Impact Analysis Using Static Execute After in WebKit

Judit Jász, Lajos Schrettner, Árpád Beszédes, Csaba Osztrogonác, Tibor Gyimóthy
Department of Software Engineering
University of Szeged
Szeged, Hungary
{jasy, schrettner, beszedes, oszi, gyimi}@inf.u-szeged.hu

***Abstract*—Insufficient propagation of changes causes the majority of regression errors in heavily evolving software systems. Impact analysis of a particular change can help identify those parts of the system that also need to be investigated and potentially propagate the change. A static code analysis technique called Static Execute After can be used to automatically infer such impact sets. The method is safe and comparable in precision to more detailed analyses. At the same time it is significantly more efficient, hence we could apply it to different large industrial systems, including the open source WebKit project. We overview the benefits of the method, its existing implementations, and present our experiences in adapting the method to such a complex project. Finally, using this particular analysis on the WebKit project, we verify whether applying the method we can actually predict the required change propagation and hence reduce regression errors. We report on the properties of the resulting impact sets computed for the change history, and their relationship to the actual fixes required. We looked at actual defects provided by the regression test suite along with their fixes taken from the version control repository, and compared these fixes to the predicted impact sets computed at the changes that caused the failing tests. The results show that the method is applicable for the analysis of the system, and that the impact sets can predict the required changes in a fair amount of cases, but that there are still open issues for the improvement of the method.**

***Keywords*-Change impact analysis, source code analysis, Static Execute After, regression testing.**

I. Introduction

Change propagation in heavily evolving systems is often incomplete [10]. This is mainly due to our inability to perfectly identify the parts of the system directly or indirectly affected by the change. Incomplete change propagation will then cause defects in the system, which can lead to failures. Many development processes try to minimize the amount of such failures in released software and rely on regression testing to identify them in advance [22].

Hence, any method that can aid change propagation can reduce the risk of regression. Specifically, the field of change impact analysis is heavily researched [8]. The task is to assess the expected impact of a change (e.g. in terms of software modules affected), and then this impact is used to perform selective regression testing [22] or guided code review [10]. However, everyday industrial application of sophisticated impact analysis methods is not typical, which can be attributed to a number of causes. First, automatic methods should be precise, complete and efficient at the same time, which is a challenge with state-of-the-art approaches. Second, there is little objective evidence of the efficiency of specific impact analysis methods in terms of their actual ability to prevent regression.

In this work, we concentrate on automatic impact analysis based on static code analysis. Recently, an algorithm called Static Execute After (SEA) has been introduced with promising results about its precision and efficiency [5], [18], [17]. The impact sets computed by the algorithm include the sets computed by the more precise dependence based algorithms, but at the same time – computed at procedure level – the sets are comparable in their sizes. Furthermore, the efficiency of the algorithm is much better, which enables its application to real, big software systems.

In this paper, our first aim is to overview the benefits of the method, its existing implementations and earlier experiences in applying it to experimental and also real size industrial systems. The algorithm has been implemented in different environments, and has been applied successfully to large systems like Mozilla and OpenOffice. Second, we wish to verify the applicability of the SEA algorithm on another real size, complex, and lively evolving software system, the open source WebKit web browser engine [26]. We had different goals with this research:

- Adapt the SEA method to this system and integrate it into the build process.
- Perform the analysis of a significant number of revisions and changes of WebKit.
- Make an analysis of the resulting impact sets in terms of their sizes.
- Process the regression test results for the selected revisions to find out how regression test results change over the selected evolution period.
- Finally, verify the hypothesis that applying this algorithm for impact analysis we can actually predict the required change propagation in a fair number of cases. For this, we looked at actual defects provided by the regression test suite along with their fixes taken from the version control repository, and compared these fixes to the predicted impact sets computed at the changes that caused the failing tests.

Our first, biggest experience is that the adaptation of the algorithm implementation to be able to regularly perform an analysis of the WebKit system required significant effort. We report on how it was possible to seamlessly integrate into the build, change and testing processes of the project. We managed to perform the analysis of a significant number of versions of the system, but the number of interesting revisions where we could simulate the effect of the impact analyis was more modest than we expected. However, we are still able to draw useful conclusions about when SEA was successful in identifying the required change, and the causes when it was not. The research in the present state is a good starting point to further improve the algorithm, and is a significant step towards its industrial application in this particular project, and similar projects. Most of the research around impact analyis deals with exploring the application areas for the method [8]. We believe that this work is one of the rare experiments where the actual prediction capability of an impact analysis method is verified.

The paper is organized as follows. In Section II, we provide some basic background information about impact analysis in general, and overview some related approaches. We introduce the basic method in Section III, and discuss some of its properties and earlier experiences. Section IV deals with existing implementations of the method by overviewing the basic related toolset. In Section V, we describe the application of the algorithm in WebKit, and the experimental study design, while the actual related results are presented in Section VI. Finally, we conclude in Section VII.

II. Related work on impact analysis

Impact analysis [8] deals with the problem of identifying those parts (the impact set) of a software system that might be affected by a change in the system, in other words finding the possible dependencies between the change and the other parts of the system. The motivation behind the analysis is that developers can concentrate their efforts to the impact set when they want to evaluate the effects of a change. Most often they are interested in which parts of a program they have to (re)test if they want to ensure that the modifications did not break existing behaviour, or which part to (re)examine if a change turns out to cause their program to misbehave.

A practical impact analysis method should be as accurate as possible, safe, and sufficiently fast at the same time. Unfortunately it seems that present technology does not allow us to meet these requirements. There are accurate methods that are inefficient for real-life systems; their size is simply too large to handle them within reasonable resource bounds. Another way of approaching the problem is to calculate the impact set with a method that sacrifices accuracy for speed. Such a method should be validated by checking whether its accuracy is still good enough for practical purposes, and also whether it is safe in a sense that it does not miss any dependency that could have been identified by other, more accurate methods.

In particular, it should be checked whether

1) the sizes of the impact sets produced by the method are reasonably small, and
2) despite the inaccuracy of the impact sets, they can serve as a basis for determining the effects of a change.

The problem of balancing between accuracy and safety can be tackled by, for instance, applying different weighting methods that limit the analysis according to some heuristic prioritization of the dependencies.

Program slicing [27] is suitable for determining the impact sets of modified program components. There are a lot of different approaches to compute the slices, but almost all general solutions are not effective for large programs, including the probably most important practical method based on dependence graphs [14], [16]. Here, first a program representation is built that captures all the different dependencies between program elements like statements, and then a reachability algorithm finds the dependencies starting from a particular point, which represents the initial change. These methods have limited applicability – despite the existence of some commercial tools – in an everyday software development activity, as part of an integrated development environment, for instance. Although there are reports on the usability of program slicing on large programs [1], to our knowledge there are no tools that are able to deliver the performance that would be needed to any of the present day big and complex software projects. The reason is twofold: first, the program representation of a program with millions of lines of code can be extremely large; second, in many cases it is not necessary to determine dependencies at statement level as is done with the general approach of dependence-based slicing.

Because of these problems, other related approaches have been proposed. Many of them determine the impact sets of the modified procedures of a program from the call graph [8], [23]. Dependencies among classes can be approximated by using cohesion metrics [9], [28], if one would like to carry out class-level impact analysis in an object oriented system. Although these methods are quite simple, they are not safe, and it is easy to show that they miss to identify a set of real dependencies [5].

In previous works, we introduced the *Static Execute After* (SEA) relation as an alternative to traditional software dependencies, i.e. static forward program slicing. We presented empirical evidence that the SEA relations can be a good approximation of program slices at the procedure level, while its computational complexity is better than that of program slicing [5], [18], [17]. Our approach was motivated by Apiwattanapong *et al.* [3], who introduced the notion of the *Execute After* relation and applied it in dynamic impact analysis. The basic idea of both the dynamic version and the SEA approach is that we treat a program component to be dependent on the other if it is possible that they will be executed one after the other (not necessarily consecutively) in some execution of the program. This definition implies that the SEA relation will be a superset of the dependencies computed by the dependency based program slicing as its dependencies all rely on the existence of execution flow between the components. In the next section we will elaborate in more detail about the SEA relation and the algorithms for computing them.

Our method and the above mentioned approaches concentrated only on the source code and its changes in a system. But in many cases we can gain important information from other software artifacts like software repositories, bug tracking systems, or natural language texts. This information can then be used to derive special kinds of impact sets, but this work is not concerned with this area [29], [15], [21].

III. Static Execute After

A. *Definition of the relation*

The Static Execute After relation is defined as follows. For program elements (procedures, classes, statements, etc) f and g, we say that $(f, g) \in \text{SEA}$ if and only if it is possible that any part of g is executed after any part of f in any one of the executions of the program. Based on Apiwattanapong *et al.* [3] and Beszédes *et al.* [4] we formulate the SEA relation as follows:

$$\text{SEA} = \text{CALL} \cup \text{SEQ} \cup \text{RET}$$

where

$$\begin{array}{l} (f,g) \in \text{CALL} \\ (g,f) \in \text{RET} \end{array} \quad \Longleftrightarrow \quad \left\{ \begin{array}{l} f \text{ (transitively) calls } g \\ \text{(or } g \text{ (trans.) returns into } f\text{)} \end{array} \right.$$

$$(f,g) \in \text{SEQ} \quad \Longleftrightarrow \quad \begin{array}{l} \exists h : f \text{ (transitively) returns into} \\ h, \text{ then } h \text{ (transitively) calls } g \end{array}$$

It can be more convenient to consider the reflexive closure of this relation, since every change in a procedure f can affect any part of f from an impact analysis point of view. This is in line with slicing methods, where the starting point (criterion) is contained in the resulting slice.

It can be easily seen that the SEA relation covers all possible cases when a procedure can be called after the other. From computation point of view, SEA actually means following all possible control flow [2] paths from a procedure to the rest of the system, so in this sense it is much simpler than computing control- and data-dependencies with slicing [16], which are subsets of control flow relations.

Our method for computing SEA relations is the following. We have to build a suitable program representation in order to determine the subsets (CALL, RET, SEQ) of the SEA relation. The traditional call graph representation [23] is not sufficient for this purpose because it tells us nothing about the order of the procedure calls within a procedure. The program representation called *Interprocedural Control Flow Graph (ICFG)* [20] contains additional information but it is too detailed for our purposes.

Hence we base our approach on a novel program representation called *Interprocedural Component Control Flow Graph (ICCFG)* [5], which contains sufficient extra information to extract the required relations, while being much smaller and simpler than other graphs including the System Dependence Graph [16] used by slicing. In the ICCFG graph each procedure is represented by a *Component Control Flow Graph – (CCFG)* that has an entry node and several component nodes. Each component in a CCFG represents a strongly connected subgraph of the control flow graph of the procedure, but we keep only those subgraphs that contain at least one call site. The components inside a CCFG are connected by control flow edges, while the collection of CCFG graphs are connected by call edges.

The SEA relations are then computed by traversing the ICCFG program representation and collecting the visited nodes. Depending on the application, we can compute the dependencies only for one starting node or for more of them in parallel. The space/time behavior of the computation of the SEA relation can be controlled by adjusting the number of independent call paths a particular algorithm pursues concurrently. We consume the least amount of memory by following a single path at any given moment, but of course this method takes the longest to execute in the case of multiple passes for different change sets. At the other extreme, we can try to pursue as many paths as possible (possibly all), but this may require a large amount of working memory. In this case, memory consumption is offset to some extent by gains in running time, because we can avoid reprocessing of already visited nodes. Finally, we can also combine the two approaches and compute the dependencies for a given set of starting nodes simultaneously.

B. *Properties of the SEA sets*

In an earlier work [18], we showed that the SEA relations can be a good approximation of the static slices. In this experiment we used a suite of C programs by Binkley and Harman [6], and calculated the precision of our relation compared to the results of the static slicing as the golden standard of static impact analysis. To this purpose we investigated the differences in the sizes of the respective dependency sets. The precision values we found are shown in Figure 1. It can be seen that the precision is very good, meaning that there is a comparably small amount of additional dependencies produced by the SEA method due to its conservative nature. Since SEA does not produce false negatives, we always get 100% recall.

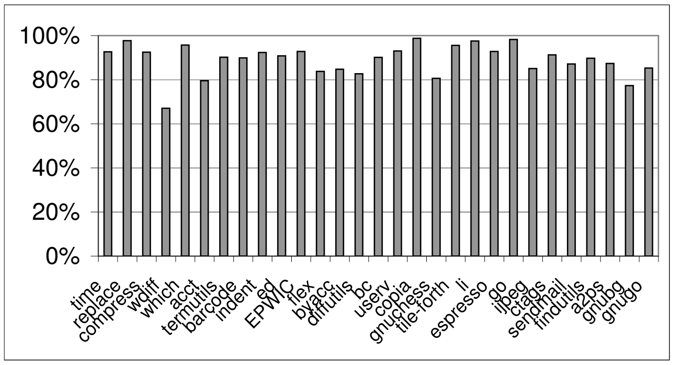


Figure 1. Precision of the SEA sets relative to program slices (recall is always 100%)

We also showed in another empirical experiment [19] that the precision of the SEA approximation is acceptable at procedure or higher level but not at statement level. The dependencies among the statements computed by the program slices are assimilated on higher level. Figure 2 shows the statement level precisions and procedure level precisions for the same subject programs.

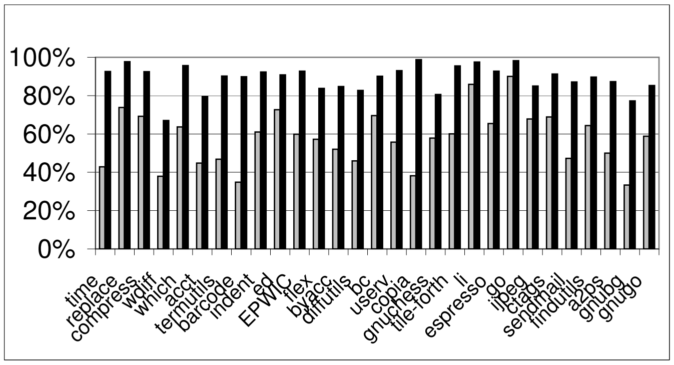


Figure 2. Precision of the SEA sets relative to program slices at statement and procedures levels (statement level precisions (light bars) and procedure level precisions (dark bars))

IV. SEA TOOLS FRAMEWORK

The original SEA algorithm has been implemented in different environments and for different purposes. For our first experiments, we used prototype implementations of the algorithm, but later our main aim became to be able to apply it in industrial settings. With the latest implementations we were able to analyze big systems like OpenOffice or Mozilla [18]. Computing program slices for these programs is almost impossible due to their program sizes, but the SEA computation gives us a possible way to determine impact relations in such big systems as well. The present paper is also about this topic but for a different system.

For the application in a real setting it is equally important to have a good collection of tools and a framework, which we will overview in the following.

A. The CodeRipple framework

In collaboration with FrontEndART Ltd. [13], a general impact analysis framework and toolset has been developed with the aim to support the application of different impact analysis methods in real application scenarios during development and maintenance. The CodeRipple tool [24] integrates a set of specific analyses, with the possibility to combine and parameterize them to best fit for a specific purpose. The flexible and rich configuration and programming interface allows its configuration for different needs, for example regression test selection, program comprehension, and also as an interactive tool and programming aid.

The framework applies the SEA method together with other impact analysis algorithms – both static and dynamic, and is able to determine dependencies between procedures in software with heterogenous architecture, including remote procedure calls, web services and calls to relational databases.

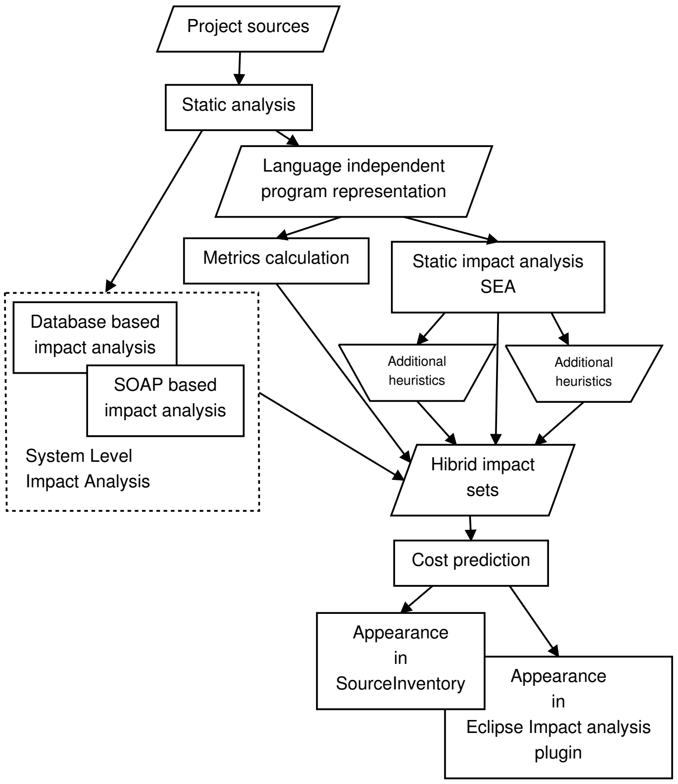

Project sources
Static analysis
Language independent program representation
Metrics calculation
Static impact analysis SEA
Database based impact analysis
SOAP based impact analysis
System Level Impact Analysis
Additional heuristics
Additional heuristics
Hibrid impact sets
Cost prediction
Appearance in SourceInventory
Appearance in Eclipse Impact analysis plugin

Figure 3. Overall architecture of the CodeRipple framework

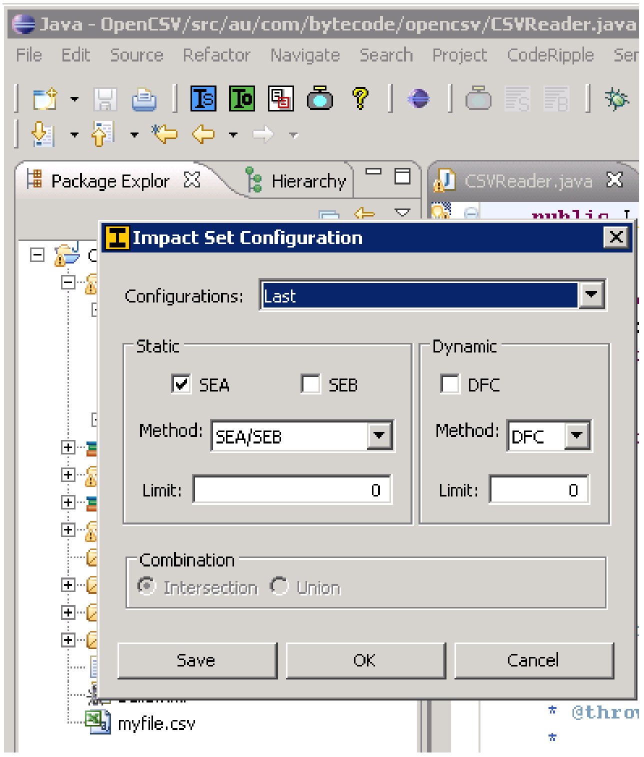
These extensions make it possible to analyze the dependencies not only in isolation for a specific technology but in an integrated manner. The underlying analyzers, including SEA, support the C/C++ and Java languages. CodeRipple is composed of a set of specific tools and components that are interconnected as shown in a high level overview in Figure 3.

The framework can be used in different usage scenarios. A set of configuration and automatic analysis scripts allows its integration in many different build environments and other software quality systems, while there is also a graphical user interface integrated into the development environment.

The Eclipse-based user interface in form of a plug-in provides several additional features like change cost estimation, test selection and test optimization. Using the plug-in users can configure and combine different impact analysis algorithms and their different settings. It is possible to display the analysis results directly on the source code, and some other useful features are included as well. This type of operation is similar to some other research impact analysis tools integrated into the development environment [10]. A screenshot with some typical windows of the graphical user interface of CodeRipple can be seen in Figure 4.

B. SEA computation tools and applications

SEA is a major component in the CodeRipple framework. In it, we use an advanced version based on the Columbus analysis toolset [11]. Columbus provides some essential analysis components like the parser front end, and the internal


Java - OpenCSV/src/au/com/bytecode/opencsv/CSVReader.java
File Edit Source Refactor Navigate Search Project CodeRipple
Package Explor
Hierarchy
CSVReader.java
myfile.csv

Impact Set Configuration

Configurations: Last

Static: [✓ SEA] SEB — Method: SEA/SEB — Limit: 0

Dynamic: DFC — Method: DFC — Limit: 0

Combination: [✓ Intersection] Union

Save OK Cancel

Figure 4. Screenshot of the CodeRipple tool

program representation infrastructure. Another major benefit of the toolset is that it is able to integrate into the program compilation process with the help of a *compiler wrapper* environment, and obtain those pieces of static information that are needed to create the graph representation of the program [12]. Without using the build process, the accurate collection of project information would have been extremely hard if not impossible. In the research of the present article we also used the wrapper technology to analyze the subject system.

There are other implementations of the algorithm, which were used in some other research projects. For our first experiments [5] we computed the SEA impact sets in parallel for all methods in a program. This algorithm traversed the graph that represented the program in one step and was able to define the impact set of every procedure at the same time (we call this a *global* algorithm). The benefit of the algorithm is that several impact sets could be obtained in one step, which was useful for experimenting with general sizes of the sets and similar. However, the downside was that the time and space costs of the algorithm were not acceptable for really large systems.

For a number of other applications (like impact analysis in this paper), it is enough to compute dependencies for a particular starting point, so we designed a specific algorithm with this in mind (a *demand driven* algorithm) [17].

The other property of our early implementation – when comparing the SEA sets with the program slices – was that we used the API of a slicing tool to get the representation of the needed graph. We found that CodeSurfer [14] was not only the best available program slicing tool, but it also offered the possibility to compute the information needed to create the graph for computing SEA. This seemed to be an appropriate solution since it was easy to compare the SEA sets with the results of slicing.

V. SEA IN WEBKIT

The SEA algorithm presented in the preceding sections had not been evaluated in detail on a real size and complex industrial system earlier. Earlier, we experimented with its application to big systems like Mozilla and OpenOffice, but in the present work we quantitatively and qualitatively investigate the properties of the algorithm on a different but equally complex software system, the WebKit open source web browser engine [26]. It is a layout engine that renders web pages in some of the leading web browsers and other applications [25].

In the current stage of the experimentation, we solved a number of technical issues to be able to analyze the system automatically in a regular manner. Furthermore, we started to investigate the actual prediction capability of the algorithm of the required change propagation, which we verified by computing impact sets of actual revision changes where regression errors have been introduced, and compared them to the actual changes where the regression errors have been fixed. In this section, we overview this experiment, while the next section deals with our findings.

A. Overview

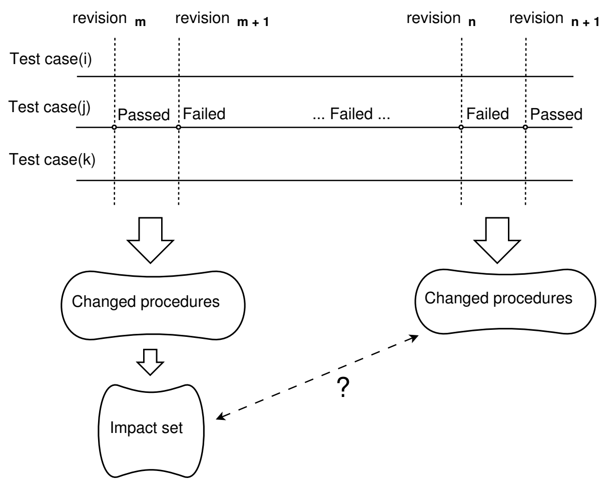


Figure 5. Change propagation experiment

Figure 5 serves as a guide to summarize our approach to evaluate the SEA method in the WebKit environment. Revisions of the system are represented from left-to-right as vertical lines. We can examine the differences in subsequent revisions to arrive at a set of procedures (we use this term for C functions and C++ methods in this paper) that were modified from one revision to the next. In the figure we depict these sets between two pairs of revisions, $revision_m$ and $revision_{m+1}$, then later between $revision_n$ and $revision_{n+1}$.

Next, the test cases of the system under examination are illustrated by long horizontal lines that belong to the common test suite, one line for each test case, which are all executed at each revision. All test cases are run on every revision to find out whether any regression errors have been introduced by the latest modifications. The outcome of running a test case can be either $Passed$ or $Failed$. Let us consider a scenario in which there is a test case tc_j that produces the following outcomes: $Passed$ in $revision_m$, then $Failed$ a number of times from $revision_{m+1}$ up until $revision_n$, then $Passed$ again in $revision_{n+1}$. In this scenario we can assume that some of the changes made between $revision_m$ and $revision_{m+1}$ are responsible for the failed test case tc_j. The error that was introduced in $revision_{m+1}$ is worked on, then it is corrected in $revision_{n+1}$, when test case tc_j passes again. Our hypothesis is that the impact set of the modified procedures at the time the error was introduced in $revision_{m+1}$ contains the procedures that were modified between $revision_n$ and $revision_{n+1}$. If that is true, then we would have an evidence that impact analysis in general and the SEA algorithm in particular is useful in predicting the required change propagation.

B. Analysis of WebKit

In our experimental study we analyzed WebKit across 781 revisions. WebKit contains about 1.8 million lines of C/C++ code and it has a relatively big collection of regression tests, which helps developers to keep code quality. The regression test suite consists of more than 20 thousand test cases. The aims of the regression tests are to maintain compatibility, standards compliance gains, and some stability, performance, security, portability, usability, hackability issues. Theoretically the layout regression tests must pass before patches can land in the repository. Unfortunately this requirement is not met in many cases. Due to the amount of regression tests and platforms, the developers often skip the full testing process before the commits. The above mentioned features make the WebKit system suitable for our experimental study.

We verify our hypothesis about the prediction capability of the algorithm by comparing the impacts of changes of a revision with a failed test and the changed procedures of the revision which corrects the test. The comparison is made on procedure level, meaning that the changes, impact sets and test information is analyzed on the granularity of functions and methods. The information about changes in the system are taken from the version control system, and about failing and passing test cases from the regression test suite logs.

The main steps of the experiment are shown in Figure 6. First we analyze the selected revisions of WebKit and build up the ICCFG graph representations of each version. At the same time we determine the modified procedures with the help of the Subversion version control system logs and the ICCFG graphs using path/line information. Since we need the information on procedure level we calculate the corresponding procedures from the path/line information.

Table I
AVERAGE EXECUTION TIMES OF THE ANALYSES

Build time:	20 min
Build time with static analysis:	1.5 - 2 hours
Regression tests running time:	16 - 20 min
SEA computation time:	1 - 10 min

In our measurements we used the Qt port of WebKit called QtWebKit on x86_64 Linux platform. For the compilation we used Qt 4.7.4, the latest stable version of Qt. In revisions where there were no source code changes or the changes affected other ports of WebKit, it was not necessary to compute the ICCFG, it was enough to run the regression tests to determine the states of the tests after applying the modification of the revision.

If one of the regressions tests of $revision_x$ failed, we searched for the first n, where the regression test passed in $revision_{x+n}$. Hence we obtained a pair of revisions: a failure inducing revision, and a fixing revision. If any member of a pair was a revision without changes in our port, we did not take that pair into account.

VI. RESULTS OF WEBKIT ANALYSIS

In this section, we present our results about the experiment with the WebKit system for impact set calculation. First, we present our experiences with applying the tools to the subject system. We will see that the analysis incurs some time overhead compared to the original build process. Then we provide some data about the WebKit revisions we analyzed, and the corresponding impact sets. Finally, we overview our findings about the comparison of the impact sets with the actual changes at the regression test pairs we investigated.

A. Performance of the measurement tools

The basis of our experiment is the analysis of the relevant revisions of the system, and the computation of the corresponding impact sets. This means those revisions that contain relevant changes at procedure level. In parallel to the build process, we compute the ICCFG graph, as we have seen in Figure 6. Additionally, we need to execute the regression tests as well. In Table I, we can see the times typically required for the mentioned steps for one revision (on an Intel Xeon X5670 (12 core 2.93 GHz) machine with 96 GB RAM).

As can be seen, the time required to compute the static information increases the build time, but it is still much better than any more accurate analysis method like computing program slices based on a dependence graph [1]. In fact, many other algorithms or tools would be probably unable to complete the analysis at all.

The time required for computing the SEA sets varies in a wide range, which is obviously due to the different number of changed procedures per revision. The whole analysis also includes the execution of additional tools like the extraction of the changed procedure names, but these additional costs are negligible.

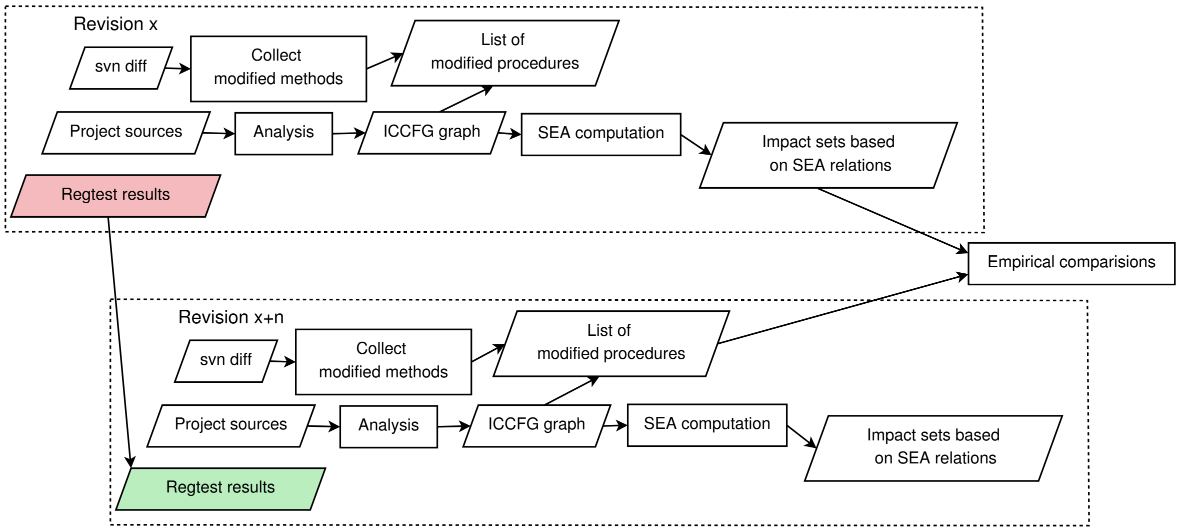


Figure 6. The steps of the WebKit experimental study

Table II
REVISIONS ANALYZED

Number of the investigated revisions:	781
Number of revisions without source code changes:	360
Number of revisions built and analyzed:	421
Number of revisions with changed procedures:	**196**

Table III
TEST PAIRS INVESTIGATED

Number of test pairs:	161
Number of test pairs without source code changes:	118
Number of test pairs without changed procedures:	31
Number of test pairs with changed procedures:	**12**

B. Revision and test information

We investigated revisions from 96590 to 97370, which corresponds to a development period of 9 days. Some relevant information about the revisions can be seen in Table II. 360 revisions contained no changes in the source code but some other files like configuration scripts, test inputs, new test cases, etc. The remaining revisions were all built and analyzed individually. Out of these versions, further 225 revisions could not be taken into account, because the source code changes affected a different port of WebKit. Since our graph representation about the system is built up during the compilation, we only analyzed and investigated the sources connected to the Qt platform. So we ended up with 196 useful revisions for further analysis.

The next step was to identify the relevant test pairs to our experiment out of the relevant revisions. As already discussed earlier, we were interested in pairs of revisions in which there was a change in the $Passed/Failed$ and $Failed/Passed$ status of one of the tests (see Figure 5). To compute these pairs we used our helper tool that processes the testing result logs of the regression test suite.

Unfortunately, we could not use all of the initial pairs in our experiments due to the following reasons. Table III shows some statistics about the revision pairs. Initially, we found an overall of 161 revision pairs based on test case statuses. We had to exclude those pairs in which there was no change in the source code at all (in either element of a pair), as well as those which we could not uniquely relate to C/C++ procedures of the examined Qt platform. The former issue was typically due to the changes of the layout tests, while the latter can be usually attributed to those changes in C/C++ files that involve global scope outside of any functions or methods or changes on a different platform than we used.

So, we could investigate 12 revision pairs in more detail. Since this number was smaller than we expected at the beginning of the experiment, we decided to follow a more qualitative than a quantitative evaluation as presented in the following. We have to note, though, that due to the so-called flakey tests this pairing may still include some false pairs. Flakey tests are those tests whose outcomes are dependent on the testing environment. For example in layout tests, they are most often caused by use of delays that become brittle when test conditions change.

C. Impact sets

We analyzed the sizes of the impact sets computed for the changed procedures for the relevant revisions. We wanted to find out how much the distribution of the set sizes resembles our earlier findings [18].

The WebKit systems consists of about 71000 procedures on average (this number varies from revision to revision but stays around this number for the investigated revisions). On average, in the investigated revisions 10 procedures have been modified. This is a relatively low number but it can be justified with the fact that the development process in this system involves frequent commits. A detailed distribution of the changed set sizes can be seen in Figure 7.

The sizes of the impact sets vary over a wide range. There

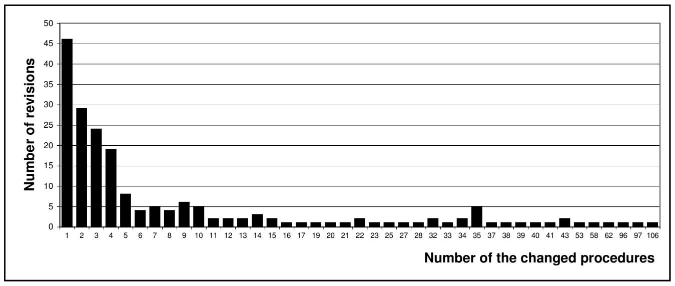


Figure 7. Histogram of the number of the changed procedures

are sets consisting of only the changed procedure, while the biggest sets we investigated included more than half of the total procedures. This is similar to what we found in earlier research for other software systems [18], and we expect that these impact sets bear the same properties as the more accurate traditional dependency based sets, namely that they are not much less precise. We investigated the overall size distribution of the different sets using a Monotone Size Graph [7] (see Figure 8). It was interesting to observe that there were a number of cases when the dependence set was of the same size, which was more than half of the program. This is probably caused by a big dependence cluster.

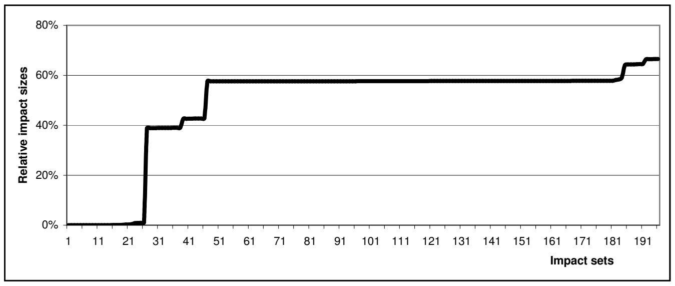


Figure 8. MSG graph of impact set sizes relative to the program size.

The size of the impact set is crucial to the successful application of the method. In certain applications, like guided code review, the smaller the better, but in other application like regression test selection even a modest size reduction can mean a lot of saving. In Figure 9, we can observe the overall distribution of the impact set sizes for the whole WebKit system. It can be seen that there is a significant number of sets with a small number of elements, usually below 100. This is plausible since in those cases the change propagation can be effective. However, even in the case when the sets contain 30000–40000 elements, we could achieve an improvement since this size is half of the system size.

However, from the point of view of our experiment, those impact sets are most important that were computed at the revisions taking part in the measurements. Furthermore, it is interesting to see what the distribution of those changes that were assumed to cause regression faults is. In Figure 10, we can see a similar histogram as the previous one but only for the 196 analyzed revisions, and those changed procedures in them

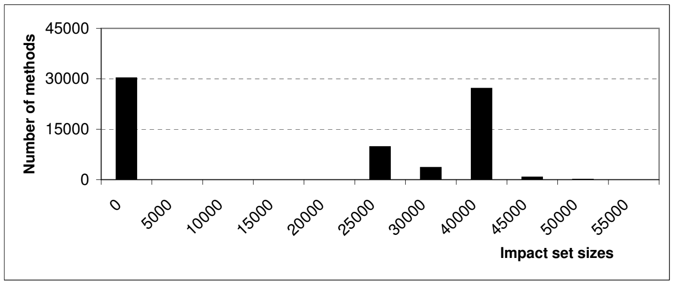


Figure 9. Histogram of the impact set sizes for the whole system

that caused failed tests. As expected, the layout is similar, but we can make an interesting observation that in this case there were relatively fewer procedures with small impact sets. We can speculate from this that if a change has a larger impact set, it will be more probable to result in a failure.

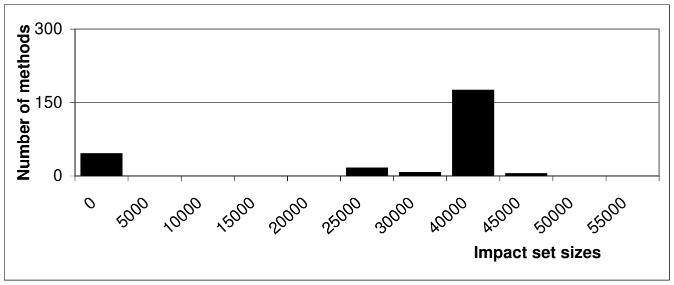


Figure 10. Histogram of the impact set sizes for the failure inducing changed procedures only

D. Change propagation prediction

As mentioned earlier, there were 12 revision pairs in the revision interval investigated that contain modified procedures at both ends.

In our final investigation, we manually reviewed all 12 revision pairs to find out the rate at which the impact sets at failure inducing changes include procedures at failure fixing places, and also to see the causes when the hit was missed. Unfortunately 8 revision pairs from the 12 were paired due to flakey tests, so they were dropped.

The remaining 4 pairs were the following:

- rev. 96996 – rev. 97006
- rev. 97121 – rev. 97123
- rev. 97282 – rev. 97289
- rev. 96779 – rev. 96789

We investigated these revision pairs manually. Revisions 97006 and 97123 reverted the modifications of the failed revisions, so these pairs are not so good in our investigations either.

Table IV summarizes our final results. In the remaining 2 revision pairs the impacts of the modifications of the $Failed$ revisions covered 75% of the modifications of the $Passed$ revisions. There are many factors that prevented to get a complete hit in all cases:

Table IV
HIT RATE OF THE IMPACT SETS

Revision pairs	rev. 97282 – rev. 97289	rev. 96779 – rev. 96789
Changed procedures at		
Failed revision	28	11
Passed revision	3	1
Covered by impact	2	1
Hit rate	66,67 %	100 %

- Introduction of new procedures. If the introduction of a new procedure is part of the fixing revision (or any of the revisions after the failed revision), naturally they could not be parts of the impact sets. In our measurement that was the reason of the partial hits.
- Deficiencies due to changes in a procedure name only. The connection of the changed procedures and the impact set members was done based on their names, parameters and return types, which was unsuccessful in some cases.
- Weaknesses due to imperfect analysis and representation. Like any other analysis, SEA also suffers from different problems that can be related to imperfect analysis and underlying program representation, most notably due to procedure call edges. Particularly, the ICCFG graph had deficiencies in some cases handling certain types of calls, for example, calls to operator functions.
- It is possible that other technical or algorithmic problems are responsible for missing hits, among them are those cases that can be attributed to hidden dependencies between program elements that even SEA could not find, but are probably causing incomplete change propagation.

E. Discussion

Although the final results of our experiment about the comparison of impact sets with the failed and fixed tests are somewhat different from what we expected, we think that the presented results are very useful, and provide a basis for further research. First, in applications like automatic regression testing any reduction in test size is beneficial. Second, the advantages of the method may be exploited to improve the manual change propagation process even if only small impact sets could be taken into account. In this case, large impact sets could be simply ignored, but with small impact sets we would get an increased defect detection probability.

The applicability of the SEA approach for a real size system is further supported. First, we implemented a complex analysis framework for our algorithm implementation that is integrated into the regular build and test process of the WebKit system. The algorithm is capable of calculating the dependencies for this big and complex system in a reasonable time compared to the original build process. This tool setup is now suitable for continuous measurement of each occurring revision and producing up to date information, and of course, we are now able to perform an arbitrary number of measurements for past revisions.

The impact set sizes show similar distribution to our earlier findings, though we still need to investigate ways to further reduce the impact set sizes to be more suitable for specific applications using, for example, weighting of the relations.

Finally, the experiences from manually investigating the relevant cases of the experiment we could identify further possibilities for the perfection of the algorithm and measurement method. With these enhancements we would probably get much higher hit rate, and that is what we plan to verify in the future.

F. Threats to validity

There are several threats to the validity of the presented method and experiment, apart from the weaknesses mentioned in the previous section.

When verifying the hit rate of the impact analysis, we could not precisely know which atomic changes caused the failure and the fix since

1) we aggregated the changes and the dependencies to procedure level, and
2) we could not take into account that a revision may implement more than one functionality at the same time.

This means that we assumed that all changes at a given revision belong to a failure inducing change or a fix.

Our method for finding suitable revision pairs neglects the fact that the fix can be done in more steps within the middle of the failing period and the final fix made when the status changes to passed. Our method however, is conservative in the sense that in both of these last two cases it verifies more possibilities than actually needed. And, as mentioned above, there could be some errors in the pairing also due to the so-called flakey tests.

We relied on the results of the testing environment, and did not take into account other sources of defect information like the bug database.

We verified only whether the impact set contains the actual required change, and did not discuss whether it would be actually possible to use the impact set for a specific review or regression testing activity. This could be hindered by, for example, the size of the impact set. Therefore we will conduct experiments in the future to find ways to reduce the size of the sets by applying some heuristics to prioritize the dependencies.

Finally, as with all other static code analysis techniques, our method still cannot guarantee complete coverage of possible dependencies due to various problems related to the dynamic nature of the languages, and other semantic, conceptual or logical dependencies in the software system. However, we believe that it is safe in a sense that all discoverable dependencies captured by more accurate methods are captured by SEA as well.

VII. CONCLUSIONS

In this work we summarized the algorithmic properties and existing implementations of our static impact analysis method Static Execute After and demonstrated its application to a large open source system WebKit. The algorithm bears several properties that make it suitable for the analysis of dependencies in large systems, which was impossible with more accurate dependency based methods, while not being significantly less precise.

Earlier, properties of the algorithm had been verified on a set of small to medium benchmark programs, but recently we applied the implementation to large systems like Mozilla, OpenOffice and WebKit. The latter was used as a case study in this work. It was difficult to arrive at the point where the tool can be executed regularly for each revision integrated into the build process. Using the set up tools we then performed a series of experiments to find out the size properties of the impact sets computed for a series of revisions, and to verify the traditional hypothesis of impact analysis that applying the impact set we can actually predict the required change propagation. We know of very few case studies verifying this using real change and regression test data.

Although we have already invested a significant amount of work into the implementation of the algorithm, we think we are still in the middle towards its regular industrial application. In this work we presented its applicability to large systems but it still remains to be checked how will it perform for a larger set of revision data and, eventually, in real situations. We are working toward applying it for real maintenance problems of the WebKit developers and see the actual gains in terms of reduced failure rate.

We also plan to perform more experiments with some relaxed variants of the algorithm that consider the most important dependencies first by, for instance, weighing the dependence edges according to their distance or multiplicity. Finally, it will also be interesting to perform a similar experiment using code coverage information of the tests, by which we could identify the failure inducing tests more reliably.

ACKNOWLEDGMENTS

The authors would like to thank György Hegedűs and Péter Siket for their help with the experiments. This research was supported by the Hungarian national grant OTKA K-73688.

REFERENCES

[1] M. Acharya and B. Robinson, "Practical change impact analysis based on static program slicing for industrial software systems," in *Proceedings of the 33rd ACM SIGSOFT International Conference on Software Engineering (ICSE)*, May 2011, pp. 746–765.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers. Principles, Techniques and Tools*. Addison Wesley, 1986.

[3] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, May 2005, pp. 432–441.

[4] Á. Beszédes, T. Gergely, Sz. Faragó, T. Gyimóthy, and F. Fischer, "The dynamic function coupling metric and its use in software evolution," in *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*. IEEE Computer Society, Mar. 21–23, 2007, pp. 103–112.

[5] Á. Beszédes, T. Gergely, J. Jász, G. Tóth, T. Gyimóthy, and V. Rajlich, "Computation of static execute after relation with applications to software maintenance," in *Proceedings of the 2007 IEEE International Conference on Software Maintenance (ICSM'07)*. IEEE Computer Society, Oct. 2007, pp. 295–304.

[6] D. Binkley and M. Harman, "A large-scale empirical study of forward and backward static slice size and context sensitivity," in *Proceedings of the International Conference on Software Maintenance (ICSM'03)*, Sep. 2003, pp. 44–53.

[7] ——, "Locating dependence clusters and dependence pollution," in *Proceedings of the 21st International Conference on Software Maintenance (ICSM'05)*. IEEE Computer Society, Sep. 2005, pp. 177–186.

[8] S. A. Bohner and R. S. Arnold, Eds., *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[9] L. C. Briand, J. Wüst, and H. Lounis, "Using coupling measurement for impact analysis in object-oriented systems," in *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, Sep. 1999, pp. 475–482.

[10] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich, "JRipples: A tool for program comprehension during incremental change," in *IWPC*, 2005, pp. 149–152.

[11] R. Ferenc, F. Magyar, Á. Beszédes, Á. Kiss, and M. Tarkiainen, "Columbus – Tool for Reverse Engineering Large Object Oriented Software Systems," in *Proceedings of the 7th Symposium on Programming Languages and Software Tools (SPLST 2001)*. University of Szeged, Jun. 2001, pp. 16–27.

[12] R. Ferenc, I. Siket, and T. Gyimóthy, "Extracting Facts from Open Source Software," in *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*. IEEE Computer Society, Sep. 2004, pp. 60–69.

[13] "The FrontEndART Homepage," `http://www.frontendart.com/`.

[14] "Homepage of GrammaTech's CodeSurfer," `http://www.grammatech.com/products/codesurfer`, GrammaTech, Inc.

[15] A. E. Hassan and R. C. Holt, "Predicting change propagation in software systems," in *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 284–293.

[16] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–61, 1990.

[17] J. Jász, "Static execute after algorithms as alternatives for impact analysis," *Peryodica Politechnica*, pp. pp. 163–176, 2008.

[18] J. Jász, Á. Beszédes, T. Gyimóthy, and V. Rajlich, "Static execute after/before as a replacement of traditional software dependencies," in *Proceedings of the 2008 IEEE International Conference on Software Maintenance (ICSM'08)*. IEEE Computer Society, Oct. 2008, pp. 137–146.

[19] J. Jász, "Dependence-based static program slicing and its approximations," PhD dissertation, University of Szeged, May 2009.

[20] W. Landi and B. G. Ryder, "Pointer-induced aliasing: a problem taxonomy," in *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, Jan. 1991, pp. 93–103.

[21] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," in *The 11th IEEE Working Conference on Reverse Engineering (WCRE'04)*, 2004.

[22] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Trans. Softw. Eng.*, vol. 22, 1996.

[23] B. G. Ryder, "Constructing the call graph of a program," *IEEE Trans. Softw. Eng.*, vol. SE-5, no. 3, pp. 216–226, May 1979.

[24] G. Tóth, C. Nagy, J. Jász, Á. Beszédes, and L. Fülöp, "CIASYS – change impact analysis at system level," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR'10)*, Mar. 2010, pp. 203–206.

[25] "The WebKit Wiki Homepage," `http://trac.webkit.org/wiki/Applications using WebKit`.

[26] "The WebKit Homepage," `http://www.webkit.org`.

[27] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 352–357, 1984.

[28] F. G. Wilkie and B. A. Kitchenham, "Coupling measures and change ripples in C++ application software," *Journal of Systems and Software*, vol. 52, no. 2–3, pp. 157–164, 2000.

[29] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 429–445, 2005.