Chapter 17

SOFTWARE-BUG LOCALIZATION WITH GRAPH MINING

Frank Eichinger
Institute for Program Structures and Data Organization (IPD)
Universität Karlsruhe (TH), Germany
eichinger@ipd.uka.de

Klemens Böhm
Institute for Program Structures and Data Organization (IPD)
Universität Karlsruhe (TH), Germany
boehm@ipd.uka.de

Abstract In the recent past, a number of frequent subgraph mining algorithms has been proposed They allow for analyses in domains where data is naturally graph-structured. However, caused by scalability problems when dealing with large graphs, the application of graph mining has been limited to only a few domains. In software engineering, debugging is an important issue. It is most challenging to localize bugs automatically, as this is expensive to be done manually. Several approaches have been investigated, some of which analyze traces of repeated program executions. These traces can be represented as call graphs. Such graphs describe the invocations of methods during an execution. This chapter is a survey of graph mining approaches for bug localization based on the analysis of dynamic call graphs. In particular, this chapter first introduces the subproblem of reducing the size of call graphs, before the different approaches to localize bugs based on such reduced graphs are discussed. Finally, we compare selected techniques experimentally and provide an outlook on future issues.

Keywords: Software Bug Localization, Program Call Graphs

1. Introduction

Software quality is a huge concern in industry. Almost any software contains at least some minor bugs after being released. In order to avoid bugs, which incur significant costs, it is important to find and fix them before the release. In general, this results in devoting more resources to quality assurance. Software developers usually try to find and fix bugs by means of in-depth code reviews, along with testing and classical debugging. Locating bugs is considered to be the most time consuming and challenging activity in this context [6, 20, 24, 26] where the resources available are limited. Therefore, there is a need for semi-automated techniques guiding the debugging process [34]. If a developer obtains some hints where bugs might be localized, debugging becomes more efficient.

Research in the field of software reliability has been extensive, and various techniques have been developed addressing the identification of defect-prone parts of software. This interest is not limited to software-engineering research. In the machine-learning community, automated debugging is considered to be one of the ten most challenging problems for the next years [11]. So far, no bug localization technique is perfect in the sense that it is capable of discovering any kind of bug. In this chapter, we look at a relatively new class of bug localization techniques, the analysis of *call graphs* with *graph-mining* techniques. It can be seen as an approach orthogonal to and complementing existing techniques.

Graph mining, or more specifically *frequent subgraph mining*, is a relatively young discipline in data mining. As described in the other chapters of this book, there are many different techniques as well as numerous applications for graph mining. Probably the most prominent application is the analysis of chemical molecules. As the NP-complete problem of subgraph isomorphism [16] is an inherent part of frequent subgraph mining algorithms, the analysis of molecules benefits from the relatively small size of most of them. Compared to the analysis of molecular data, software-engineering artifacts are typically mapped to graphs that are much larger. Consequently, common graph-mining algorithms do not scale for these graphs. In order to make use of call graphs which reflect the invocation structure of specific program executions, it is key to deploy a suitable *call-graph-reduction* technique. Such techniques help to alleviate the scalability problems to some extent and allow to make use of graph-mining algorithms in a number of cases. As we will demonstrate, such approaches work well in certain cases, but some challenges remain. Besides scalability issues that are still unsolved, some call-graph-reduction techniques lead to another challenge: They introduce edge weights representing call frequencies. As graph-mining research has concentrated on structural and categorical domains, rather than on quantitative weights, we are not aware of any algorithm specialized in mining *weighted graphs*. Though this chapter presents a technique to analyze graphs with weighted edges, the technique is a composition of established algorithms rather than a universal weighted graph mining algorithm. Thus, besides mining large graphs, weighted graph mining is a further challenge for graph-mining research driven by the field of software engineering.

The remainder of this chapter is structured as follows: Section 2 introduces some basic principles of call graphs, bugs, graph mining and bug localization with such graphs. Section 3 gives an overview of related work in software engineering employing data-analysis techniques. Section 4 discusses different call-graph-reduction techniques. The different bug-localization approaches are presented and compared in Section 5 and Section 6 concludes.

2. Basics of Call Graph Based Bug Localization

This section introduces the concept of dynamic call graphs in Subsection 2.1. It presents some classes of bugs in Subsection 2.2 and Subsection 2.3 explains how bug localization with call graphs works in principle. A brief overview of key aspects of graph and tree mining in the context of this chapter is given in Subsection 2.4.

2.1 Dynamic Call Graphs

Call graphs are either *static* or *dynamic* [17]. A *static call graph* [1] can be obtained from the source code. It represents all methods[1] of a program as nodes and all possible method invocations as edges. *Dynamic call graphs* are of importance in this chapter. They represent an execution of a particular program and reflect the actual invocation structure of the execution. Without any further treatment, a call graph is a *rooted ordered tree*. The `main`-method of a program usually is the root, and the methods invoked directly are its children. Figure 17.1a is an abstract example of such a call graph where the root Node a represents the `main`-method.

Unreduced call graphs typically become very large. The reason is that, in modern software development, dedicated methods typically encapsulate every single functionality. These methods call each other frequently. Furthermore, iterative programming is very common, and methods calling other methods occur within loops, executed thousands of times. Therefore, the execution of even a small program lasting some seconds often results in call graphs consisting of millions of edges.

The size of call graphs prohibits a straightforward mining with state-of-the-art graph-mining algorithms. Hence, a reduction of the graphs which com-

[1] In this chapter, we use *method* interchangeably with *function*.

presses the graphs significantly but keeps the essential properties of an individual execution is necessary. Section 4 describes different reduction techniques.

2.2 Bugs in Software

In the software-engineering literature, there is a number of different definitions of *bugs*, *defects*, *errors*, *failures*, *faults* and the like. For the purpose of this chapter, we do not differentiate between them. It is enough to know that a bug in a program execution manifests itself by producing some other results than specified or by leading to some unexpected runtime behavior such as crashes or non-terminating runs. In the following, we introduce some types of bugs which are particularly interesting in the context of call graph based bug localization.

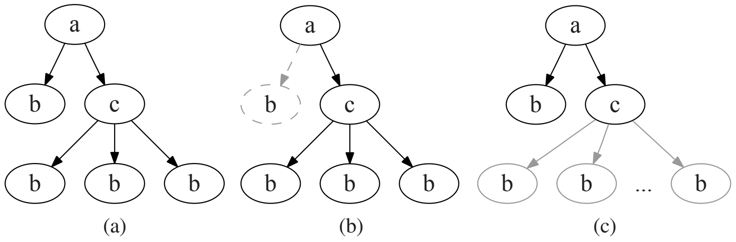

Figure 17.1. An unreduced call graph, a call graph with a structure affecting bug, and a call graph with a frequency affecting bug.

- **Crashing and non-crashing bugs:** *Crashing bugs* lead to an unexpected termination of the program. Prominent examples include null pointer exceptions and divisions by zero. In many cases, e.g., depending on the programming language, such bugs are not hard to find: A stack trace is usually shown which gives hints where the bug occurred. Harder to cope with are *non-crashing bugs*, i.e., failures which lead to faulty results without any hint that something went wrong during the execution. As non-crashing bugs are hard to find, all approaches to discover bugs with call-graph mining focus on them and leave aside crashing bugs.

- **Occasional and non-occasional bugs:** *Occasional bugs* are bugs which occur with some but not with any input data. Finding occasional bugs is particularly difficult, as they are harder to reproduce, and more test cases are necessary for debugging. Furthermore, they occur more frequently, as *non-occasional bugs* are usually detected early, and occasional bugs might only be found by means of extensive testing. As all bug-localization techniques presented in this chapter rely on comparing call graphs of failing and correct program executions, they deal with occasional bugs only. In other words, besides examples of failing program executions, there needs to be a certain number of correct executions.

- **Structure and call frequency affecting bugs**: This distinction is particularly useful when designing call graph based bug-localization techniques. *Structure affecting bugs* are bugs resulting in different shapes of the call graph where some parts are missing or occur additionally in faulty executions. An example is presented in Figure 17.1b, where Node b called from a is missing, compared to the original graph in Figure 17.1a. In this example, a faulty `if`-condition in Node a could have caused the bug. In contrast, *call frequency affecting bugs* are bugs which lead to a change in the number of calls of a certain subtree in faulty executions, rather than to completely missing or new substructures. In the example in Figure 17.1c, a faulty loop condition or a faulty `if`-condition inside a loop in Method c are typical causes for the increased number of calls of Method b.

As probably any bug-localization technique, call graph based bug localization is certainly not able to find all kinds of software bugs. For example, it is possible that bugs do not affect the call graph at all. For instance, if some mathematical expression calculates faulty results, this does not necessarily affect subsequent method calls and call graph mining can not detect this. Therefore, call graph based bug localization should be seen as a technique which complements other techniques, as the ones we will describe in Section 3. In this chapter we concentrate on deterministic bugs of single-threaded programs and leave aside bugs which are specific for such situations. However, the techniques described in the following might locate such bugs as well.

2.3 Bug Localization with Call Graphs

So far, several approaches have been proposed to localize bugs by means of call-graph mining [9, 13, 14, 25]. We will present them in detail in the following sections. In a nutshell, the approaches consist of three steps:

1 Deduction of call graphs from program executions,
assignment of labels *correct* or *failing*.

2 Reduction of call graphs.

3 Mining of call graphs,
analysis of the resulting frequent subgraphs.

Step 1: Deriving call graphs is relatively simple. They can be obtained by tracing program executions while testing, which is assumed to be done anyway. Furthermore, a classification of program executions as *correct* or *failing* is needed to find discriminative patterns in the last step. Obtaining the necessary information can be done easily, as quality assurance widely uses test suites which provide the correct results [18].

Step 2: Call-graph reduction is necessary to overcome the huge sizes of call graphs. This is much more challenging. It involves the decision how much information lost is tolerable when compressing the graphs. However, even if reduction techniques can facilitate mining in many cases, they currently do not allow for mining of arbitrary software projects. Details on call-graph reduction are presented in Section 4.

Step 3: This step includes frequent subgraph mining and the analysis of the resulting frequent subgraphs. The intuition is to search for patterns typical for faulty executions. This often results in a ranking of methods suspected to contain a bug. The rationale is that such a ranking is given to a software developer who can do a code review of the suspicious methods. The specifics of this step vary widely and highly depend on the graph-reduction scheme used. Section 5 discusses the different approaches in detail.

2.4 Graph and Tree Mining

Frequent subgraph mining has been introduced in earlier chapters of this book. As such techniques are of importance in this chapter, we briefly recapitulate those which are used in the context of bug localization based on call graph mining:

- **Frequent subgraph mining:** Frequent subgraph mining searches for the complete set of subgraphs which are frequent within a database of graphs, with respect to a user defined minimum support. Respective algorithms can mine connected graphs containing labeled nodes and edges. Most implementations also handle directed graphs and pseudo graphs which might contain self-loops and multiple edges. In general, the graphs analyzed can contain cycles. A prominent mining algorithm is *gSpan* [32].

- **Closed frequent subgraph mining:** Closed mining algorithms differ from regular frequent subgraph mining in the sense that only closed subgraphs are contained in the result set. A subgraph sg is called closed if no other graph is contained in the result set which is a supergraph of sg and has exactly the same support. Closed mining algorithms therefore produce more concise result sets and benefit from pruning opportunities which may speed up the algorithms. In the context of this chapter, the *CloseGraph* algorithm [33] is used, as closed subgraphs proved to be well suited for bug localization [13, 14, 25].

- **Rooted ordered tree mining:** Tree mining algorithms (a survey with more details can be found in [5]) work on databases of trees and exploit their characteristics. Rooted ordered tree mining algorithms work on *rooted ordered trees*, which have the following characteristics: In contrast to *free trees*, *rooted trees* have a dedicated root node, the `main`-method in call trees. *Ordered trees* preserve the order of outgoing edges of a node, which is not encoded in arbitrary graphs. Thus, call trees can keep the information that a certain node is called before another one from the same parent. Rooted ordered tree mining algorithms produce result sets of rooted ordered trees. They can be embedded in the trees from the original tree database, preserving the order. Such algorithms have the advantage that they benefit from the order, which speeds up mining significantly. Techniques in the context of bug localization sometimes use the *FREQT* rooted ordered tree mining algorithm [2]. Obviously, this can only be done when call trees are not reduced to graphs containing cycles.

3. Related Work

This chapter of the book surveys bug localization based on graph mining and dynamic call graphs. As many approaches orthogonal to call-graph mining have been proposed, this section on related work provides an overview of such approaches.

The most important distinction for bug localization techniques is if they are *static* or *dynamic*. Dynamic techniques rely on the analysis of program runs while static techniques do not require any execution. An example for a static technique is source code analysis which can be based on code metrics or different graphs representing the source code, e.g., static call graphs, control-flow graphs or program-dependence graphs. Dynamic techniques usually trace some information during a program execution which is then analyzed. This can be information on the values of variables, branches taken during execution or code segments executed.

In the remainder of this section we briefly discuss the different static and dynamic bug localization techniques. At the end of this section we present recent work in *mining of static program-dependence graphs* in a little more detail, as this approach makes use of graph mining. However, it is static in nature as it does not involve any program executions. It is therefore not similar to the mining schemes based on dynamic call graphs described in the remainder of this chapter.

Mining of Source Code. Software-complexity metrics are measures derived from the source code describing the complexity of a program or its methods. In many cases, complexity metrics correlate with defects in software [26, 34]. A standard technique in the field of 'mining software repositories' is to map post-release failures from a bug database to defects in static source code. Such a mapping is done in [26]. The authors derive standard complexity metrics from source code and build regression models based on them and the information if the software entities considered contain bugs. The regression models can then predict post-release failures for new pieces of software. A similar study uses decision trees to predict failure probabilities [21]. The approach in [30] uses regression techniques to predict the likelihood of bugs based on static usage relationships between software components. All approaches mentioned require a large collection of bugs and version history.

Dynamic Program Slicing. Dynamic program slicing [22] can be very useful for debugging although it is not exactly a bug localization technique. It helps searching for the exact cause of a bug if the programmer already has some clue or knows where the bug appears, e.g., if a stack trace is available. Program slicing gives hints which parts of a program might have contributed to a faulty execution. This is done by exploring data dependencies and revealing which statements might have affected the data used at the location where the bug appeared.

Statistical Bug Localization. Statistical bug localization is a family of dynamic, mostly data focused analysis techniques. It is based on instrumentation of the source code, which allows to capture the values of variables during an execution, so that patterns can be detected among the variable values. In [15], this approach is used to discover program invariants. The authors claim that bugs can be detected when unexpected invariants appear in failing executions or when expected invariants do not appear. In [23], variable values gained by instrumentation are used as features describing a program execution. These are then analyzed with regression techniques, which leads to potentially faulty pieces of code. A similar approach, but with a focus on the control flow, is [24]. It instruments variables in condition statements. It then calculates a ranking which yields high values when the evaluation of these statements differs significantly in correct and failing executions.

The instrumentation-based approaches mentioned either have a large memory footprint [6] or do not capture all bugs. The latter is caused by the usual practice not to instrument every part of a program and therefore not to watch every value, but to instrument sampled parts only. [23] overcomes this problem by collecting small sampled parts of information from productive code on large numbers of machines via the Internet. However, this does not facilitate the discovery of bugs before the software is shipped.

Analysis of Execution Traces. A technique using tracing and visualization is presented in [20]. It relies on a ranking of program components based on the information which components are executed more often in failing program executions. Though this technique is rather simple, it produces good bug-localization results. In [6], the authors go a step further and analyze sequences of method calls. They demonstrate that the temporal order of calls is more promising to analyze than considering frequencies only. Both techniques can be seen as a basis for the more sophisticated call graph based techniques this chapter focuses on. The usage of call sequences instead of call frequencies is a generalization which takes more structural information into account. Call graph based techniques then generalize from sequence-based techniques. They do so by using more complex structural information encoded in the graphs.

Mining of Static Program-Dependence Graphs. Recent work of Chang et al. [4] focuses on discovering *neglected conditions*, which are also known as *missing paths*, *missing conditions* and *missing cases*. They are a class of bugs which are in many cases *non-crashing occasional bugs* (cf. Subsection 2.2) – dynamic call graph based techniques target such bugs as well. An example of a neglected condition is a forgotten `case` in a `switch`-statement. This could lead to wrong behavior, faulty results in some occasions and is in general non-crashing.

Chang et al. work with static *program-dependence graphs* (*PDGs*) [28] and utilize graph-mining techniques. PDGs are graphs describing both control and data dependencies (edges) between elements (nodes) of a method or of an entire program. Figure 17.2a provides an example PDG representing the method $add(a, b)$ which returns the sum of its two parameters. Control dependencies are displayed by solid lines, data dependencies by dashed lines. As PDGs are static, only the number of instructions and dependencies within a method limit their size. Therefore, they are usually smaller than dynamic call graphs (see Sections 2 and 4). However, they typically become quite large as well, as methods often contain many dependencies. This is the reason why they cannot be mined directly with standard graph-mining algorithms. PDGs can be derived from source code. Therefore, like other static techniques, PDG analysis does not involve any execution of a program.

The idea behind [4] is to first determine *conditional rules* in a software project. These are rules (derived from PDGs, as we will see) occurring frequently within a project, representing fault-free patterns. Then, rule violations are searched, which are considered to be *neglected conditions*. This is based on the assumption that the more a certain pattern is used, the more likely it is to be a valid rule. The conditional rules are generated from PDGs by deriving *(topo-*

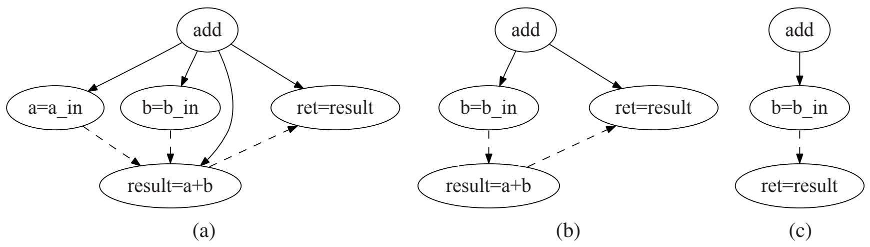

Figure 17.2. An example PDG, a subgraph and a topological graph minor.

logical) graph minors[2]. Such graph minors represent transitive intraprocedural dependencies. They can be seen – like subgraphs – as a set of smaller graphs describing the characteristics of a PDG. The PDG minors are obtained by employing a *heuristic maximal frequent subgraph-mining algorithm* developed by the authors. Then, an expert has to confirm and possibly edit the graph minors (also called *programming rules*) found by the algorithm. Finally, a *heuristic graph-matching algorithm*, which is developed by the authors as well, searches the PDGs to find the rule violations in question.

From a technical point of view, besides the PDG representation, the approach relies on the two new heuristic algorithms for maximal frequent subgraph mining and graph matching. Both techniques are not investigated from a graph theoretic point of view nor evaluated with standard data sets for graph mining. Most importantly, there are no guarantees for the heuristic algorithms: It remains unclear in which cases graphs are not found by the algorithms. Furthermore, the approach requires an expert to examine the rules, typically hundreds, by hand. However, the algorithms do work well in the evaluation of the authors.

The evaluation on four open source programs demonstrates that the approach finds most neglected conditions in real software projects. More precisely, 82% of all rules are found, compared to a manual investigation. A drawback of the approach is the relatively high false-positive rate which leads to a bug-detection precision of 27% on average.

Though graph-mining techniques similar to dynamic call graph mining (as presented in the following) are used in [4], the approaches are not related. The work of Chang et al. relies on static PDGs. They do not require any program execution, as dynamic call graphs do.

[2]A *graph minor* is a graph obtained by repeated deletions and edge contractions from a graph [10]. For *topological graph minors* as used in [4], in addition, paths between two nodes can be replaced with edges between both nodes. Figure 17.2 provides (a) an example PDG along with (b) a subgraph and (c) a topological graph minor. The latter is a minor of both, the PDG and the subgraph. Note that in general any subgraph of a graph is a minor as well.

4. Call-Graph Reduction

As motivated earlier, reduction techniques are essential for call graph based bug localization: Call graphs are usually very large, and graph-mining algorithms do not scale for such sizes. Call-graph reduction is usually done by a lossy compression of the graphs. Therefore, it involves the tradeoff between keeping as much information as possible and a strong compression. As some bug localization techniques rely on the temporal order of method executions, the corresponding reduction techniques encode this information in the reduced graphs.

In Subsection 4.1 we describe the possibly easiest reduction technique, which we call *total reduction*. In Subsection 4.2 we introduce various techniques for the reduction of iteratively executed structures. As some techniques make use of the temporal order of method calls during reduction, we describe these aspects in Subsection 4.3. We provide some ideas on the reduction of recursion in Subsection 4.4 and conclude the section with a brief comparison in Subsection 4.5.

4.1 Total Reduction

The *total reduction* technique is probably the easiest technique and yields good compression. In the following, we introduce two variants:

- **Total reduction (R_{total}).** Total reduction maps every node representing the same method in the call graph to a single node in the reduced graph. This may give way to the existence of loops (i.e., the output is a regular graph, not a tree), and it limits the size of the graph (in terms of nodes) to the number of methods of the program. In bug localization, [25] has introduced this technique, along with a temporal extension (see Subsection 4.3).

- **Total reduction with edge weights ($R_{\text{total\_w}}$).** [14] has extended the plain total reduction scheme (R_{total}) to include call frequencies: Every edge in the graph representing a method call is annotated with an edge weight. It represents the total number of calls of the callee method from the caller method in the original graph. These weights allow for more detailed analyses.

Figure 17.3 contains examples of the total reduction techniques: (a) is an unreduced call graph, (b) its total reduction (R_{total}) and (c) its total reduction with edge weights ($R_{\text{total\_w}}$).

In general, total reduction (R_{total} and $R_{\text{total\_w}}$) reduces the graphs quite significantly. Therefore, it allows graph mining based bug localization with software projects larger than other reduction techniques. On the other hand, much

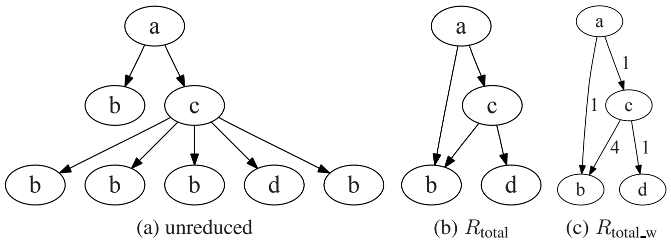

Figure 17.3. Total reduction techniques.

information on the program execution is lost. This concerns frequencies of the executions of methods (R_{total} only) as well as information on different structural patterns within the graphs (R_{total} and $R_{\text{total\_w}}$). In particular, the information is lost in which context (at which position within a graph) a certain substructure is executed.

4.2 Iterations

Next to total reduction, reduction based on the compression of iteratively executed structures (i.e., caused by loops) is promising. This is due to the frequent usage of iterations in today's software. In the following, we introduce two variants:

- **Unordered zero-one-many reduction ($R_{\text{01m\_unord}}$).** This reduction technique omits equal substructures of executions which are invoked more than twice from the same node. This ensures that many equal substructures called within a loop do not lead to call graphs of an extreme size. In contrast, the information that some substructure is executed several times is still encoded in the graph structure, but without exact numbers. This is done by doubling substructures within the call graph. Compared to total reduction (R_{total}), more information on a program execution is kept. The downside is that the call graph generally is much larger.

  This reduction technique is inspired by Di Fatta et al. [9] (cf. $R_{\text{01m\_ord}}$ in Subsection 4.3), but does not take the temporal order of the method executions into account. [13, 14] have used it for comparisons with other techniques which do not make use of temporal information.

- **Subtree reduction (R_{subtree}).** This reduction technique, proposed in [13, 14], reduces subtrees executed iteratively by deleting all but the first subtree and inserting the call frequencies as edge weights. In general, it therefore leads to smaller graphs than $R_{\text{01m\_unord}}$. The edge weights allow for a detailed analysis; they serve as the basis of the analysis technique described in Subsection 5.2. Details of the reduction technique are given in the remainder of this subsection.

Note that with R_{total}, and with $R_{\text{01m\_unord}}$ in most cases as well, the graphs of a correct and a failing execution with a *call frequency affecting bug* (cf. Subsection 2.2) are reduced to exactly the same graph. With R_{subtree} (and with $R_{\text{total\_w}}$ as well), the edge weights would be different when call frequency affecting bugs occur. Analysis techniques can discover this (cf. Subsection 5.2).

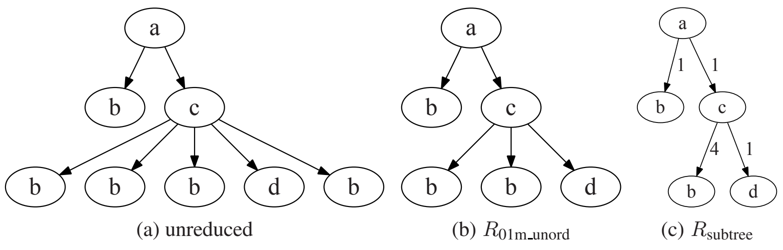

Figure 17.4. Reduction techniques based on iterations.

Figure 17.4 illustrates the two iteration-based reduction techniques: (a) is an unreduced call graph, (b) its zero-one-many reduction without temporal order ($R_{\text{01m\_unord}}$) and (c) its subtree reduction (R_{subtree}). Note that the four calls of b from c are reduced to two calls with $R_{\text{01m\_unord}}$ and to one edge with weight 4 with R_{subtree}. Further, the graph resulting from R_{subtree} has one node more than the one obtained from $R_{\text{total\_w}}$ in Figure 17.3c, but the same number of edges.

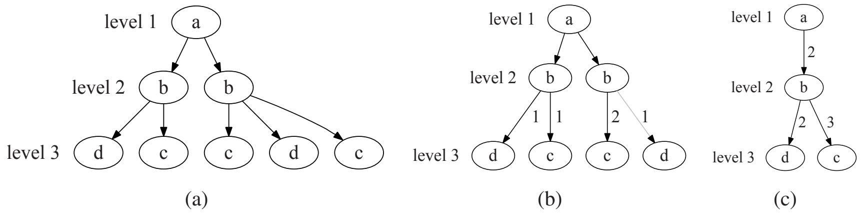

Figure 17.5. A raw call tree, its first and second transformation step.

For the subtree reduction (R_{subtree}), [14] organizes the call tree into n horizontal levels. The root node is at *level* 1. All other nodes are in levels numbered with the distance to the root. A na"ve approach to reduce the example call tree in Figure 17.5a would be to start at *level* 1 with Node a. There, one would find two child subtrees with a different structure – one could not merge anything. Therefore, one proceeds level by level, starting from *level* $n - 1$, as described in Algorithm 22. In the example in Figure 17.5a, one starts in *level* 2.

The left Node b has two different children. Thus, nothing can be merged there. In the right b, the two children c are merged by adding the edge weights of the merged edges, yielding the tree in Figure 17.5b. In the next level, $level$ 1, one processes the root Node a. Here, the structure of the two successor subtrees is the same. Therefore, they are merged, resulting in the tree in Figure 17.5c.

Algorithm 22 Subtree reduction algorithm.

```
1: Input: a call tree organized in $n$ levels
2: for $level = n - 1$ to 1 do
3:     for each $node$ in $level$ do
4:         merge all isomorph child-subtrees of $node$,
           sum up corresponding edge weights
5:     end for
6: end for
```

4.3 Temporal Order

So far, the call graphs described just represent the occurrence of method calls. Even though, say, Figures 17.3a and 17.4a might suggest that b is called before c in the root Node a, this information is not encoded in the graphs. As this might be relevant for discriminating faulty and correct program executions, the bug-localization techniques proposed in [9, 25] take the temporal order of method calls within one call graph into account. In Figure 17.6a, increasing integers attached to the nodes represent the order. In the following, we present the corresponding reduction techniques:

- **Total reduction with temporal edges ($R_{\textbf{total\_tmp}}$).** In addition to the total reduction (R_{total}), [25] uses so called *temporal edges*. The authors insert them between all methods which are executed consecutively and are invoked from the same method. They call the resulting graphs *software-behavior graphs*. This reduction technique includes the temporal order from the raw ordered call trees in the reduced graph representations. Technically, temporal edges are directed edges with another label, e.g., 'temporal', compared to other edges which are labeled, say, 'call'.

  As the graph-mining algorithms used for further analysis can handle edges labeled differently, the analysis of such graphs does not give way to any special challenges, except for an increased number of edges. In consequence, the totally reduced graphs loose their main advantage, their small size. However, taking the temporal order into account might help discovering certain bugs.

- **Ordered zero-one-many reduction ($R_{\mathrm{01m\_ord}}$).** This reduction technique proposed by Di Fatta et al. [9] makes use of the temporal order. This is done by representing the graph as a *rooted ordered tree*, which can be analyzed with an order aware mining algorithm. To include the temporal order, the reduction technique is changed as follows: While $R_{\mathrm{01m\_unord}}$ omits any equal substructure which is invoked more than twice from the same node, here only substructures are removed which are executed more than twice in direct sequence. This facilitates that all temporal relationships are retained. E.g., in the reduction of the sequence b, b, b, d, b (see Figure 17.6) only the third b is removed, and it is still encoded that b is called after d once.

  Depending on the actual execution, this technique might lead to extreme sizes of call trees. For example, if within a loop a Method a is called followed by two calls of b, the reduction leads to the repeated sequence a, b, b, which is not reduced at all. The rooted ordered tree miner in [9] partly compensates the additional effort for mining algorithms caused by such sizes, which are huge compared to $R_{\mathrm{01m\_unord}}$. Rooted ordered tree mining algorithms scale significantly better than usual graph mining algorithms [5], as they make use of the order.

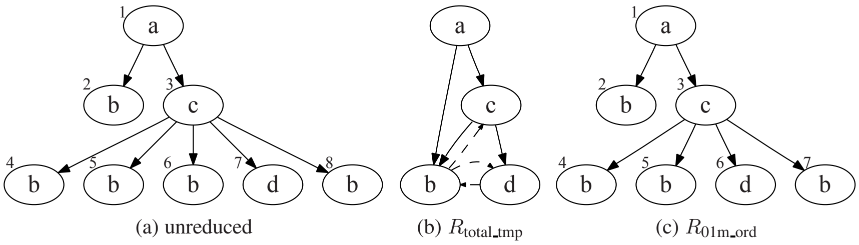

Figure 17.6. Temporal information in call graph reductions.

Figure 17.6 illustrates the two graph reductions which are aware of the temporal order. (The integers attached to the nodes represent the invocation order.) (a) is an unreduced call graph, (b) its total reduction with temporal edges (dashed, $R_{\mathrm{total\_tmp}}$) and (c) its ordered zero-one-many reduction ($R_{\mathrm{01m\_ord}}$). Note that, compared to $R_{\mathrm{01m\_unord}}$, $R_{\mathrm{01m\_ord}}$ keeps a third Node b called from c, as the direct sequence of nodes labeled b is interrupted.

4.4 Recursion

Another challenge with the potential to reduce the size of call graphs is recursion. The total reductions (R_{total}, $R_{\mathrm{total\_w}}$ and $R_{\mathrm{total\_tmp}}$) implicitly handle recursion as they reduce both iteration and recursion. E.g., when every method is collapsed to a single node, (self-)loops implicitly represent recursion. Besides that, recursion has not been investigated much in the context of call-graph reduction and in particular not as a starting point for reductions in addition to iterations. The reason for that is, as we will see in the following, that the reduction of recursion is less obvious than reducing iterations and might finally result in the same graphs as with a total reduction. Furthermore, in compute-intensive applications, programmers frequently replace recursions with iterations, as this avoids costly method calls. Nevertheless, we have investigated recursion-based reduction of call graphs to a certain extent and present some approaches in the following. Two types of recursion can be distinguished:

- **Direct recursion.** When a method calls itself directly, such a method call is called a *direct recursion*. An example is given in Figure 17.7a where Method b calls itself. Figure 17.7b presents a possible reduction represented with a self-loop at Node b. In Figure 17.7b, edge weights as in R_{subtree} represent both frequencies of iterations and the depth of direct recursion.

- **Indirect recursion.** It may happen that some method calls another method which in turn calls the first one again. This leads to a chain of method calls as in the example in Figure 17.7c where b calls c which again calls b etc. Such chains can be of arbitrary length. Obviously, such *indirect recursions* can be reduced as shown in Figures 17.7c–(d). This leads to the existence of loops.

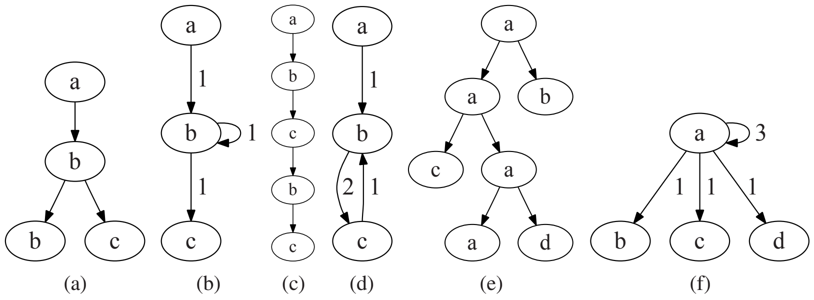

Figure 17.7. Examples for reduction based on recursion.

Both types of recursion are challenging when it comes to reduction. Figures 17.7e–(f) illustrate one way of reducing direct recursions. While the subsequent reflexive calls of a are merged into a single node with a weighted self-loop, b, c and d become siblings. As with total reductions, this leads to new structures which do not occur in the original graph. In bug localization, one might want to avoid such artifacts. E.g., d called from exactly the same method as b could be a structure-affecting bug which is not found when such artifacts occur. The problem with indirect recursion is that it can be hard to detect and becomes expensive to detect all occurrences of long-chained recursion. To conclude, when reducing recursions, one has to be aware that, as with total reduction, some artifacts may occur.

4.5 Comparison

To compare reduction techniques, we must look at the level of compression they achieve on call graphs. Table 17.1 contains the sizes of the resulting graphs (increasing in the number of edges) when different reduction techniques are applied to the same call graph. The call graph used here is obtained from an execution of the Java diff tool taken from [8] used in the evaluation in [13, 14]. Clearly, the effect of the reduction techniques varies extremely depending on the kind of program and the data processed. However, the small program used illustrates the effect of the various techniques. Furthermore it can be expected that the differences in call-graph compressions become more significant with increasing call graph sizes. This is because larger graphs tend to offer more possibilities for reductions.

Reduction	Nodes	Edges
R_{total}, $R_{\text{total\_w}}$	22	30
R_{subtree}	36	35
$R_{\text{total\_tmp}}$	22	47
$R_{\text{01m\_unord}}$	62	61
$R_{\text{01m\_ord}}$	117	116
unreduced	2199	2198

Table 17.1. Examples for the effect of call graph reduction techniques.

Obviously, the total reduction (R_{total} and $R_{\text{total\_w}}$) achieves the strongest compression and yields a reduction by two orders of magnitude. As 22 nodes remain, the program has executed exactly this number of different methods. The subtree reduction (R_{subtree}) has significantly more nodes but only five more edges. As – roughly speaking – graph-mining algorithms scale with the number of edges, this seems to be tolerable. We expect the small increase in the number of edges to be compensated by the increase in structural information encoded. The unordered zero-one-many reduction technique ($R_{\text{01m\_unord}}$) again yields somewhat larger graphs. This is because repetitions are represented as doubled substructures instead of edge weights. With the total reduction with temporal edges ($R_{\text{total\_tmp}}$), the number of edges increases by roughly 50% due to the temporal information, while the ordered zero-one-many reduction ($R_{\text{01m\_ord}}$) almost doubles this number. Subsection 5.4 assesses the effectiveness of bug localization with the different reduction techniques along with the localization methods.

Clearly, some call graph reduction techniques also are expensive in terms of runtime. However, we do not compare the runtimes, as the subsequent graph mining step usually is significantly more expensive.

To summarize, different authors have proposed different reduction techniques, each one together with a localization technique (cf. Section 5): the total reduction ($R_{\text{total\_tmp}}$) in [25], the zero-one-many reduction ($R_{\text{01m\_ord}}$) in [9] and the subtree reduction (R_{subtree}) in [13, 14]. Some of the reductions can be used or at least be varied in order to work together with a bug localization technique different from the original one. In Subsection 5.4, we present original and varied combinations.

5. Call Graph Based Bug Localization

This section focuses on the third and last step of the generic bug-localization process from Subsection 2.3, namely frequent subgraph mining and bug localization based on the mining results. In this chapter, we distinguish between structural approaches [9, 25] and the frequency-based approach used in [13, 14].

In Subsections 5.1 and 5.2 we describe the two kinds of approaches. In Subsection 5.3 we introduce several techniques to integrate the results of structural and frequency-based approaches. We present some comparisons in Subsection 5.4.

5.1 Structural Approaches

Structural approaches for bug localization can locate *structure affecting bugs* (cf. Subsection 2.2) in particular. Approaches following this idea do so either in isolation or as a complement to a frequency-based approach. In most cases, a likelihood $P(m)$ that Method m contains a bug is calculated, for every method. This likelihood is then used to rank the methods. In the following, we refer to it as *score*. In the remainder of this subsection, we introduce and discuss the different structural scoring approaches.

The Approach by Di Fatta et al. In [9], the $R_{\text{01m\_ord}}$ call-graph reduction is used (cf. Section 4), and the rooted ordered tree miner *FREQT* [2] is employed to find frequent subtrees. The call trees analyzed are large and lead to scalability problems. Hence, the authors limit the size of the subtrees searched to a maximum of four nodes. Based on the results of frequent subtree mining, they define the *specific neighborhood* (SN). It is the set of all subgraphs contained in all call graphs of failing executions which are not frequent in call graphs of correct executions:

$$SN := \{sg \mid (supp(sg, D_{\text{fail}}) = 100\%) \wedge \neg(supp(sg, D_{\text{corr}}) \geq minSup)\}$$

where $supp(g, D)$ denotes the support of a graph g, i.e., the fraction of graphs in a graph database D containing g. D_{fail} and D_{corr} denote the sets of call graphs of failing and correct executions. [9] uses a minimum support $minSup$ of 85%.

Based on the *specific neighborhood*, a *structural score* P_{SN} is defined:

$$P_{\text{SN}}(m) := \frac{supp(g_m, SN)}{supp(g_m, SN) + supp(g_m, D_{\text{corr}})}$$

where g_m denotes all graphs containing Method m. Note that P_{SN} assigns the value 0 to methods which do not occur within SN and the value 1 to methods which occur in SN but not in correct program executions D_{corr}.

The Approach by Eichinger et al. The notion of *specific neighborhood* (SN) has the problem that no support can be calculated when the SN is empty.[3] Furthermore, experiments of ours have revealed that the P_{SN}-scoring only works well if a significant number of graphs is contained in SN. This depends on the graph reduction and mining techniques and has not always been the case in the experiments. Thus, to complement the frequency-based scoring (cf. Subsection 5.2), another structural score is defined in [14]. It is based on the set of frequent subgraphs which occur in failing executions only, SG_{fail}. The structural score P_{fail} is calculated as the support of m in SG_{fail}:

$$P_{\text{fail}}(m) := supp(g_m, SG_{\text{fail}})$$

Further Support-based Approaches. Both the P_{SN}-score [9] and the P_{fail}-score [14] have their weaknesses. Both approaches consider structure affecting bugs which lead to additional substructures in call graphs corresponding to failing executions. In the SN, only substructures occurring in *all* failing executions (D_{fail}) are considered – they are ignored if a single failing execution does not contain the structure. The P_{fail}-score concentrates on subgraphs occurring in failing executions only (SG_{fail}), although they do not need to be contained in *all* failing executions. Therefore, both approaches might not find structure affecting bugs leading not to additional structures but to fewer structures. The weaknesses mentioned have not been a problem so far, as they have rarely affected the respective evaluation, or the combination with another ranking method has compensated it.

[3] [9] uses a simplistic fall-back approach to deal with this effect.

One possible solution for a broader structural score is to define a score based on two support values: The support of every subgraph sg in the set of call graphs of correct executions $supp(sg, D_{\text{corr}})$ and the respective support in the set of failing executions $supp(sg, D_{\text{fail}})$. As we are interested in the support of methods and not of subgraphs, the maximum support values of all subgraphs sg in the set of subgraphs SG containing a certain Method m can be derived:

$$s_{\text{fail}}(m) := \max_{\{sg \mid sg \in SG,\, m \in sg\}} supp(sg, D_{\text{fail}})$$
$$s_{\text{corr}}(m) := \max_{\{sg \mid sg \in SG,\, m \in sg\}} supp(sg, D_{\text{corr}})$$

Example 17.1. *Think of Method* a*, called from the* `main`*-method and containing a bug. Let us assume there is a subgraph* main $\rightarrow$ a *(where '$\rightarrow$' denotes an edge between two nodes) which has a support of 100% in failing executions and 40% in correct ones. At the same time there is the subgraph* main $\rightarrow$ a $\rightarrow$ b *where* a *calls* b *afterwards. Let us say that the bug occurs exactly in this constellation. In this situation,* main $\rightarrow$ a $\rightarrow$ b *has a support of 0% in D_{corr} while it has a support of 100% in D_{fail}. Let us further assume that there also is a much larger subgraph* sg *which contains* a *and occurs in 10% of all failing executions. The value s_{fail}(a) therefore is 100%, the maximum of 100% (based on subgraph* main $\rightarrow$ a*), 100% (based on* main $\rightarrow$ a $\rightarrow$ b*) and 10% (based on sg).*

With the two relative support values s_{corr} and s_{fail} as a basis, new structural scores can be defined. One possibility would be the absolute difference of s_{fail} and s_{corr}:

$$P_{\text{fail-corr}}(m) = |s_{\text{fail}}(m) - s_{\text{corr}}(m)|$$

Example 17.2. *To continue Example 17.1, $P_{fail\text{-}corr}$(a) is 60%, the absolute difference of 40% (s_{corr}(a)) and 100% (s_{fail}(a)). We do not achieve a higher value than 60%, as Method* a *also occurs in bug-free subgraphs.*

The intuition behind $P_{\text{fail-corr}}$ is that both kinds of structure affecting bugs are covered: (1) those which lead to additional structures (high s_{fail} and low to moderate s_{corr} values like in Example 17.2) and (2) those leading to missing structures (low s_{fail} and moderate to high s_{corr}). In cases where the support in both sets is equal, e.g., both are 100% for the `main`-method, $P_{\text{fail-corr}}$ is zero. We have not yet evaluated $P_{\text{fail-corr}}$ with real data. It might turn out that different but similar scoring methods are better.

The Approach by Liu et al. Although [25] is the first study which applies graph mining techniques to dynamic call graphs to localize non-crashing bugs, this work is not directly compatible to the other approaches described so far. In [25], bug localization is achieved by a rather complex classification process, and it does not generate a ranking of methods suspected to contain a bug, but a set of such methods.

The work is based on the $R_{\text{total\_tmp}}$ reduction technique and works with total reduced graphs with temporal edges (cf. Section 4). The call graphs are mined with a variant of the *CloseGraph* algorithm [33]. This step results in frequent subgraphs which are turned into binary features characterizing a program execution: A boolean feature vector represents every execution. In this vector, every element indicates if a certain subgraph is included in the corresponding call graph. Using those feature vectors, a support-vector machine (SVM) is learned which decides if a program execution is correct or failing. More precisely, for every method, two classifiers are learned: one based on call graphs including the respective method, one based on graphs without this method. If the precision rises significantly when adding graphs containing a certain method, this method is deemed more likely to contain a bug. Such methods are added to the so-called *bug-relevant function set*. Its functions usually line up in a form similar to a stack trace which is presented to a user when a program crashes. Therefore, the bug-relevant function set serves as the output of the whole approach. This set is given to a software developer who can use it to locate bugs more easily.

5.2 Frequency-based Approach

The frequency-based approach for bug localization by Eichinger et al. [13, 14] is in particular suited to locate *frequency affecting bugs* (cf. Subsection 2.2), in contrast to the structural approaches. It calculates a score as well, i.e., the likelihood to contain a bug, for every method.

After having performed frequent subgraph mining with the *CloseGraph* algorithm [33] on call graphs reduced with the R_{subtree} technique, Eichinger et al. analyze the edge weights. As an example, a call-frequency affecting bug increases the frequency of a certain method invocation and therefore the weight of the corresponding edge. To find the bug, one has to search for edge weights which are increased in failing executions. To do so, they focus on frequent subgraphs which occur in both correct and failing executions. The goal is to develop an approach which automatically discovers which edge weights of call graphs from a program are most significant to discriminate between *correct* and *failing*. To do so, one possibility is to consider different *edge types*, e.g., edges having the same calling Method m_{s} (start) and the same method called m_{e} (end). However, edges of one type can appear more than once within one subgraph and, of course, in several different subgraphs. Therefore, the authors analyze every edge in every such location, which is referred to as a *context*. To specify the exact location of an edge in its context within a certain subgraph, they do not use the method names, as they may occur more than once. Instead, they use a unique id for the calling node (id_s) and another one for the method called (id_e). All ids are valid within their subgraph. To sum up, edges in its context in a certain subgraph sg are referenced with the following tuple: (sg, id_s, id_e). A certain bug does not affect all method calls (edges) of the same type, but method calls of the same type in the same context. Therefore, the authors assemble a feature table with every edge in every context as columns and all program executions in the rows. The table cells contain the respective edge weights. Table 17.2 serves as an example.

	$a \rightarrow b$ (sg_1, id_1, id_2)	$a \rightarrow b$ (sg_1, id_1, id_3)	$a \rightarrow b$ (sg_2, id_1, id_2)	$a \rightarrow c$ (sg_2, id_1, id_3)	$\cdots$	Class
g_1	0	0	13	6513	$\cdots$	*correct*
g_2	512	41	8	12479	$\cdots$	*failing*
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

Table 17.2. Example table used as input for feature-selection algorithms.

The first column contains a reference to the program execution or, more precisely, to its reduced call graph $g_i \in G$. The second column corresponds to the first subgraph (sg_1) and the edge from id_1 (Method a) to id_2 (Method b). The third column corresponds to the same subgraph (sg_1) but to the edge from id_1 to id_3. Note that both id_2 and id_3 represent Method b. The fourth column represents an edge from id_1 to id_2 in the second subgraph (sg_2). The fifth column represents another edge in sg_2. Note that ids have different meanings in different subgraphs. The last column contains the class *correct* or *failing*. If a certain subgraph is not contained in a call graph, the corresponding cells have value 0, like g_1, which does not contain sg_1. Graphs (rows) can contain a certain subgraph not just once, but several times at different locations. In this case, averages are used in the corresponding cells of the table.

The table structure described allows for a detailed analysis of edge weights in different contexts within a subgraph. Algorithm 23 describes all subsequent steps in this subsection. After putting together the table, Eichinger et al. deploy a standard feature-selection algorithm to score the columns of the table and thus the different edges. They use an entropy-based algorithm from the *Weka* data-mining suite [31]. It calculates the information gain $InfoGain$ [29] (with respect to the class of the executions, *correct* or *failing*) for every column (Line 23 in Algorithm 23). The information gain is a value between 0 and 1, interpreted as a likelihood of being responsible for bugs. Columns with an information gain of 0, i.e., the edges always have the same weights in both classes, are discarded immediately (Line 23 in Algorithm 23).

Call graphs of failing executions frequently contain bug-like patterns which are caused by a preceding bug. Eichinger et al. call such artifacts *follow-up*

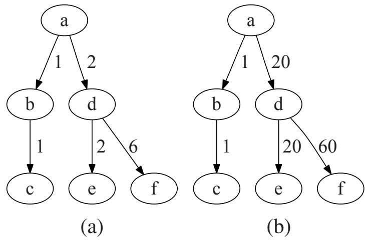

Figure 17.8. Follow-up bugs.

bugs. Figure 17.8 illustrates a follow-up bug: (a) represents a bug free version, (b) contains a call frequency affecting bug in Method a which affects the invocations of d. Here, this method is called 20 times instead of twice. Following the R_{subtree} reduction, this leads to a proportional increase in the number of calls in Method d. [14] contains more details how follow-up bugs are detected and removed from the set of edges E (Line 23 of Algorithm 23).

Algorithm 23 Procedure to calculate $P_{\text{freq}}(m_{\text{s}}, m_{\text{e}})$ and $P_{\text{freq}}(m)$.

```
1: Input: a set of edges $e \in E$, $e = (sg, id_{\text{s}}, id_{\text{e}})$
2: assign every $e \in E$ its information gain $InfoGain$
3: $E = E \setminus \{e \mid e.InfoGain = 0\}$
4: remove follow-up bugs from $E$
5: $E_{(m_{\text{s}},m_{\text{e}})} = \{e \mid e \in E \wedge e.id_{\text{s}}.label = m_{\text{s}} \wedge e.id_{\text{e}}.label = m_{\text{e}}\}$
6: $P_{\text{freq}}(m_{\text{s}}, m_{\text{e}}) = \max_{e \in E_{(m_{\text{s}},m_{\text{e}})}} (e.InfoGain)$
7: $E_m = \{e \mid e \in E \wedge e.id_{\text{s}}.label = m\}$
8: $P_{\text{freq}}(m) = \max_{e \in E_m} (e.InfoGain)$
```

At this point, Eichinger et al. calculate likelihoods of method invocations containing a bug, for every invocation (described by a calling Method m_{s} and a method called m_{e}). They call this score $P_{\text{freq}}(m_{\text{s}}, m_{\text{e}})$, as it is based on the call frequencies. To do the calculation, they first determine sets $E_{(m_{\text{s}},m_{\text{e}})}$ of edges $e \in E$ for every method invocation in Line 23 of Algorithm 23. In Line 23, they use the $\max()$ function to calculate $P_{\text{freq}}(m_{\text{s}}, m_{\text{e}})$, the maximum $InfoGain$ of all edges (method invocations) in E. In general, there are many edges in E with the same method invocation, as an invocation can occur in different contexts. With the $\max()$ function, the authors assign every invocation the score from the context ranked highest.

Example 17.3. *An edge from* a *to* b *is contained in two subgraphs. In one subgraph, this edge* a $\rightarrow$ b *has a low $InfoGain$ value of 0.1. In the other subgraph, and therefore in another context, the same edge has a high $InfoGain$*

value of 0.8, i.e., a bug is relatively likely. As one is interested in these cases, lower scores for the same invocation are less important, and only the maximum is considered.

At the moment, the ranking does not only provide the score for a method invocation, $P_{\text{freq}}(m_{\text{s}}, m_{\text{e}})$, but also the subgraphs where it occurs and the exact embeddings. This information might be important for a software developer. The authors report this information additionally. To ease comparison with other approaches not providing this information, they also calculate $P_{\text{freq}}(m)$ for every calling Method m in Lines 23 and 23 of Algorithm 23. The explanation is analogous to the one of the calculation of $P_{\text{freq}}(m_{\text{s}}, m_{\text{e}})$ in Lines 23 and 23.

5.3 Combined Approaches

As discussed before, structural approaches are well-suited to locate *structure affecting bugs*, while frequency-based approaches focus on *call frequency affecting bugs*. Therefore, it seems to be promising to combine both approaches. [13] and [14] have investigated such strategies.

In [13], Eichinger et al. have combined the frequency-based approach with the P_{SN}-score [9]. In order to calculate the resulting score, the authors use the approach by Di Fatta et al. [9] without temporal order: They use the $R_{\text{01m\_unord}}$ reduction with a general graph miner, *gSpan* [32], in order to calculate the structural P_{SN}-score. They derived the frequency-based P_{freq}-score as described before after mining the same call graphs but with the R_{subtree} reduction and the *CloseGraph* algorithm [33] and different mining parameters. In order to combine the two scores derived from the results of two graph mining runs, they calculated the arithmetic mean of the normalized scores:

$$P_{\text{comb[13]}}(m) = \frac{P_{\text{freq}}(m)}{2 \max\limits_{n \in sg \in D}(P_{\text{freq}}(n))} + \frac{P_{\text{SN}}(m)}{2 \max\limits_{n \in sg \in D}(P_{\text{SN}}(n))}$$

where n is a method in a subgraph sg in the database of all call graphs D.

As the combined approach in [13] leads to good results but requires two costly graph-mining executions, the authors have developed a technique in [14] which requires only one graph-mining execution: They combine the frequency-based score with the simple structural score P_{fail}, both based on the results from one *CloseGraph* [33] execution. They combine the results with the arithmetic mean, as before:

$$P_{\text{comb[14]}}(m) = \frac{P_{\text{freq}}(m)}{2 \max\limits_{n \in sg \in D}(P_{\text{freq}}(n))} + \frac{P_{\text{fail}}(m)}{2 \max\limits_{n \in sg \in D}(P_{\text{fail}}(n))}$$

5.4 Comparison

We now present the results of our experimental comparison of the bug localization and reduction techniques introduced in this chapter. The results are based on the (slightly revised) experiments in [13, 14].

Most bug localization techniques as described in this chapter produce ordered lists of methods. Someone doing a code review would start with the first method in such a list. The maximum number of methods to be checked to find the bug therefore is the position of the faulty method in the list. This position is our measure of result accuracy. Under the assumption that all methods have the same size and that the same effort is needed to locate a bug within a method, this measure linearly quantifies the intellectual effort to find a bug. Sometimes two or more subsequent positions have the same score. As the intuition is to count the maximum number of methods to be checked, all positions with the same score have the number of the last position with this score. If the first bug is, say, reported at the third position, this is a fairly good result, depending on the total number of methods. A software developer only has to do a code review of maximally three methods of the target program.

Our experiments feature a well known Java diff tool taken from [8], consisting of 25 methods. We instrumented this program with fourteen different bugs which are artificial, but mimic bugs which occur in reality and are similar to the bugs used in related work. Each version contains one – and in two cases two – bugs. See [14] for more details on these bugs. We have executed each version of the program 100 times with different input data. Then we have classified the executions as correct or failing with a test oracle based on a bug free reference program.

The experiments are designed to answer the following questions:

1 How do *frequency-based approaches* perform compared to *structural* ones? How can *combined approaches* improve the results?

2 In Subsection 4.5 we have compared *reduction techniques* based on the compression ratio achieved. How do the different reduction techniques perform in terms of bug localization precision?

3 Some approaches make use of the *temporal order* of method calls. The call graph representations tend to be much larger than without. Do such graph representations improve precision?

In concrete terms, we compare the following five alternatives:

- $E_{\mathbf{01m}}$**:** The structural P_{SN}-scoring approach similar to [9] (cf. Subsection 5.1), but based on the unordered $R_{\mathrm{01m\_unord}}$ reduction.
- $E_{\mathbf{subtree}}$**:** The frequency-based P_{freq}-scoring approach as in [13, 14] (cf. Subsection 5.2) based on the R_{subtree} reduction.
- $E_{\mathbf{comb[13]}}$**:** The combined approach from [13] (cf. Subsection 5.3) based on the $R_{\mathrm{01m\_unord}}$ and R_{subtree} reductions.
- $E_{\mathbf{comb[14]}}$**:** The combined approach from [14] (cf. Subsection 5.3) based on the R_{subtree} reduction.
- $E_{\mathbf{total}}$**:** The combined approach as in [14] (cf. Subsection 5.3) but with the $R_{\mathrm{total\_w}}$ reduction like in [25] (but with weights and without temporal edges, cf. Subsection 5.1).

We present the results (the number of the first position in which a bug is found) of the five experiments for all fourteen bugs in Table 17.3. We represent a bug which is not discovered with the respective approach with '25', the total number of methods of the program. Note that with the frequency-based and the combined method rankings, there usually is information available where a bug is located within a method, and in the context of which subgraph it appears. The following comparisons leave aside this additional information.

Exp.\Bug	1	2	3	4	5	6	7	8	9	10	11	12	13	14
E_{01m}	25	3	1	3	2	4	3	1	1	6	4	4	25	4
E_{subtree}	3	3	1	1	1	3	3	1	25	2	3	3	3	3
$E_{\mathrm{comb[13]}}$	1	3	1	2	2	1	2	1	3	1	2	4	8	5
$E_{\mathrm{comb[14]}}$	3	2	1	1	1	2	2	1	18	2	2	3	3	3
E_{total}	1	5	1	4	3	5	5	2	25	2	5	4	6	3

Table 17.3. Experimental results.

Structural, Frequency-Based and Combined Approaches. Comparing the results from E_{01m} and E_{subtree}, the frequency-based approach (E_{subtree}) performs almost always as good or better than the structural one (E_{01m}). This demonstrates that analyzing numerical call frequencies is adequate to locate bugs. Bugs 1, 9 and 13 illustrate that both approaches alone cannot find certain bugs. Bug 9 cannot be found by comparing call frequencies (E_{subtree}). This is because Bug 9 is a modified condition which always leads to the invocation of a certain method. In consequence, the call frequency is always the same. Bugs 1 and 13 are not found with the purely structural approach (E_{01m}). Both are typical call frequency affecting bugs: Bug 1 is in an `if`-condition inside a loop and leads to more invocations of a certain method. In Bug 13, a modified `for`-condition slightly changes the call frequency of a method inside the loop. With the $R_{\text{01m\_unord}}$ reduction technique used in E_{01m}, Bug 2 and 13 have the same graph structure both with correct and with failing executions. Thus, it is difficult to impossible to identify structural differences.

The combined approaches in $E_{\text{comb[13]}}$ and $E_{\text{comb[14]}}$ are intended to take structural information into account as well to improve the results from E_{subtree}. We do achieve this goal: When comparing E_{subtree} and $E_{\text{comb[14]}}$, we retain the already good results from E_{subtree} in nine cases and improve them in five.

When looking at the two combination strategies, it is hard to say which one is better. $E_{\text{comb[13]}}$ turns out to be better in four cases while $E_{\text{comb[14]}}$ is better in six ones. Thus, the technique in $E_{\text{comb[14]}}$ is slightly better, but not with every bug. Furthermore, the technique in $E_{\text{comb[13]}}$ is less efficient as it requires two graph-mining runs.

Reduction Techniques. Looking at the call-graph-reduction techniques, the results from the experiments discussed so far reveal that the subtree-reduction technique with edge weights (R_{subtree}) used in E_{subtree} as well as in both combined approaches is superior to the zero-one-many reduction ($R_{\text{01m\_unord}}$). Besides the increased precision of the localization techniques based on the reduction, R_{subtree} also produces smaller graphs than $R_{\text{01m\_unord}}$ (cf. Subsection 4.5).

E_{total} evaluates the total reduction technique. We use $R_{\text{total\_w}}$ as an instance of the total reduction family. The rationale is that this one can be used with $E_{\text{comb[14]}}$. In most cases, the total reduction (E_{total}) performs worse than the subtree reduction ($E_{\text{comb[14]}}$). This confirms that the subtree-reduction technique is reasonable, and that it is worth to keep more structural information than the total reduction does. However, in cases where the subtree reduction produces graphs which are too large for efficient mining, and the total reduction produces sufficiently small graphs, $R_{\text{total\_w}}$ can be an alternative to R_{subtree}.

Temporal Order. The experimental results listed in Table 17.3 do not shed any light on the influence of the temporal order. When applied to the buggy programs used in our comparisons, the total reduction with temporal edges ($R_{\text{total\_tmp}}$) produces graphs of a size which cannot be mined in a reasonable time. This already shows that the representation of the temporal order with additional edges might lead to graphs whose size is not manageable any more. In preliminary experiments of ours, we have repeated E_{01m} with the $R_{\text{01m\_ord}}$ reduction and the *FREQT* [2] rooted ordered tree miner in order to evaluate the usefulness of the temporal order. Although we systematically varied the different mining parameters, the results of these experiments in general are not better than those in E_{01m}. Only in two of the 14 bugs the temporal-aware approach has performed better than E_{01m}, in the other cases it has performed worse. In a comparison with the R_{subtree} reduction and the *gSpan* algorithm [32], the $R_{\text{01m\_ord}}$ reduction with the ordered tree miner displayed a significantly increased runtime by a factor of 4.8 on average.[4] Therefore, our preliminary result is that the incorporation of the temporal order does not increase the precision of bug localizations. This is based on the bugs considered so far, and more comprehensive experiments would be needed for a more reliable statement.

Threats to Validity. The experiments carried out in this subsection, as well as in the respective publications [9, 13, 14, 25], illustrate the ability to locate bugs based on dynamic call graphs using graph mining techniques. From a software engineering point of view, three issues remain for further evaluations: (1) All experiments are based on artificially seeded bugs. Although these bugs mimic typical bugs as they occur in reality, a further investigation with real bugs, e.g., from a real software project, would prove the validity of the proposed techniques. (2) All experiments feature rather small programs containing the bugs. The programs rarely consist of more than one class and represent situations where bugs could be found relatively easy by a manual investigation as well. When solutions for the current scalability issues are found, localization techniques should be validated with larger software projects. (3) None of the techniques considered has been directly compared to other techniques such as those discussed in Section 3. Such a comparison, based on a large number of bugs, would reveal the advantages and disadvantages of the different techniques. The *iBUGS* project [7] provides real bug datasets from large software projects such as *AspectJ*. It might serve as a basis to tackle the issues mentioned.

6. Conclusions and Future Directions

This chapter has dealt with the problem of localizing software bugs, as a use case of graph mining. This localization is important as bugs are hard to detect manually. Graph mining based techniques identify structural patterns in trace data which are typical for failing executions but rare in correct. They serve as hints for bug localization. Respective techniques based on call graph mining first need to solve the subproblem of call graph reduction. In this chapter we have discussed both reduction techniques for dynamic call graphs and approaches analyzing such graphs. Experiments have demonstrated the usefulness of our techniques and have compared different approaches.

[4]In this comparison, *FREQT* was restricted as in [9] to find subtrees of a maximum size of four nodes. Such a restriction was not set in *gSpan*. Furthermore, we expect a further significant speedup when *CloseGraph* [33] is used instead of *gSpan*.

All techniques surveyed in this chapter work well when applied to relatively small software projects. Due to the NP-hard problem of subgraph isomorphism inherent to frequent subgraph mining, none of the techniques presented is directly applicable to large projects. One future challenge is to overcome this problem, be it with more sophisticated graph-mining algorithms, e.g., scalable approximate mining or discriminative techniques, or smarter bug-localization frameworks, e.g., different graph representations or constraint based mining. One starting point could be the granularity of call graphs. So far, call graphs represent method invocations. One can think of smaller graphs representing interactions at a coarser level, i.e., classes or packages. [12] presents encouraging results regarding the localization of bugs based on class-level call graphs. As future research, we will investigate how to turn these results into a scalable framework for locating bugs. Such a framework would first do bug localization on a coarse level before 'zooming in' and investigating more detailed call graphs.

Call graph reduction techniques introducing edge weights trigger another challenge for graph mining: weighted graphs. We have shown that the analysis of such weights is crucial to detect certain bugs. Graph-mining research has focused on structural issues so far, and we are not aware of any algorithm for explicit mining of weighted graphs. Next to reduced call graphs, such algorithms could mine other real world graphs as well [3], e.g., in logistics [19] and image analysis [27].

Acknowledgments

We are indebted to Matthias Huber for his contributions. We further thank Andreas Zeller for fruitful discussions and Valentin Dallmeier for his comments on early versions of this chapter.

References

[1] F. E. Allen. Interprocedural Data Flow Analysis. In *Proc. of the IFIP Congress*, 1974.

[2] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient Substructure Discovery from Large Semi-structured Data. In *Proc. of the 2nd SIAM Int. Conf. on Data Mining (SDM)*, 2002.

[3] D. Chakrabarti and C. Faloutsos. Graph Mining: Laws, Generators, and Algorithms. *ACM Computing Surveys (CSUR)*, 38(1):2, 2006.

[4] R.-Y. Chang, A. Podgurski, and J. Yang. Discovering Neglected Conditions in Software by Mining Dependence Graphs. *IEEE Transactions on Software Engineering*, 34(5):579–596, 2008.

[5] Y. Chi, R. Muntz, S. Nijssen, and J. Kok. Frequent Subtree Mining – An Overview. *Fundamenta Informaticae*, 66(1–2):161–198, 2005.

[6] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight Defect Localization for Java. In *Proc. of the 19th European Conf. on Object-Oriented Programming (ECOOP)*, 2005.

[7] V. Dallmeier and T. Zimmermann. Extraction of Bug Localization Benchmarks from History. In *Proc. of the 22nd IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, 2007.

[8] I. F. Darwin. *Java Cookbook.* O'Reilly, 2004.

[9] G. Di Fatta, S. Leue, and E. Stegantova. Discriminative Pattern Mining in Software Fault Detection. In *Proc. of the 3rd Int. Workshop on Software Quality Assurance (SOQUA)*, 2006.

[10] R. Diestel. *Graph Theory.* Springer, 2006.

[11] T. G. Dietterich, P. Domingos, L. Getoor, S. Muggleton, and P. Tadepalli. Structured Machine Learning: The Next Ten Years. *Machine Learning*, 73(1):3–23, 2008.

[12] F. Eichinger and K. Bөhm. Towards Scalability of Graph-Mining Based Bug Localisation. In *Proc. of the 7th Int. Workshop on Mining and Learning with Graphs (MLG)*, 2009.

[13] F. Eichinger, K. Bөhm, and M. Huber. Improved Software Fault Detection with Graph Mining. In *Proc. of the 6th Int. Workshop on Mining and Learning with Graphs (MLG)*, 2008.

[14] F. Eichinger, K. Bөhm, and M. Huber. Mining Edge-Weighted Call Graphs to Localise Software Bugs. In *Proc. of the European Conf. on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, 2008.

[15] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[16] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[17] S. L. Graham, P. B. Kessler, and M. K. Mckusick. gprof: A Call Graph Execution Profiler. In *Proc. of the ACM SIGPLAN Symposium on Compiler Construction*, 1982.

[18] M. J. Harrold, R. Gupta, and M. L. Soffa. A Methodology for Controlling the Size of a Test Suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3):270–285, 1993.

[19] W. Jiang, J. Vaidya, Z. Balaporia, C. Clifton, and B. Banich. Knowledge Discovery from Transportation Network Data. In *Proc. of the 21st Int. Conf. on Data Engineering (ICDE)*, 2005.

[20] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of Test Information to Assist Fault Localization. In *Proc. of the 24th Int. Conf. on Software Engineering (ICSE)*, 2002.

[21] P. Knab, M. Pinzger, and A. Bernstein. Predicting Defect Densities in Source Code Files with Decision Tree Learners. In *Proc. of the Int. Workshop on Mining Software Repositories (MSR)*, 2006.

[22] B. Korel and J. Laski. Dynamic Program Slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[23] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug Isolation via Remote Program Sampling. *ACM SIGPLAN Notices*, 38(5):141–154, 2003.

[24] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: Statistical Model-Based Bug Localization. *SIGSOFT Software Engineering Notes*, 30(5):286–295, 2005.

[25] C. Liu, X. Yan, H. Yu, J. Han, and P. S. Yu. Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs. In *Proc. of the 5th SIAM Int. Conf. on Data Mining (SDM)*, 2005.

[26] N. Nagappan, T. Ball, and A. Zeller. Mining Metrics to Predict Component Failures. In *Proc. of the 28th Int. Conf. on Software Engineering (ICSE)*, 2006.

[27] S. Nowozin, K. Tsuda, T. Uno, T. Kudo, and G. Bakir. Weighted Substructure Mining for Image Analysis. In *Proc. of the Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2007.

[28] K. J. Ottenstein and L. M. Ottenstein. The Program Dependence Graph in a Software Development Environment. *SIGSOFT Software Engineering Notes*, 9(3):177–184, 1984.

[29] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.

[30] A. Schrөter, T. Zimmermann, and A. Zeller. Predicting Component Failures at Design Time. In *Proc. of the 5th Int. Symposium on Empirical Software Engineering*, 2006.

[31] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann Publishers, 2005.

[32] X. Yan and J. Han. gSpan: Graph-Based Substructure Pattern Mining. In *Proc. of the 2nd IEEE Int. Conf. on Data Mining (ICDM)*, 2002.

[33] X. Yan and J. Han. CloseGraph: Mining Closed Frequent Graph Patterns. In *Proc. of the 9th ACM Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, 2003.

[34] T. Zimmermann, N. Nagappan, and A. Zeller. Predicting Bugs from History. In T. Mens and S. Demeyer, editors, *Software Evolution*, pages 69–88. Springer, 2008.