Towards Scalability of Graph-Mining Based Bug Localisation

Frank Eichinger EICHINGER@IPD.UKA.DE
Klemens Böhm BOEHM@IPD.UKA.DE
Institute for Program Structures and Data Organisation (IPD), Universität Karlsruhe (TH), Germany

Keywords: graph mining, software bug localisation, weighted graph mining, call graphs

Abstract

(Semi-)automated bug localisation is an important issue in software engineering. Recent techniques based on call graphs and graph mining can locate bugs in relatively small programs, but do not scale for real-world applications. In this paper we describe a bug-localisation approach based on graph mining that has this property, at least according to preliminary experiments. Our main contribution is the definition and analysis of class-level call graphs, with encouraging results.

1. Introduction

Software quality is a big concern in industry. In many cases, bugs are not discovered before the software is delivered. In other cases, its release is delayed due to tedious quality-assurance measures. Both situations incur huge costs. As the localisation of bugs is known to be the most time-consuming part of debugging, automated methods for bug localisation are needed.

In software engineering, different bug localisation techniques have been proposed (for related work, see Eichinger et al., 2008). Recently, graph-mining techniques have been applied to this problem as well (Liu et al., 2005; Di Fatta et al., 2006; Eichinger et al., 2008). They make use of *dynamic call graphs*, which are representations of program executions. Such call graphs map methods to nodes and method calls to directed edges. The intuition behind graph mining based techniques is that they can detect frequent subgraphs in a set of call graphs representing correct program executions as well as frequent subgraphs in a set of failing executions. Then one can identify patterns which are typical in failing executions. This gives hints where a bug might be located. Finally, a score is derived which represents a likelihood of containing a bug, for every method. This score can then guide a manual debugging process, starting with the method scored highest.

Though these techniques work well with small programs, they suffer from the same problem: They do not scale for real-world software projects, where bug localisation would indeed be helpful. This is caused by the NP-complete problem of subgraph isomorphism, which is inherent in the graph-mining algorithms used by all techniques. Elaborate call graph reduction techniques are available that yield smaller graphs. This gives way to graph based bug localisation with somewhat larger programs. However, dynamic method call graphs of large real-life applications will probably remain too large for frequent subgraph mining.

To overcome the problem, we envision a scalable approach based on call graphs representing another level of granularity. Our starting point are the software entities represented by the nodes of such a graph. While the nodes represent methods in previous approaches, this level of detail can be changed in two directions: The granularity can become finer if nodes represent variables, and edges represent variable accesses. It can as well become coarser if the graph stands for classes and inter-class method calls (or packages and inter-package calls etc.). In this paper, we propose a call-graph representation at the class level. We develop an approach to mine such graphs and to localise bugs with state-of-the-art techniques. In our experiments, we demonstrate that class-level call graphs are suitable to locate certain bugs. (As probably any bug localisation technique, call-graph mining cannot locate any kind of bugs, i.e., if the bug does not affect the graph. Therefore, one should look at such techniques as orthogonal to other approaches.) We conclude with our vision how the techniques examined so far can evolve into a scalable approach for bug localisation.

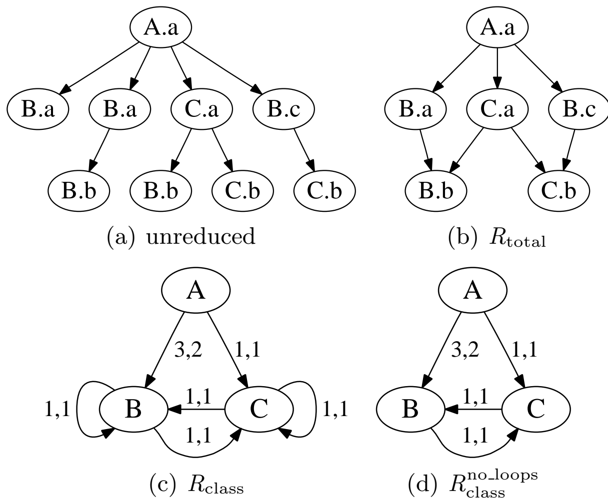
Presented at *ILP-MLG-SRL*, Leuven, Belgium, 2009.

	sg1 A→B		*sg1* A→C		*sg2* A→B		...	*sg1* A→A		*sg1* B→B		*sg1* C→C		*sg2* A→A		*sg2* B→B		...	Class
g_1	1	1	8	2	1	1	...	0	0	8	3	7	1	0	0	8	3	...	*failing*
g_2	0	0	0	0	5	2	...	0	0	0	0	0	0	0	0	6	3	...	*correct*

Table 1. Example table for analysis with feature selection algorithms.

2. Class-Level Call Graphs

Call graphs at the method level as obtained from program executions become very large. They may contain substructures repeatedly, caused by iterations and recursions, which lead to the huge size. All call graph mining based bug localisation approaches therefore use a graph reduction technique. The technique which achieves the strongest compression is *total reduction*, R_{total}. It maps all nodes representing the same method to one node. Figure 1 illustrates an unreduced call graph (a) and an R_{total} one (b). Notation: dots separate classes (capitals) from methods (lowercase).



(a) unreduced (b) R_{total} (c) R_{class} (d) $R_{\text{class}}^{\text{no_loops}}$

Figure 1. Call-graph representations.

Some call-graph representations at the method level use edge weights to represent call frequencies, which can be analysed as well. Our class-level call graphs, R_{class}, feature both notions, total reduction and edge weights. We map every node from the same class to the same node. (For a stronger compression, we merge nested Java classes into the node of the class where they are defined.) As it might be relevant information for bug localisation, we extend the edge weights and annotate every edge with a 2-tuple. It consists (1) of the total number of method invocations from the calling class to the callee class and (2) of the number of different methods called. Figure 1 contains an example of a R_{class} call graph (c) representing the original graph (a). As almost every method in a certain class usually calls other methods within the same class, the nodes in these graphs frequently have self-loops. For a further reduction of the R_{class} graphs for more efficient graph mining, we remove those loops as well ($R_{\text{class}}^{\text{no_loops}}$, see Figure 1(d)). However, we keep the tuples associated with the loops for the analysis step that follows the graph mining.

3. Bug Localisation Process

With method-level call graphs, Liu et al. (2005) and Di Fatta et al. (2006) have focused on frequent subgraph structures discriminating between correct and failing executions. In (Eichinger et al., 2008) we have shown that, in addition to structural approaches, it is essential to analyse call frequencies (edge weights) as well. We have applied frequent subgraph mining in a first step and analysed the edge weights in a subsequent postprocessing step. In this article we present a similar approach for class-level call graphs. In a first step we apply the *CloseGraph* algorithm (Yan & Han, 2003) to $R_{\text{class}}^{\text{no_loops}}$ graphs to obtain frequent subgraphs. We concentrate on those subgraphs which occur in both correct and failing executions, as we are interested in discriminating edge tuples.

We then assemble a table for the analysis of the edge tuples. These tuples can have a different significance in different *contexts*, which we represent with the subgraphs we have obtained before.[1] Each row in the table stands for a call graph g_i, i.e., for a program execution. The table contains one column for each element of the tuples of every edge in every frequent subgraph sg_j. Table 1 serves as an example. To explain some values, the edge from Class A to Class B in Subgraph sg_1 in program execution g_1 describes one call of a single method ('1,1'). In contrast, the edge from A to C in the same subgraph and the same execution is annotated with '8,2' (eight calls of two different methods). sg_1 is not contained in g_2 – zeros in all corresponding cells indicate this. In addition to the regular edge tuples, we include the tuples associated with the self-loops which we had omitted for the graph-mining step. We use one column for each tuple value for every node (class) in every subgraph (con-

[1] Preliminary experiments have demonstrated the usefulness of closed frequent subgraphs as contexts, compared to a direct analysis of edge weights in R_{class} graphs.

text). In the example, only Class A has no self-loops, and therefore all corresponding columns contain zeros.

We analyse the table with the *information gain* feature selection algorithm (Quinlan, 1993). This leads to a ranking of edges discriminating well between correct and failing executions. We use this ranking as a call frequency based likelihood that the classes contain a bug. As every class is contained in more than one column, the algorithm assigns more than one value to every class. We use the maximum for the ranking.

In contrast to all method-level approaches, we do not consider structural aspects. This is because preliminary experiments with $R_{\text{class}}^{\text{no_loops}}$ graphs have revealed that discriminating frequent subgraphs are rare. This is due to the finding that the reduction on the class level leads to call graphs with exactly the same structure in many cases. Only the edge tuples may be significantly different in correct and failing executions.

4. Experimental Results

For our evaluation we used the open source program *HtmlCleaner* (`htmlcleaner.sourceforge.net`) and instrumented it with nine different bugs, resulting in nine versions (see Table 2). The bugs are similar to the ones used in previous evaluations. The program consists of 28 classes from which 12 to 14 were executed in our experiments. The program is considerably larger than those used before (approx. 5,000 lines of code compared to 200 to 700). This size is too large to be mined with a method-level approach, and the program is well suited to demonstrate bug localisation on the class level. We applied our approach to $R_{\text{class}}^{\text{no_loops}}$ reduced graphs obtained from 100 executions of each version. The results are included in Table 2: The column 'Line' contains the position of the class in which the bug was instrumented in the ranking.

Bug	Modification	Line
Bug 1	method always returns `null`	1
Bug 2	wrong variable used	1
Bug 3	`return false` instead of `true`	6
Bug 4	`==` instead of `!=`	2
Bug 5	`if(true)` instead of condition	7
Bug 6	`==` instead of `>`	7
Bug 7	`x` instead of `x + 1`	7
Bug 8	omitted or-clause in `if`-condition	3
Bug 9	negated `if`-condition	1

Table 2. Bugs and results (small line numbers are better).

For five bugs our approach works very well: The bugs are found in Position 1 to 3 which facilitates a fast localisation. Note that only the buggy class is presented in the ranking, but its relationship to another class (an edge) is reported as well. This additionally aids debugging. Four bugs in turn are found in Position 6 or 7. These are cases where the localisation is less helpful for software developers. A closer inspection reveals that these bugs change variable values or affect library calls. Both does not affect the call graphs as they do not contain such values or calls. No approach based on the same kind of graphs can locate these bugs.

5. Conclusions and Future Directions

In our evaluation we have demonstrated that graph mining based bug localisation does work on class-level call graphs ($R_{\text{class}}^{\text{no_loops}}$), and that satisfactory results can be obtained. This allows to mine software projects which are significantly larger than before. Obviously, class-level bug localisation is not as precise as method-level techniques. To obtain more information where bugs might be hidden within a class, we propose to 'zoom in' into the suspicious regions (classes) and to apply a method-level approach on a graph containing methods from the suspicious classes only. This gives way to a proposal for scalable bug localisation: Starting at a high level of abstraction (say, packages) one identifies suspicious regions before 'zooming in', investigating the next level of abstraction (say, classes) and so forth. This approach requires definitions of call graphs with numerical attributes for every level considered, possibly including library calls. An important question for our future research is to investigate how well the approach outlined will work on levels coarser than the class-level abstraction.

Acknowledgement

We thank Roland Klug for his contributions.

References

Di Fatta, G., Leue, S., & Stegantova, E. (2006). Discriminative Pattern Mining in Software Fault Detection. *SOQUA Workshop.*

Eichinger, F., Böhm, K., & Huber, M. (2008). Mining Edge-Weighted Call Graphs to Localise Software Bugs. *ECML PKDD.*

Liu, C., Yan, X., Yu, H., Han, J., & Yu, P. S. (2005). Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs. *SDM Conference.*

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers.

Yan, X., & Han, J. (2003). CloseGraph: Mining Closed Frequent Graph Patterns. *KDD Conference.*