# Using Version Control History to Follow the Changes of Source Code Elements

Zoltán Tóth, Gábor Novák, and Rudolf Ferenc
*University of Szeged, Department of Software Engineering*
*Szeged, Hungary*
{*zizo,novakg,ferenc*}*@inf.u-szeged.hu*

István Siket
*Research Group on Artificial Intelligence*
*MTA-SZTE, Szeged, Hungary*
*siket@inf.u-szeged.hu*

***Abstract*—Version control systems store the whole history of the source code. Since the source code of a system is organized into files and folders, the history tells us the concerned files and their changed lines only but software engineers are also interested in which source code elements (e.g. classes or methods) are affected by a change. Unfortunately, in most programming languages source code elements do not follow the file system hierarchy, which means that a file can contain more classes and methods and a class can be stored in more files, which makes it difficult to determine the changes of classes by using the changes of files. To solve this problem we developed an algorithm, which is able to follow the changes of the source code elements by using the changes of files and we successfully applied it on the WebKit open source system.**

***Keywords*-Version Control System, Repository Mining, Static Analysis**

## I. Introduction

Version control systems (VCS) are used during the development of most noticeable projects. Although these tools have different features the main idea behind them is the same, namely, all of them are able to follow and store the changes in the software from the beginning of its development. Since the source code of the software is organized into files and folders, the VCS can store the changes of files only. On the other hand, software systems are built upon source code elements, like classes and methods, but these elements very rarely coincide with files. This means that although VCS stores the whole history of files found in the system, it cannot support the developers sufficiently in understanding the evolution of the software.

This information would be very useful in many areas. For example, if we want to create a bug prediction model for a software system based on its history, we need to know which classes or methods changed and which of them were faulty in the past [1], [2]. As an other example, the knowledge about the changed classes and methods can be used to support testing because only those test cases have to be re-run that test the classes and methods that are affected by the change.

Different approaches have been proposed to identify the changes. In case of Java, a file can contain up to one public class and an arbitrary number of non-public classes which are usually strongly connected to the public class, so we can accept the approach where the changes of a file are assigned to the public class. However, this solution cannot be applied for methods and cannot be generalized to other languages (e.g. for C++ or C#) where a file can contain more classes and a class can be split into parts and these parts are organized into different files. If we want to see the changes between two major versions at source code level, one solution can be that on all revisions between the two versions static source code analysis is performed one by one to find the source code positions (path, begin and end line) of the elements (e.g. classes, methods), and by using VCS information the consecutive versions can be compared to see the changes of the elements. Because of the many static analyses this solution is very time and resource consuming so it is difficult to apply it in practice and it does not work if the source code is inconsistent (for example, if the implementation of a class is not finished and the system cannot be built).

In this paper we present an algorithm, which needs only a few versions to be analyzed statically, and starting from an analyzed revision, where the source positions of the elements are precisely known, following the file level changes stored in the VCS, the positions of the elements can be traced in the unanalyzed revisions with good precision.

In Section III we present the technical background of our solution for following the source code positions between two analyzed versions of a system. The result of a preliminary experiment on the *WebKit* open source browser engine is presented in Section IV. In Section V the possible application area and the future work will be discussed. But first, in Section II we overview the related works.

## II. Related Work

Different approaches have been proposed to track the changes of source code elements regarding to elapsed time [1], [3], [2], and then use this information e.g. for bug prediction, finding the possible code fault positions, or calculating process metrics of each source code element. Although the aim of the approaches is the same, there are fundamental differences among the data mining methods.

Hattori *et al.* [4] created an impact analysis tool, called IMPALA, which was able to identify possible impacted entities by using a proposed change set of the software. They presented a method which processes every commit for the source code information. This method (*Change Extractor*) converts every revision of a class into an abstract syntax tree (AST), compares every two subsequent revisions and stores every structural change that occurred from one revision to another. The method is able to find e.g. the change in the signature of a method, the creation of a new class, the removal of an old one, or the elimination of an inheritance relationship. Our approach is different, we do not create AST for each revision, we only analyze the information from the VSC system (e.g. diff, info command results).

Hassan and Holt [2] presented an approach (*The Top Ten List*) which highlights the ten most suspicious subsystems (directories) which have faults. They used the approach presented by Graves *et al.* [5], which describes a way to get information from VCS for the process metrics of the source code elements. They followed all changes in every source code element from the version control system. They focused on the differences between revisions (added, deleted, and modified lines). Hassan and Holt validated their work on six large open source projects (NetBSD, FreeBSD, OpenBSD, KDE, KOffice, Postgres). They calculated the process metrics on the level of files, while we follow the changes of source code elements (e.g. classes or methods).

Sliwerski *et al.* [1] worked on extracting data from version control system and connecting it with a source code elements (and bugs). They connected the information on file level, but not all lines in a commit. They analyzed every important (bug fix) commit's every file change, and defined the changed lines in a set (every added and deleted line was treated as changed). Then they created the intersection of the subsequent revisions up to a bug fix to find out which part of the code caused the bug. They tested their approach on the source code of Mozilla (on 392,972 revisions) and Eclipse (on 278,010 revisions). The authors continued this work in their tool called HATARI [6]. The main difference can be found in the processing of the change sets, as they added each changed line (added, deleted, or modified) to the set from every bug fix from the earlier revisions.

## III. Extracting Information from VCS

Although the VCS stores all changes of the source code it is not easy to mine information from it at source code elements level (e.g. classes or methods). The main difficulty is that the source code of the system is organized into files and folders and the VCS follows only their textual changes, it is not aware of the program structure. Unfortunately, the physical representation of the file system rarely coincides with the logical structure of the system, therefore the changes of the source code elements cannot be extracted directly from the version control history. This means that if we are interested in the changes of classes or methods, first their source code positions have to be located for each examined revision of the system, and only then the changes at source code elements level can be calculated by combining the changes in the files and the known source code positions. On the other hand, we have to carry out static source code analysis for each commit to determine the positions of the elements, which is a very time and resource consuming task and in extreme cases when the source code is incomplete it might be even impossible. This solution can be improved by applying incremental analysis, which means that only the changed parts of the system are analyzed and for the rest the information of the previous analyzed revision can be used. Although this solution can be applied for small or medium sized systems, where there are not too much commits in the examined period, it cannot be applied for systems which are large, complex and have too many commits, therefore it is too expensive to analyze every revision even incrementally. Besides, the programming language also influences the cost because, for example, Java systems can be usually analyzed fast incrementally, while analyzing C/C++ systems is much more expensive.

Different approaches have been proposed to overcome this problem. For example, in Java only one public class can be in a file, so associating the change of a file with its public class is a good approximation but it cannot be generalized for methods or other languages. The other solution can be that the system is analyzed after each *n*th commits and when a given revision of the system is examined the positions of the elements are approximated with the information of the closest analyzed version [7]. This solution is general, but it is not precise and it still requires lots of resources.

### A. Following the changes

To reduce the resource demand of the process we present a general algorithm which needs language dependent static analysis only for certain revisions of the software. Static analysis is needed only to collect the source code positions of the source code elements. In order to know the position information of the source code elements in the unanalyzed revisions we process the changes between two analyzed consecutive revisions and map the changes to source code elements.

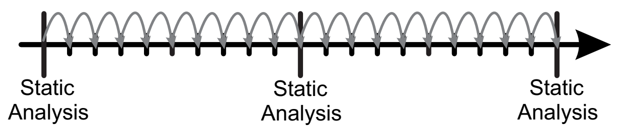

Figure 1. Maintaining source code positions between analyzed revisions

Figure 1 shows how the algorithm maintains the source code positions between two analyzed revisions. It starts from the first analyzed revision where the source code positions of all elements are known. First, the diff between the actual and the following revision is asked from the VCS. The diff contains all changes between the two revisions; more precisely, it tells us which lines were added, deleted, or modified in the affected files and which files were added, deleted or moved. Although the developer carried out the modification in one step the diff can be decomposed into *elementary change blocks* and it is enough to examine the effect of these blocks individually and to sum up the result of the examination to see the effect of the modification instead of processing the whole diff at once. It is evident that changes in different files can be processed separately and those changes that affect different parts of a file can also be processed one by one. On the other hand, an elementary change block must contain all neighboring lines that the change affected, otherwise its effect to the source code elements is handled incorrectly. Figure 2 shows an example where two elementary change blocks can be identified.

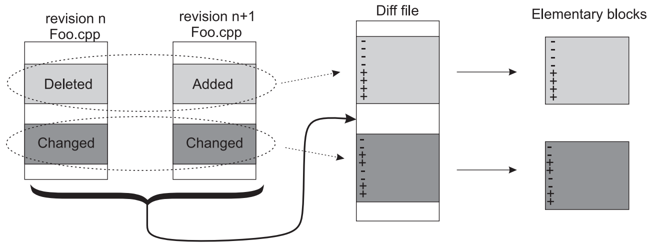

Figure 2. Decomposing diff into elementary change blocks

Since the source positions of the elements are known in the actual version, the algorithm can determine which source code elements (e.g. classes, methods) are affected by the given elementary block and it can calculate and update the positions of the elements in the unambiguous cases (see Subsection III-B) and do nothing when the changes are unclear (see Subsection III-C). This updating process is repeated for each revision between those revisions on which static analysis was performed.

In the following we will demonstrate the cases when the effect of the elementary change blocks can be identified clearly and when they are unclear. Since many different variations can occur in both cases, only a short summary will be presented here.

## B. *Elementary change blocks in unambiguous cases*

There are many cases when the new positions of the elements can be determined unambiguously. Figure 3 shows an example where there is a class A in the examined file Foo.cpp and the lines deleted from the file were located inside class A. In this case the new position of class A can be calculated easily because its beginning did not change and its new end line can be calculated by subtracting the number of deleted lines from its original end line.

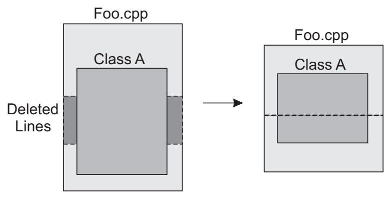

Figure 3. Unambiguous change in the source file

In general we can say that elementary change blocks that affect the inner part of a source code element (inserting lines to it, deleting lines from or changing its lines) can be processed easily and the new position information can be determined precisely. Similarly, if an elementary change block does not overlap the element it is also easy to calculate its effect, namely, if the block is located before the element in the file the element has to be pushed by the "measure" of the block[1] but if the change is made after the element its position remains the same.

## C. *Elementary change blocks in ambiguous cases*

Although the changes are stored precisely at file level, these changes cannot be determined at source code element level in all cases, which makes our algorithm imprecise. Figure 4 shows an example where the file Foo.cpp contains class A and in this case the elementary change block overlaps with the end of class A. In this case the effect of the diff on class A cannot be determined and to demonstrate the problem we presented three different possible results of the modification: (1) the size of the class decreased; (2) although the lines were changed the class remained more or less the same; or (3) the class was extended so its size increased. Unfortunately, by examining the changed lines only, it is impossible to decide how the end line of class A gets modified because the new line can be anywhere in the changed section (see Figure 4).

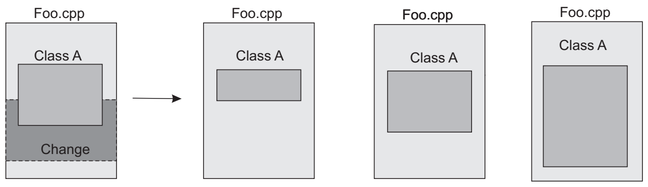

Figure 4. Ambiguous change in the source file

In general, elementary change blocks that overlap the beginning or the end of source code elements make it impossible to determine the change of the elements which means that in these cases our algorithm cannot follow the changes at source code level. A solution can be that we use some heuristics to estimate the new position, for example, using the results of the following static analyses. Or we can determine the new beginning or end line of the element by searching for the appropriate string in the diff or source file. But in our experiment presented in Section IV we did not apply any further heuristic because its results were very promising without it.

We have to mention that we highlighted only the most typical cases for both the unambiguous and the ambiguous cases, but there are many other unusual and tricky changes that are really hard or rather impossible to associate with source code elements even if all revisions are analyzed. For example, in C/C++ systems there are many hidden dependencies (e.g. include paths and macros) that influence a

[1]The "measure" of the block can be calculated by subtracting the number of lines deleted from the number of lines added. If a line changes only it does not change the positions of the other lines.

| | July | Aug | Sept | Oct | Nov | Dec | Jan | Feb | Marc | Apr | **Avg.** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Number of blocks | 21 167 | 19 531 | 23 820 | 48 935 | 40 859 | 23 400 | 26 174 | 32 440 | 33 543 | 29 509 | **29 937.8** |
| Ambiguous blocks | 614 | 547 | 715 | 777 | 702 | 645 | 725 | 1 125 | 1 148 | 585 | **758.3** |
| % of ambiguous blocks | 2.9% | 2.8% | 3.0% | 1.6% | 1.7% | 2.8% | 2.8% | 3.5% | 3.4% | 2.0% | **2.6%** |
| Unambiguous relevant blocks | 2 699 | 2 500 | 3 958 | 4 538 | 4 121 | 3 167 | 3 041 | 5 690 | 4 295 | 2 790 | **3 679.9** |
| % of ambiguous blocks | 18.5% | 18.0% | 15.3% | 14.6% | 14.6% | 16.9% | 19.3% | 16.5% | 21.1% | 17.3% | **17.2%** |

Table I
THE RESULTS OF THE EXPERIMENT ON WEBKIT

class so the class can be changed without modifying the class itself, it is enough to modify any of its hidden dependences.

## IV. PRELIMINARY RESULTS

In this section we present our experiment on the WebKit open source web browser (http://webkit.org/) which is implemented in C++. We examined 10 periods, each one month long, between July 2011 and May 2012; and we followed the changes of the classes and methods. We analyzed the first revision of each month and we applied the algorithm to see how it performs on the rest of the month. Table I shows the result where the first 10 columns represent the results for the examined months and the last column shows their average. *Number of blocks* tells us how many elementary change blocks were identified in the examined month and *Ambiguous blocks* represent the number of elementary blocks whose effect could not be processed properly. From the *% of ambiguous blocks* row showing how many percent of the elementary blocks are ambiguous we can see that about 2.6% of the changes are problematic while all other changes could be processed properly.

However, these results can be misleading because we examined all changes carried out in the repository but a notable part of the source code[2] is not taken into account during the analysis so the changes of these parts are always unambiguous. Therefore, we counted only those unambiguous changes that affected the examined elements (*Unambiguous relevant blocks*) and we calculated what percent of the ambiguous and unambiguous but relevant changes are ambiguous (*% of ambiguous blocks*). In this case the percent of unclear changes is between 15% and 22% which is still tolerable. Of course, the acceptability of these results heavily depends on their usage, but we have to take into consideration that we avoided thousands of static analyses.

## V. CONCLUSION AND FUTURE WORK

In this work we presented an algorithm which is able to extract the changes from version control systems at source code element level, i.e. to track the changes of the source code positions of e.g. classes or methods. The advantages of this approach are that it is language independent, it can be adopted to different VCSs and it can be applied for incomplete source code as well. Its weakness is that it is not absolutely precise but its precision can be improved arbitrarily by increasing the number of statically analyzed revisions. We successfully applied our algorithm on the WebKit open source system.

[2]The repository of WebKit contains all WebKit related sources, like test cases, files written in other languages; moreover, we analyzed the Qt port of WebKit on Linux, so the other parts are not taken into account.

This method can be the base of many further investigations and there is a wide range of areas where it can be applied, like improving effectiveness of testing, or calculating metrics which measure the software development process.

As future work, we plan to examine in detail the precision of the algorithm by analyzing more revisions of WebKit and by extending our investigation to other software systems as well. Besides, we will examine how we can improve the performance in the ambiguous cases. For example, if the beginning or the end of a class cannot be calculated precisely based on the diff, it could be determined by searching for an appropriate string representing its boundary in the diff or source file.

## ACKNOWLEDGEMENTS

This research was supported by the Hungarian national grant GOP-1.1.1-11-2011-0006.

## REFERENCES

[1] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, May 2005.

[2] A. E. Hassan and R. C. Holt, "The top ten list: Dynamic fault prediction," in *Proceedings of the 21st International Conference on Software Maintenance*, ser. ICSM 2005, Budapest, Hungary, 2005, pp. 263–272.

[3] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting faults from cached history," in *Proc. of the 29th international conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 489–498.

[4] L. Hattori, G. dos Santos Jr, F. Cardoso, and M. Sampaio, "Mining software repositories for software change impact analysis: a case study," in *Proc. of the 23rd Brazilian symposium on Databases*, ser. SBBD '08. Porto Alegre, Brazil, Brazil: Sociedade Brasileira de Computação, 2008, pp. 210–223.

[5] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, pp. 653–661, Jul. 2000.

[6] J. Sliwerski, T. Zimmermann, and A. Zeller, "Hatari: raising risk awareness," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 107–110, Sep. 2005.

[7] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 897–910, Oct. 2005.