, our prototype outperforms existing tools by multiple orders of magnitude when analyzing thousands of revisions.**

I. Motivation

Software engineering researchers use static code analysis tools for a variety of purposes, such as learning more about how software evolves over time, identifying patterns and anti-patterns, and developing new means of assessing software quality. Most of these tools are built for one particular purpose and one particular programming language. Furthermore, most of them can only analyze a single revision at a time. Because of these design limitations, software evolution studies are currently limited to analyzing only a few major releases of a handful of projects at a time (e.g. [5], [7], [16]). It is infeasible to rapidly analyze every commit of a project or to analyze the source code of a large number of projects. This contrasts other research approaches in software engineering, such as Mining Software Repositories (MSR), where it is now common to analyze entire commit logs, mailing list communication and issue tracker data for many projects and then draw conclusions not only by direct observation, but also by applying explorative machine learning techniques (e.g. [6], [12]).

Existing code analysis tools are unsuited to operate at that scale and the most limiting factor is speed. While feature-rich code analysis tools are sufficiently fast for performing one-time analyses, analyzing thousands of commits contained in large software repositories one after the other would require weeks or even months. But speed is not the only problem. Modern projects are often written in multiple languages. A web application, for example, may contain Java, Javascript, XML, HTML and CSS code. All of this code is subject to Lehman's laws of software evolution [10] and it may all be relevant to software engineering research. Yet the fact that most tools are written for one programming language severely limits the ability to gain new knowledge by analyzing such mixed-code projects. Finally, the interactive nature of existing tools burdens researchers with performing manual tasks, such as downloading source code, configuring and running tools, as well as extracting analysis results by hand.

II. Goal and Potential

For the reasons stated above, we are convinced that we need a new method for analyzing source code. A method which is fast enough to operate at the scale of thousands of revisions and projects and which is flexible enough to be easily adapted for multiple programming languages and different purposes.

There are many possible uses for such a multi-purpose, high-performance code analysis tool. It could be used to compute code metrics over the entire lifetime of many projects, creating sufficient data for explorative data mining to uncover previously unknown patterns in software evolution. Another use case would be to provide highly detailed information on affected code artifacts in bug reporting, such as the nature of previous changes to a file (answering questions such as "Are there other classes that changed their interaction with this buggy class, and in which commits are these changes?" or "How has the list of arguments to this method changed over time?" or simply providing the developer with a condensed overview on the most important changes that occurred in a particular code artifact). Furthermore, a fast, automated analysis tool could be used to provide instant developer feedback, either continuously inside an IDE or during continuous integration, or it could be used to automatically analyze the impact of code contained in merge requests.

As a first step towards the goal of large-scale, automated software analysis, we have developed a graph-based analysis approach which is able to analyze the code of any number of revisions in a git repository simultaneously. Our prototype currently has the necessary capabilities to analyze Java source code, but adding support for additional languages is straightforward. In a pilot study, we compared our prototype implementation to existing tools for the purpose of computing code metrics. We use the AspectJ project throughout this paper to illustrate the approach.

III. Approach

In a nutshell, our approach views source code analysis as a graph computational problem and exploits the fact that the vast majority of source code in a repository does not change in most commits. Our Scala prototype, called LISA, uses a dedicated graph data structure to perform all analyses and does not require a language-specific compiler or preprocessor, but only a parser to translate the abstract syntax tree (AST) of

the source language into a graph. All subsequent analyses of the source code can be implemented in Scala and are executed in parallel over the entire history of the entire source code. LISA uses the JDK's own parser (which is contained in `javax.tools.JavaCompiler`) to parse Java code, although LISA could work with an ANTLR, or any other parser, too. JGit is used for interacting with git repositories and Signal/Collect [14] is used to perform graph computations. Signal/Collect implements a computational paradigm, where nodes in a directed (possibly cyclic) graph communicate with each other by sending and collecting signals, as well as modifying their own state. Using this simple mechanism, powerful, parallelized computations can be formulated. Since our approach is aimed at the very large scale, LISA is implemented as a service (currently exposing only a rudimentary Web-API), intended to be installed on powerful server hardware or a cloud virtual machine. To analyze a project, the user simply submits the Git URL of a repository to the service, optionally restricting the branches and the ranges of revisions to be analyzed. LISA will then automatically clone the repository, parse the source code and analyze the project.

This process can be roughly divided into parsing & graph generation, followed by analysis.

A. Parsing

We do not use a simplified source code model such as FAMIX [15] or UML and instead load the ASTs of all revisions one-to-one into a graph using just a parser and an optional *AST node name mapping*. Parsers are readily available for all programming languages and can be integrated into LISA by simple glue code. A parser only needs to produce the following pieces of information for every node in the AST: *a*) A unique identifier (such as the source file path followed by the path of the node in the AST), *b*) the AST node type (as defined by the parser), *c*) the parent AST id and, *d*) any optional literals (for example the literal number of an *int* node). Note that the edges between nodes always point from children to their parents (upward). Of course, it is unlikely that all AST nodes are required in an analysis, but it is worthwhile to trade the extra memory required to accommodate superfluous AST nodes for the ability to easily implement new parsers, without having to implement any sort of source code transformation.

The *AST node name mappings* can be used to implement analyses which are compatible with multiple languages. For example, a class method count can be formulated the same way for Java, Python and C++. In this case, the parser-specific names for *method* nodes can be mapped to a single generic identifier used when formulating the method count analysis. These mappings are lighter and more flexible than a rigid ontology or model definition and can be added ad hoc by the researcher.

B. Graph Generation

LISA uses JGit to checkout the first selected revision, and parses it entirely. Subsequent revisions are loaded incrementally by only parsing files which are new or changed. Every uniquely identified AST node exists only once in the graph, even if it appears in multiple revisions. However, all nodes keep track of which ranges of revisions of the actual source code they appear in, so no information is lost.

Figure 1 serves as an illustration and contains AST representations of four subsequent revisions A, B, C and D of a very simple exemplary program. There are various changes between the revisions, but some AST nodes are unchanged across one or more revisions. Instead of calculating and storing data redundantly for each of these revisions, they are collapsed into a single range defined by a *start* and *end* revision, as illustrated in Tree (5), where nodes are represented as existing in one or more contiguous ranges. In real-life examples, a single range may encompass thousands of revisions.

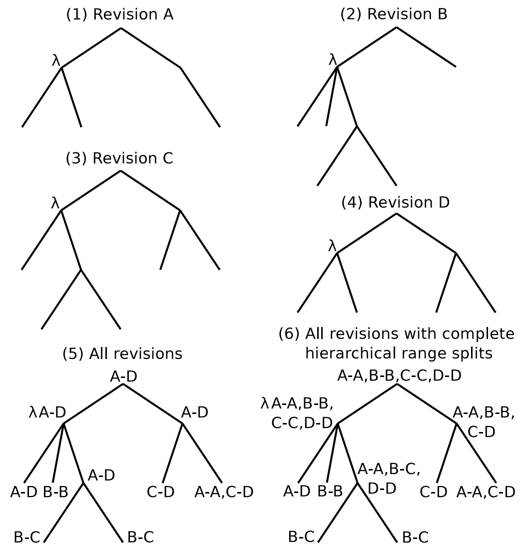

Fig. 1: Four revisions and their merged representations.

It may be necessary to further split revision ranges of a node if the data produced by an analysis for this node depends not only on its local, unchanging state, but also on the state of other, changing nodes, such as descendants. Assume for example that we are computing a method count, and that the AST node marked with λ is a *class* node, whereas its children are *method* nodes. In that case, the method count of the (never changing) *class* node actually varies across different revisions. Tree (6) illustrates a possible split arrangement, where revision ranges for parents are split such that they can accommodate all possible permutations of their descendant's revision ranges. Note however, that this is only an example and that LISA splits ranges only when necessary and also deduplicates adjacent ranges if their states become equal. In other words: even if an unchanging AST node has changing descendants, if its data remains the same across a revision range, no split occurs.

This approach represents tremendous space and complexity savings. For example, a recent AspectJ revision can be represented using $\sim$2.2 million nodes, yet *all* 7686 revisions can be represented using a mere $\sim$6.5 million nodes. This is possible because the vast majority of a program does not

change across revisions. It also reduces the computational complexity, because computations are performed only once for any contiguous revision range of any subgraph.

C. Analysis

Once all revisions have been parsed, analyses can be run on the graph. Analyses are formulated by writing small portions of Scala code to define the *signal* and *collect* behavior of different AST node types as well as data containers to store results for each revision range of a node. For example, to formulate a method count for classes, we define a data container on *class* nodes that stores a single integer, we define *signal* on *method* nodes so that it emits the number 1, and we define *collect* on *class* nodes so that it will add incoming method count numbers to its local data container. Analyses may freely create additional nodes and edges in the graph (for example to create control flow graphs or to connect ASTs of different files for analyzing dependencies) and also restrict the kinds of signals that travel along different edges. In this fashion, arbitrarily complex analyses can be formulated.

Any number of analyses can be defined and they all run in parallel. The only exception are analyses which depend on data created by other analyses. These are scheduled so that they run one after the other. Since analyses are formulated using isolated pieces of functionality, a very high degree of parallelism can be achieved. As mentioned previously, analyses can be formulated for multiple languages by the use of *AST node type name mappings*. The computation finishes, when there are no more signals being transmitted through the graph. At that point, the data can be selectively persisted from the nodes (where each node may contain separate data for each revision range) to a suitable database.

IV. Pilot Study

To evaluate our approach, we formulated analyses for a number of code metrics (such as the number of packages, classes, attributes and methods, cyclomatic complexity and maximum control flow nesting) as well as the *Brain Method* code smell [9]. We then used our prototype as well as two existing tools to analyze the AspectJ project. A recent revision of AspectJ contains 448 808 lines of code (not counting comments and blank lines) in 6 331 files and the AspectJ git repository contains 7 686 commits from close to 14 years of development. We first used SOFAS [4], a service-oriented software analysis framework, and inFusion [8], a commercial stand-alone tool, to analyze a recent revision of AspectJ. We then used LISA to analyze that same revision, as well as the most recent 10 revisions, the most recent 100 and the most recent 1000-7000 revisions (in increments of 1000) and finally all revisions. Table I shows the run time for each tool when analyzing one revision, plus the average run time per revision for LISA when analyzing multiple revisions. Figure 2 shows that LISA's implementation appears to scale linearly with a growing number of revisions. When analyzing an increasing number of revisions, LISA's average run time per revision drops rapidly and converges at about 0.66 seconds per revision.

TABLE I: Execution times of different tools when analyzing AspectJ. Where LISA has analyzed more than one revision, the average time per revision is included.

Tool	#Revs.	**Total** (min)	(Avg.) (s)	**Parsing** (min)	(Avg.) (s)	**Analysis** (min)	(Avg.) (s)
SOFAS	1	14:50	—	1:45	—	13:05	—
inFusion	1	6:34	—	2:10	—	4:24	—
LISA	1	1:31	—	0:31	—	1:00	—
LISA	10	1:43	10.300	0:40	4.000	1:03	6.300
LISA	100	2:49	1.690	1:39	0.990	1:10	0.700
LISA	1000	13:22	0.802	11:10	0.670	2:12	0.132
LISA	2000	23:20	0.701	20:39	0.620	2:41	0.081
LISA	3000	33:03	0.661	30:12	0.604	2:51	0.057
LISA	4000	44:37	0.670	40:54	0.614	3:43	0.056
LISA	5000	54:19	0.654	48:15	0.579	6:14	0.075
LISA	6000	66:16	0.663	57:27	0.575	8:49	0.088
LISA	7000	71:37	0.614	62:08	0.533	9:29	0.081
LISA	7642	85:21	0.670	74:08	0.582	11:13	0.088

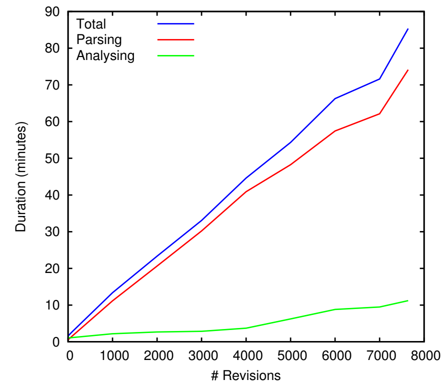

Fig. 2: Total, parsing and analysis durations for LISA when analyzing multiple revisions of AspectJ.

However, roughly 90% of this time is spent on parsing. This is because revisions are sequentially checked out and parsed in a working directory, incurring both expensive read and write operations to check out the code and then another read operation to parse the files. To analyze all revisions, LISA had a peak memory requirement of 20 gigabytes.

Note that an entirely fair comparison between these tools is not possible, since SOFAS and inFusion both perform more complex analyses (such as code clone detection and detecting other kinds of code smells) in addition to the analyses performed by LISA. However, we can expect that even if many complex analyses are run using LISA, its performance should at least match that of the other tools for analyzing *a single* revision, and still vastly exceed it when analyzing multiple revisions, especially if we consider how little time the actual analyses require compared to the time spent parsing.

V. Limitations and Future Work

LISA indiscriminately parses and analyzes all source code files in a repository for which it has a supporting parser. However, contrary to existing tools that partially compile the source code before analyzing it, LISA currently does not have knowledge of the global structure of a software project. For example, we may need to implement heuristics

to tell apart production from testing code, which is usually stored together in a repository. Furthermore, it is somewhat difficult to determine relationships between different parts of the source code. In other tools, method calls and attribute access are resolved by the compiler. As of today, we have only formulated an analysis that can resolve the types of Java variables and link them to their corresponding classes in the graph. Analyses for method call resolution, attribute access and inheritance tree linking need to be formulated to perform code coupling analyses. This may also need to be partially re-done for each additional programming language. There is currently no support for detecting code clones or code which has been moved to a different part of the AST across revisions, so for example renaming a class or folder causes all descendant nodes to be considered changed.

While LISA currently runs on a single server instance, the necessary scaffolding for a distributed execution among several virtual machines is already in place. This will be necessary for effectively analyzing many projects in parallel, because of LISA's memory requirements. To analyze multiple projects, LISA will automatically start and stop virtual machines to run analyses as required. We are also working on multiple ways of speeding up the parsing step.

We will add support for new languages and develop a web front-end for easy submission and retrieval of analysis results. We expect, that the tool will be publicly accessible and that researchers can freely use it to analyze projects. We would also like to create a data warehouse with analysis results on as many projects as possible.

VI. Related Work

There is fairly little existing research on speeding up static source code analysis and on large scale software analysis.

Dyer et al. have developed Boa [2], a code repository mining tool which translates queries formulated in a domain specific language into parallelized code that runs on a Hadoop cluster. It can be used to mine repository metadata as well as source code across the full history of 100 000s of repositories. By formulating queries using a classical visitor pattern, Boa can be used to query for the existence of particular code fragments (for example to count the number of assert statements or to query for specific class names), but not to perform more elaborate investigations, such as complexity calculations, code clone and code smell detection or other structural analyses.

Shang et al. have ported two code analysis tools, J-REX and CC-Finder as well as one log analysis tool, JACK, to run on MapReduce [13]. Moving from a single machine to a cluster of 10 machines decreased run times by a factor of 5 to 10.

Open HUB (previously *Ohloh*) is a web service offering analytics and code search for numerous open source projects [1]. It offers data on programming languages used, on contributors, and it offers some basic code analyses, such as the relative number of lines of code and comments.

Gerlec et al. describe a framework for language-independent software analysis [3]. They use ANTLR to parse code of different languages into an enriched Concrete Syntax Tree (eCST) and store it as XML, which is re-read to calculate basic code metrics. The eCST is comparable to a FAMIX model, but it is more fine-grained. Our approach differs from theirs in that we use the ASTs as provided by the parser directly, without any enrichment or XML representation. Their approach only works on a single revision.

VII. Conclusion

We have presented a new approach for rapidly performing complex source code analyses over the entire history of a software project, showing that our prototype is able to compute basic code metrics as well as detect one particular code smell for thousands of revisions within a short time. With further development, we will be able to use this approach to create vast code metric databases, to conduct full-history code quality evolution studies, to build tools for analyzing multiple programming languages and to enable other kinds of previously infeasible research.

References

[1] *Open HUB*, Black Duck Software, Inc. http://www.openhub.com

[2] Dyer, R., Nguyen, H., Rajan, H. and Nguyen, T. N.: *Declarative Visitors to Ease Fine-grained Source Code Mining with Full History on Billions of AST Nodes*, Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences GPCE'13 (2013)

[3] Gerlec, Č., Rakić, G., Budimac, Z. and Heričko, M.: *A Programming Language Independent Framework for Metrics-based Software Evolution and Analysis*, Computer Science and Information Systems, Vol. 9, No. 3, p1155-1186 (2012)

[4] Ghezzi, G. and Gall, H. C.: *SOFAS: A Lightweight Architecture for Software Analysis as a Service*, Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture WICSA '11 (2011)

[5] Godfrey, M. W., and Tu, Q.: *Evolution in Open Source Software: A Case Study*, Proceedings of the International Conference on Software Maintenance ICSM '00 (2000)

[6] Guzman, E., Azócar, D., Li, Y.: *Sentiment Analysis of Commit Comments in GitHub: An Empirical Study*, Proceedings of the 11th Working Conference on Mining Software Repositories MSR '14 (2014)

[7] Herraiz, I., Robles, G., Gonzalez-Barahona, J. M., Capiluppi, A. and Ramil, J. F.: *Comparison between SLOCs and number of files as size metrics for software evolution analysis*, Proceedings of the 10th European Conference on Software Maintenance and Reengineering (2006)

[8] *inFusion*, Intooitus s.r.l. http://www.intooitus.com/products/infusion

[9] Lanza, M., Marinescu, R. and Ducasse, S.: *Object-Oriented Metrics in Practice*, Springer-Verlag New York, Inc. (2005)

[10] Lehman, M.M.: *Programs, life cycles, and laws of software evolution*, Proceedings of the IEEE, Vol. 68, Issue 9 (1980)

[11] Linares-Vasquez, M., Hossen, K., Dang, H., Kagdi, H., Gethers, M. and Poshyvanyk, D.: *Triaging incoming change requests: Bug or commit history, or code authorship?*, Proceedings of the 28th IEEE International Conference on Software Maintenance ICSM '12 (2012)

[12] Meng, X., Miller, B. P., Williams, W. R. and Bernat, A. R.: *Mining Software Repositories for Accurate Authorship*, Proceedings of the 29th IEEE International Conference on Software Maintenance ICSM '13 (2013)

[13] Shang, W., Adams, B. and Hassan, A. E.: *An Experience Report on Scaling Tools for Mining Software Repositories Using MapReduce*, Proceedings of the IEEE/ACM International Conference on Automated Software Engineering ASE '10 (2010)

[14] Stutz, P., Bernstein A., Cohen, W.: *Signal/Collect: Graph Algorithms for the (Semantic) Web*, Proceedings of the 9th International Semantic Web Conference on The Semantic Web ISWC'10 (2010)

[15] Tichelaar, S., Ducasse, S. and Demeyer, S.: *FAMIX and XMI*, Proceedings of the 7th Working Conference on Reverse Engineering WCRE '00 (2000)

[16] Xie, G., Chen, J. and Neamtiu, I.: *Towards a better understanding of software evolution: An empirical study on open source software*, IEEE International Conference on Software Maintenance ICSM '09 (2009)