Error Mining: Bug Detection through Comparison with Large Code Databases

Alexander Breckel
Institute of Software Engineering and Compiler Construction
University of Ulm
Ulm, Germany
alexander.breckel@uni-ulm.de

***Abstract*—Bugs are hard to find. Static analysis tools are capable of systematically detecting predefined sets of errors, but extending them to find new error types requires a deep understanding of the underlying programming language. Manual reviews on the other hand, while being able to reveal more individual errors, require much more time. We present a new approach to automatically detect bugs through comparison with a large code database. The source file is analyzed for similar but slightly different code fragments in the database. Frequent occurrences of common differences indicate a potential bug that can be fixed by applying the modification back to the original source file.**

In this paper, we give an overview of the resulting algorithm and some important implementation details. We further evaluate the circumstances under which good detection rates can be achieved. The results demonstrate that consistently high detection rates of up to 50% are possible for certain error types across different programming languages.

***Keywords*-comparison-based bug detection; code similarity; static analysis; code databases**

I. Introduction

Various automated methods have been developed to detect and analyze programming errors in source code. Automated detection approaches, in contrast to techniques like manual debugging or reviewing, offer a constant assistance to the programmer, without being time-consuming. However, lacking human comprehension, most of these approaches focus on finding a finite set of error types. The static analysis implemented in compilers, for example, handles only few errors like type or name conflicts, albeit with high precision.

Newer techniques on the other hand, like finding potential bugs through inconsistencies in code clones, can provide additional results by making use of more data. Since the underlying data may be imprecise or insufficient, this leads to false positives and undiscovered errors. Also, as is especially the case with the code clones technique, detected errors may be limited to special regions in the code.

In this paper, we present a new error detection approach that overcomes some of these limitations and can be seen as a generalization of existing techniques. Instead of code from a single project, a precomputed database containing code from multiple projects is used for comparison. Also, unlike existing techniques, *each* fragment in the analyzed source file is treated as a potential code clone, even those too short to be classified as duplicates. The underlying idea behind this treatment is that semantically similar code fragments often also *look* similar and can be treated as "weak" clones. Small inconsistencies with similar fragments in the database could therefore indicate potential errors and at the same time provide corresponding corrections.

After a brief look at related works in section II we describe the algorithm in section III and provide the results of our evaluation in section IV. Ideas and opportunities for future research are summarized in section V, followed by a conclusion in section VI.

The major contributions of this paper are:

- A new generic algorithm to find errors in source code by comparing it to a code database.
- A heuristic to sort and filter the results independent of the underlying source language.
- An empirical evaluation of the influence of various parameters based on artificial as well as real bugs.

II. Related Work

There exists quite a lot of research on the occurrence and detection of code clones. Roy et al. [1] present a comparison of different techniques, with the two main groups being token-based approaches as in Baker [2] and approaches based on syntax trees as in Baxter et al. [3]. Commonly used software includes the token-based CCFinder [4] and the tree-based DECKARD [5].

The idea of using inconsistencies in clones for bug detection is described and implemented by Li et al. [6] in their tool CP-Miner. Jiang et al. [7] extend this idea — quite literally — by including information of the surrounding control statements to gather more specific results. Leaving the field of code clones, Wasylkowski et al. [8] use a large number of mined function call sequences to successfully uncover defects in mature software projects.

In order to test detection approaches without having to validate results manually, predefined error-fix pairs proved very useful. We collect such pairs from a software repository similar to Livshits and Zimmermann [9], who mine revision histories to find commonly occurring bug patterns.

This paper represents a continuation of our previous work [10], which contains more details on the developed approach and its evaluation.

III. Algorithm

The main algorithm behind our approach can be separated into the following three steps:

1) Collect a set of meaningful fragments from the source code. We call each fragment a *context*.
2) For each context, find all similar matches in a code database, producing an extensive *result list*.
3) Group the results, before sorting and filtering them according to a confidence metric.

As an example, consider the following C code fragment as part of some larger function operating on arrays:

`for(j = 0; j <` $\boxed{-}$`n; j++) array[`$\boxed{i}$`] = j;`

The code contains two highlighted errors: $\boxed{-}$ should be removed and $\boxed{i}$ should be replaced by `j`. Our algorithm selects the above fragment as a context and finds the following similar context in a precomputed database:

```
for(i = 0; i < k; i++) arr[i] = i;
```

After calculating the similarities, differences and frequencies, the match is categorized as a potential bug and the before-after diff is presented as a possible fix.

The following sections describe the motivation and details behind each step.

A. Context Selection

Finding the right contexts turns out to be crucial to the quality of the results. Too short, and the results drown in false positives. Too long, and most promising matches are never even considered. Contexts containing only partial or unrelated code like `(x));}for(i=`, as one often gets with fixed context lengths, are useless as well.

We choose a dynamic tree-based approach that handles the above limitations pretty well. Every single node in the syntax tree is selected as a context, which results in every input token being represented in multiple nested contexts, one for each level of the tree hierarchy. Furthermore, every sequence of sibling nodes is considered a context, in order to better handle spans of consecutive statements and declarations. We call this a *context extension*, because it produces longer matches. The additional length improves matches that would otherwise be filtered later on due to a lack of matching tokens.

Contexts chosen by this method begin and end at syntactically clean positions. They are also nested and overlapping, varying greatly in length. This is important, because an error can be found on different levels of the tree hierarchy. We can calculate a separate confidence level for each of these matches and use only the best context for later steps.

One disadvantage of using the syntax tree is that the source code has to be parsable, which is not the case for incomplete or syntactically erroneous code. For some languages island grammars can be used instead, structuring the code at known delimiters like `;` or `{}`.

B. Database Search

After collecting a set of contexts from the input file, we perform a database search for similar and identical matches. The database itself contains a large set of contexts gathered previously from other source files in either the same or different projects.

Two contexts are considered similar if an edit distance stays below a predefined threshold. We experimented with different types of tree and token-based edit distances to improve the results. Initially, a tree edit distance seemed like a natural choice when working with syntax trees. However, we found that even small token changes due to programming errors can cause large changes to the syntactical structure of the code, distorting the results. We use a token-based Levenshtein distance instead, which calculates the smallest set of inserted, removed and modified tokens needed to transform one context into the other.

The edit distance is further extended to match contexts with different variable names. This is important, as variables and other identifiers can often be renamed without affecting the code semantics. We handle this by creating identifier bindings between both contexts and treating those bindings as equal tokens, like in the following example:

```
for(j = 0; j < -n; j++) array[i] = j;
    ↓      ↓    ↓  ↓         ↓  ↯   ↓
for(i = 0; i <  k; i++)   arr[i] = i;
```

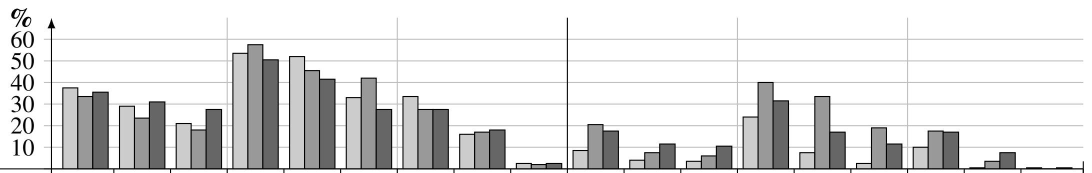
Although the two contexts contain only a single common variable occurrence `[i]`, this occurrence is considered an error due to the constructed binding `j` $\longrightarrow$ `i`.

The database search is the performance bottleneck of our algorithm and some optimizations are necessary to achieve reasonable lookup times. The edit distance calculation can be improved in most cases by returning as soon as the distance exceeds the predefined threshold. Contexts with a length difference above the threshold can also be ignored. Furthermore, the database can be prepared for faster lookups by using a trie or partial hashing.

C. Filtering the Results

Even with a low threshold setting the number of matches can reach more than a million, scaling linearly with the database size. However, most of these matches are nearly identical, as the same modification can be found multiple times in the database and on several levels of the tree hierarchy. A lot of additional matches contain very similar modifications, indicating the same error and differing only in the suggested values. By grouping these results we can reduce the number of results by several orders of magnitude. Nevertheless, the relevant matches are still hidden in an ocean of false positives that need further sorting and filtering.

We have found that some properties of a search result correlate with its quality and can therefore be used to calculate



░	C	37	29	21	53	52	33	33	16	2	8	4	3	24	7	2	10	0.5	0.5
▒	C#	33	23	18	57	45	42	27	17	2	20	7	6	40	33	19	17	3	0
▓	Haskell	35	31	27	50	41	27	27	18	2	17	11	10	31	17	11	17	7	0.5
	database	internal									external								
	error type	modification			insertion			deletion			modification			insertion			deletion		
	error length	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3

Figure 1. Detection rates (in %) for different languages, error types and databases using a minimum context size of 5 tokens.

a confidence level. Good matches frequently contain more common tokens T and corresponding identifiers V than most false positives, as well as a low edit distance D. A high ratio of identical matches in the database N to the number of similar matches M is also a good indicator. By sorting the results according to the following metric with empirically determined exponents we manage to get a median error position of 3, which means that half of the real bugs we find are contained within the first three suggestions of their respective result list:

$$Q = \frac{(T+1)^{4.1} * (N+1)^{2.3} * (V+1)^{5.2}}{(D+1)^{10.8} * (M+1)^{8.9}}$$

Our current results indicate that further filtering is possible by validating every suggested bugfix with a compiler. The remaining results can then be verified by running automated unit tests or even more generic test cases, if available. Although this increases the runtime by several orders of magnitude, it may still be worth the time, because it significantly reduces the amount of false positives.

IV. Evaluation

Good datasets containing enough errors to evaluate detection algorithms are hard to find. We therefore evaluated our approach using both artificially generated source code errors and real bugs collected from a source code repository. The generated errors consisted of one or more randomly inserted, removed or modified consecutive tokens. Obviously, this is not representative of real world bugs, but modified tokens are similar to typing errors and minor mistakes. Most typing errors can easily be detected by a lexical and syntactical analysis, so we made sure to generate only parsable errors while maintaining an equal distribution.

This data was then fed to our algorithm to analyze the influence of various parameters on the quality of the results, most important the programming language and the size and content of the database. To better understand the way the algorithm performs with different error types, we examined inserted, removed and modified tokens separately. We were also interested in the difference between using a database containing code from the analyzed project itself and from unrelated projects. By collecting two sets of projects for each programming language we could distinguish between such *internal* and *external* databases. Our results are summarized in Figure 1.

Errors consisting of inserted tokens are detected more often than removed tokens. This makes sense, as the algorithm has to reverse the process of modifying the tokens. To correct an inserted token it simply has to be removed, whereas a removed token has to be reinserted with the right value. As expected, bugs with longer modified token sequences are detected less often than short sequences.

The programming languages used in the analyzed code and database seem to have only little influence on the results compared to other parameters. This is important, as we don't use any semantic information of the language in any part of the algorithm or analysis. The only information used was a syntactical grammar. By providing a grammar we can therefore easily work with languages for which no further analysis tools exist.

Using an external database instead of the project itself results in significantly lower detection rates. We think that this is due to different projects containing less similar code compared to the files within a single project. The lack of code clones also certainly contributes to this. Nonetheless the numbers indicate that it is indeed possible to find bugs by comparing code to an independent database.

One way to improve the results is to increase the amount of files in the database. We measured the detection rate at various database sizes by adding files in a random order. The results in Figure 2 show a logarithmically increasing detection rate in the measured range. Further growth is probably possible, but requires a more efficient implementation. Surprisingly, even small databases containing less than 100 randomly selected files can be used to fix a large part of our seeded errors, which may be useful for some applications.

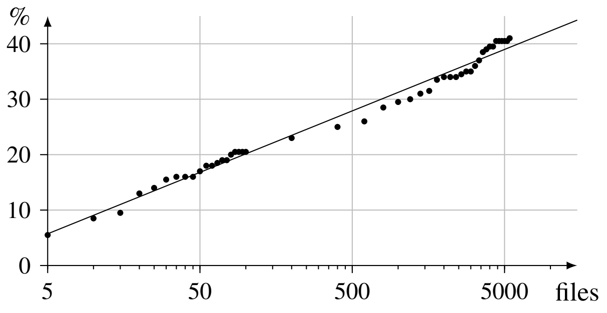


Figure 2. Detection rate (in %) with increasing database file counts.

To better understand the real world applicability, we further tried to reproduce real bug fixes with our approach. A set of bugs and accompanying fixes were collected by extracting 500 one-line changes from commits containing the word “fix” in the PostgreSQL Git repository. PostgreSQL was chosen due to its open source license and its use of the Git Source Code Management system, which motivates developers to perform regular and small commits like atomic bug fixes. Of these 500 errors, 37 (7,4%) are fixed by our algorithm exactly as they were by developers of the PostgreSQL team. The fixes can not be applied automatically, however, because the result lists also contain a lot of false positives. Although these are just preliminary results, they indicate that our approach can indeed detect and successfully fix certain real world bugs.

V. Future Research

Our evaluation shows that the detection rate scales well with increasing database sizes. In order to allow even larger databases, the runtime and size complexity needs to be improved. Faster access times can be achieved through specialized lookup structures like tries or finite automata. Efficient hashing may also be possible by choosing a different comparison function instead of a Levenshtein distance. These improvements could provide almost-instant feedback and allow an integration with existing IDEs.

The algorithm itself can be extended by allowing fragmented contexts consisting for example of all statements involving a common variable. Further semantic information can be used to select better initial contexts or improve the comparison function. Such information can also be used directly to filter false positives without the aforementioned use of a compiler or test cases.

Finally, the underlying idea of using large code databases for comparison can be extended to other applications like advanced code completion, spell checking and many others.

VI. Conclusion

Detecting bugs through comparison with code databases can achieve surprisingly positive results given the fact that no semantic information is used in the process. We showed that up to 50% of short typing errors can be fixed and even the correction of real world bugs is possible in some cases. Our approach requires little understanding of the analyzed programming language and can be adapted to new languages. However, a lot of additional work is necessary to make this approach usable, especially reducing the remaining amount of false positives. We believe that after further refinement our approach will be a useful addition to the landscape of static analysis tools.

References

[1] C. K. Roy, J. R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: a qualitative approach,” *Science of Computer Programming*, vol. 74, pp. 470–495, 2009.

[2] B. S. Baker, “On finding duplication and near-duplication in large software systems,” in *WCRE ’95: Proceedings of the Second Working Conference on Reverse Engineering*, 1995, pp. 86–95.

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier, “Clone detection using abstract syntax trees,” in *ICSM ’98: Proceedings of the International Conference on Software Maintenance*, 1998, pp. 368–377.

[4] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: a multilinguistic token-based code clone detection system for large scale source code,” *IEEE Transactions on Software Engineering*, vol. 28, pp. 654–670, 2002.

[5] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “DECKARD: scalable and accurate tree-based detection of code clones,” in *ICSE ’07: Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 96–105.

[6] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, “CP-Miner: a tool for finding copy-paste and related bugs in operating system code,” in *OSDI ’04: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, 2004, pp. 289–302.

[7] L. Jiang, Z. Su, and E. Chiu, “Context-based detection of clone-related bugs,” in *ESEC-FSE ’07: Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2007, pp. 55–64.

[8] N. Gruska, A. Wasylkowski, and A. Zeller, “Learning from 6,000 projects: lightweight cross-project anomaly detection,” in *ISSTA ’10: Proceedings of the 19th International Symposium on Software Testing and Analysis*, 2010, pp. 119–130.

[9] B. Livshits and T. Zimmermann, “Dynamine: finding common error patterns by mining software revision histories,” in *ESEC-FSE ’05: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005, pp. 296–305.

[10] A. Breckel, “Error Mining: Statische Analyse von Programmcode durch Vergleich mit umfangreichen Programmdatenbanken,” Diploma thesis, in German, University of Ulm, 2011.