Software-Defect Localisation by Mining Dataflow-Enabled Call Graphs

Frank Eichinger, Klaus Krogmann, Roland Klug, and Klemens Böhm

Karlsruhe Institute of Technology (KIT), Germany
{eichinger, krogmann, klemens.boehm}@kit.edu, klugr@ipd.uka.de

Abstract. Defect localisation is essential in software engineering and is an important task in domain-specific data mining. Existing techniques building on call-graph mining can localise different kinds of defects. However, these techniques focus on defects that affect the controlflow and are agnostic regarding the dataflow. In this paper, we introduce dataflow-enabled call graphs that incorporate abstractions of the dataflow. Building on these graphs, we present an approach for defect localisation. The creation of the graphs and the defect localisation are essentially data mining problems, making use of discretisation, frequent subgraph mining and feature selection. We demonstrate the defect-localisation qualities of our approach with a study on defects introduced into Weka. As a result, defect localisation now works much better, and a developer has to investigate on average only 1.5 out of 30 methods to fix a defect.

1 Introduction

Software quality is a huge concern in industry and in the software-engineering community. Software is rarely free from defects, and finding them is difficult. Especially when projects are large, several developers make changes in the source code, and a developer works with code somebody else has written, localising defects is tedious. (Semi-)Automatic tools for defect localisation are desirable. Clearly, such tools should be able to deal with many different kinds of defects.

In the past years, a number of studies has investigated defect localisation with *graph-mining techniques* [3, 5, 7, 8, 18]. They build on the analysis of *dynamic call graphs* (see [6] for an overview). One such graph is a concise representation of a programme execution and reflects the method-invocation structure. The localisation techniques do frequent subgraph mining with call graphs of correct and of failing executions. The rationale for defect localisation is that patterns occurring more frequently in graphs of failing executions contain methods which are more likely to be defective. Techniques that incorporate the analysis of call frequencies have proven to be more accurate and discover more types of defects than techniques without this feature [7]. See Figure 1 for a simple call graph representing a programme execution – each node stands for a method, edges represent method calls, and edge weights represent method-call frequencies.

An important characteristic of the existing call-graph-based techniques is that they merely analyse the call-graph structure and the call frequencies. They

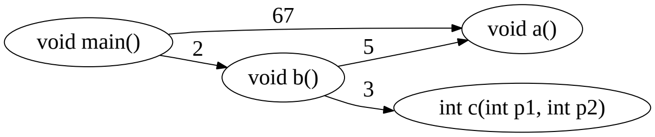

Fig. 1. Example call graph with call frequencies (not dataflow enabled).

can only localise defects which affect the call graph of a programme execution (simplified, the controlflow). While this is an important class of defects, Cheng et al. [3] point out that the current techniques are agnostic regarding defects that influence the dataflow. We refer to such defects as *dataflow-affecting bugs*. They influence the data exchanged between methods. For example, think of a method which wrongly calculates some value, and which needs to be localised. A call-graph-based technique can only recognise such a defect if the value affects a control statement. Although this happens frequently, it might occur in methods which are actually defect-free, leading to erroneous localisations.

Han and Gao identify software engineering and defect localisation as a major area in domain-specific data mining [9]. They point out that the integration of domain knowledge (in our case, the exact specification of call graphs) and dedicated analysis techniques are crucial for the success of data mining in any domain. In this paper, we present a call-graph-based technique which localises both dataflow-affecting and call-graph-affecting bugs. The specification of the underlying graphs is not obvious: On the one hand, a call graph is a compact representation of an execution. On the other hand, dataflow-related information refers to values of many method calls within one execution. This information needs to be available at a level of detail which allows to locate defects. To illustrate the difficulties, an edge in a call graph typically represents thousands to millions of method calls. Annotating each edge with the method-call parameters and method-return values of all invocations corresponding to it incurs huge annotations and is not practical. In this paper we propose *dataflow-enabled call graphs* (*DEC graphs*) which incorporate concise numeric dataflow information.

DEC graphs are augmentations of call graphs with abstractions of method-call parameters and of method-return values. To obtain DEC graphs, we treat different data types differently. In particular, we discretise numerical parameter and return values. Figure 2 is a DEC graph corresponding to Figure 1. The call from method `b` to method `c` is attributed with a tuple of integers, containing the total number of calls and the numbers of calls with parameter and return values falling into different intervals. (We explain the details later.) When the DEC graphs are assembled, we do frequent subgraph mining with the graphs, not considering the dataflow abstractions for the moment. We then analyse the tuples of integers assigned to the edges with a feature-selection algorithm in the different subgraphs mined separately. Finally, we derive a likelihood of defectiveness for every method in the programme considered. These likelihoods form a ranking of suspicious methods which can guide the manual debugging process. We demonstrate the appropriateness and precision of our DEC-graph-based ap-

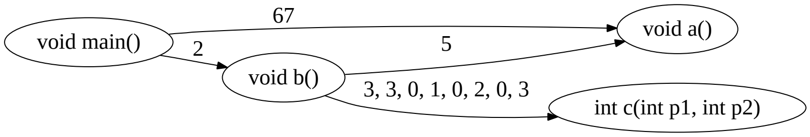

Fig. 2. Example dataflow-enabled call graph (DEC graph).

proach for the localisation of defects. In a case study we evaluate the approach using defects introduced into the Weka machine-learning suite [23].

All in all, our new technique for defect localisation features contributions at different stages of the analysis process and in the application domain:

Dataflow-Enabled Call Graphs. We introduce *DEC graphs* as sketched before, featuring dataflow abstractions. We describe an efficient implementation of their generation for Java programmes. To the best of our knowledge, this is the first study considering the dataflow within call graphs for defect localisation.

A Defect-Localisation Approach for Dataflow-Affecting Bugs. We present a defect localisation technique for DEC graphs. It is a case of weighted graph mining, which ultimately identifies defective methods.

Results in Software Engineering. We demonstrate the usefulness of our approach by means of defect-localisation experiments. Localisation works better with DEC graphs, as compared to graphs that are not dataflow enabled. Some defects can only be localised well with DEC graphs. Further, we describe and evaluate extensions of our approach which improve the defect localisation.

Paper Outline: Section 2 introduces the fundamentals of call-graph-based defect localisation. Sections 3 and 4 introduce DEC graphs and explain how we use them for defect localisation. Section 5 contains the experimental evaluation, Section 6 discusses related work, and Section 7 concludes.

2 Fundamentals of Call-Graph-Based Defect Localisation

In this section we first introduce our notion of bugs, followed by fundamentals on call-graph-based defect localisation.

Types of Bugs in Software. In the field of debugging, one usually avoids the terms *bug* and *fault*, but distinguishes between *defects*, *infections* and *failures* [26]. In this frequently-cited classification, *defects* are the places in the source code which cause a problem, an *infection* is an incorrect programme state (usually triggered by a defect), and *failures* are an observable incorrect programme behaviour (e.g., a user experiences wrong calculations). In the context of our study, we use a further differentiation, introduced in [6]:

- *Crashing bugs* lead to an unexpected termination of a programme. Examples include null-pointer exceptions and divisions by zero, which in various programming languages are easy to localise with the help of a stack trace. *Non-crashing bugs* in turn display failing behaviour but do not provide any

hints. Non-crashing bugs are hard to localise and thus in the focus of graph-mining-based defect-localisation approaches.
- *Occasional bugs* are failures whose occurrence depends on the input data of a programme. Compared to *non-occasional bugs*, occasional bugs are harder to find since they require more test cases to be reproduced.
- *Structure-affecting bugs* are infections which change the *structure* of a call graph. This is, when comparing graphs from correct and failing executions, certain graph substructures might or might not occur in either of the two variants. Such infections typically occur when `if` statements have defects or contain infected variables. In contrast, *call-frequency-affecting bugs* are infections which change the call *frequency* of a certain substructure in failing executions, rather than completely missing or adding structures. Such infections typically occur when loops are involved. Both structure-affecting and call-frequency-affecting bugs are also called *call-graph-affecting bugs.*

As any call-graph-based technique, we focus on *non-crashing* and *occasional bugs.* Opposed to other techniques, we consider *dataflow-affecting bugs* besides *call-graph-affecting bugs. Dataflow-affecting bugs* manifest themself by infected method-call parameters or return values, i.e., a method returns a wrong value. Dataflow-affecting bugs might affect the structure of a call graph and/or the call frequencies as a side effect. In this case, existing techniques can locate the defects in principle. However, infected behaviour often appears in other methods then those with the actual defect. This might disturb defect localisation. In such cases, our dataflow-enabled technique can deliver more precise localisations.

Localising Call-Graph-Affecting Bugs. In the past years, a number of call-graph-based techniques for defect localisation have been proposed [3, 5, 7, 8, 18]. Their basic idea is to mine for patterns in the call graph which are characteristic for failing executions. Then, they derive some defectiveness likelihood for the methods. We now briefly review the different call graph variants used, as well as the corresponding defect-localisation techniques. [6] is a detailed survey.

Existing techniques focus on structure-affecting bugs [3, 5, 7, 18] and call-frequency-affecting bugs [7, 8]. The graphs in [3, 5, 18] incorporate temporal information, the ones in [7, 8] do not. In [6, 7] we explain that the increased graph size when temporal information is incorporated leads to scalability issues when it comes to the mining. [3, 8, 18] build on call graphs where exactly one node represents a method, while [5, 7] allow for more than one node. This promises more precise results, as more detailed contexts of method calls can be included in the analysis. At the same time, the size of the graphs increases, which also tends to increase runtime [6]. In contrast to the other representations, the graphs in [7, 8] are weighted. Edge weights represent the number of corresponding method calls.

Besides different graph representations, the different approaches derive defectiveness likelihoods in different ways. [18] builds on graph classification with subgraph patterns. The authors first mine frequent subgraph patterns with a variant of CloseGraph [25] before they use them to train a support-vector machine (SVM). The authors consider the difference in accuracy between two SVMs – one built with graph patterns including a certain method and one without them

– as an evidence for defective methods. [3] builds on the same graphs, but relies on discriminative subgraph mining with the LEAP algorithm [24]. This directly pinpoints suspicious subgraph-structures and thus methods that are possibly defective. [5] derives defect likelihoods from support values of subgraph patterns in call graphs. Finally, [7] combines the idea from [5] to use support-based structural likelihoods with the analysis of call frequencies. While [5] incorporates only basic frequency-related information, [7, 8] analyse call frequencies by means of a feature-selection algorithm. It does so in a step subsequent to graph mining with the CloseGraph algorithm [25]. This analysis allows to localise call-frequency-affecting and structure-affecting bugs. In [8] we successfully investigate the usage of call graphs for the localisation of defects in multithreaded programmes.

In this paper, we borrow concepts from both graph representations and localisation techniques from previous work. However, our graphs and our technique are different as they are tailored for the localisation of dataflow-affecting bugs – which is new in this study – in addition to call-graph-affecting ones. The shape of our graphs is similar to those in [3, 8, 18], but without temporal information. Furthermore, we generalise the concept of edge weights and their analysis [7, 8] by introducing tuples of weights incorporating dataflow abstractions.

3 Dataflow-Enabled Call Graphs (DEC Graphs)

In this section we introduce and specify *dataflow-enabled call graphs* (*DEC graphs*) and explain how we obtain them. These graphs and their analysis (described in the following section) are the core of our approach to localise dataflow-affecting bugs. The basic idea of DEC graphs is to extend edges in call graphs with tuples which are abstractions of method parameters and return values. Obtaining these abstractions is a data-mining problem by itself: Huge amounts of values from method-call monitoring need to be condensed to enable a later analysis and ultimately the localisation of defects. We address this problem by means of discretisation. In the following, we first explain how we derive programme traces from programme executions. Next, we describe the structure of our call graphs. We then explain the dataflow abstractions and explain why they are useful for defect localisation. Finally, we say how we obtain the graphs from programme traces and give a concrete example.

Derivation of Programme Traces. We employ the aspect-oriented programming language AspectJ [13] to weave tracing functionality into Java programmes. By defining so-called pointcuts in AspectJ, we instrument method calls. At each call, we insert logging statements which save caller-callee relations. For each invocation, we log call frequency and data values (parameters and return values) that occur at runtime. Finally, we use this data to build call graphs.

When logging data values, we log primitive data types as they are, capture arrays and collections by their size, and reduce strings to their length. Such an abstraction from concrete dataflow has before successfully been used in the area of software performance prediction, e.g. [14]. Certainly, these simplifications can be severe, but logging the full data would result in overly large amounts of data.

Our evaluation (Section 5) primarily studies primitive data types. A systematic evaluation of arrays, collections and strings as well as techniques for complex data types is subject to future work.

Call-Graph Structure. Based on the experience from previous studies [6], we decide to make use of a *total-reduction* variant of call graphs. This allows for better scalability of mining algorithms for large software projects and for an on-the-fly generation. In such graphs, every individual method which is called during a programme execution forms a single node (see Figure 1 for an example).

Dataflow Abstractions. As mentioned before, we use discretisation in order to find an abstraction of method parameters and return values based on the values monitored. Discretisation gives us a number of intervals for every parameter and for the return value (we discuss respective techniques in the following). We then count the number of method invocations falling into the intervals determined and attribute these counts to the edges.

Definition 1. *An* edge-weight tuple *in a* dataflow-enabled call graph (DEC graph) *consists of the counts of method calls falling into the respective intervals:*

$$(t, p_1^{i_1}, p_1^{i_2}, ..., p_1^{i_{n_1}}, p_2^{i_1}, p_2^{i_2}, ..., p_2^{i_{n_2}}, ..., p_m^{i_1}, p_m^{i_2}, ..., p_m^{i_{n_m}}, r^{i_1}, r^{i_2}, ..., r^{i_{n_r}})$$

where t is the total number of calls, $p_1, p_2, ..., p_m$ are the method-call parameters, r is the method-return value and $i_1, i_2, ..., i_{n_x}$ (n_x denotes the number of intervals of parameter/return value x) are the intervals of the parameters/return values.

The idea is that values referring to an infection tend to fall into different intervals than values which are not infected. For example, infected values might always be lower than correct values. Alternatively, infected values might be outliers which do not fall into the intervals of correct values as well. In order to be suited for defect localisation, intervals must respect correct and failing programme executions as well as distributions of values. Generally, it might be counter-productive to divide a value range like integer into intervals of equal size. Groups of close-by values of the same class might fall into different intervals, which would complicate defect localisation.

Derivation of Dataflow-Enabled Call Graphs (DEC Graphs). The CAIM (class-attribute interdependence maximisation) algorithm [15] suits our requirements for intelligent discretisation: It (1) discretises single numerical attributes, (2) takes classes associated with tuples into account (i.e., *correct* and *failing* executions in our scenario) and (3) automatically determines a (possibly) minimal number of intervals. Internally, the algorithm maximises the attribute-class interdependence. Comparative experiments by the CAIM inventors have demonstrated a high accuracy in classification settings.

In concrete terms, we let CAIM find intervals for every method parameter and return value of every method call corresponding to a certain edge. We do so for all edges in all call graphs belonging to the programme executions considered. We then assemble the edge-weight tuples as described in Definition 1. Example 1 illustrates the discretisation. As we are faced with millions of method calls from hundreds to thousands of programme executions, frequently consisting of duplicate values, we pre-aggregate values during the execution. To avoid scalability problems, we then utilise a proprietary implementation of CAIM which is able to handle large amounts of data in pre-aggregated form. Note that the dataflow abstractions in DEC graphs can only be derived for a set of executions, as discretisation for a single execution is not meaningful.

Example 1. We consider the call of method `c` from method `b` in Figure 1 (Exec. 1 in Table 1) and three further programme executions (Exec. 2–4) invoking the same method with a frequency of one to three. Method `c` has two parameters `p1`, `p2` and returns value `r`. A discretisation of `p1`, `p2` and `r` based on the example values given in Table 1(a) leads to two intervals of `p1` and `r` ($p_1^{\mathrm{i}_1}, p_1^{\mathrm{i}_2}$ and $r^{\mathrm{i}_1}, r^{\mathrm{i}_2}$) and three for `p2` ($p_2^{\mathrm{i}_1}, p_2^{\mathrm{i}_2}, p_2^{\mathrm{i}_3}$). See Table 1(b) for the exact intervals. The occurrences of elements of edge-weight tuples can then be counted easily – see Table 1(c), the discretised version of Table 1(a). The edge-weight tuple of `b` → `c` in Exec. 1 then is as displayed in Figure 2, referring to $(t, p_1^{\mathrm{i}_1}, p_1^{\mathrm{i}_2}, p_2^{\mathrm{i}_1}, p_2^{\mathrm{i}_2}, p_2^{\mathrm{i}_3}, r^{\mathrm{i}_1}, r^{\mathrm{i}_2})$.

(a) Example call data.

Exec.	`p1`	`p2`	`r`	*class*
1	2	43	12	*correct*
1	1	44	11	*correct*
1	3	4	9	*correct*
2	12	33	8	*failing*
3	23	27	6	*failing*
3	15	28	5	*failing*
3	16	23	7	*failing*
4	6	2	10	*correct*
4	11	47	13	*correct*

(b) Intervals generated.

Value	Intervals
`p1`	$i_1 : [1, 11.5]$
	$i_2 : (11.5, 23]$
`p2`	$i_1 : [2, 13.5]$
	$i_2 : (13.5, 38]$
	$i_3 : (38, 47]$
`r`	$i_1 : [5, 8.5]$
	$i_2 : (8.5, 13]$

(c) Discretised data.

Exec.	`p1`	`p2`	`r`
1	i_1	i_3	i_2
1	i_1	i_3	i_2
1	i_1	i_1	i_2
2	i_2	i_2	i_1
3	i_2	i_2	i_1
3	i_2	i_2	i_1
3	i_2	i_2	i_1
4	i_1	i_1	i_2
4	i_1	i_3	i_2

Table 1. Example discretisation for the call of `int c(int p1, int p2)` from `b`.

4 Localising Dataflow-Affecting Bugs

We now explain how to derive defect localisations from DEC graphs. We first give an overview, then describe subgraph mining (Section 4.1) and the actual defect localisation (Section 4.2) and two extensions (Sections 4.3 and 4.4).

Overview. Algorithm 1 works with a set of traces T of programme executions. At first, it assigns a class (*correct*, *failing*) to every trace $t \in T$ (Line 3), using a test oracle. Then the procedure generates DEC graphs from every trace t (Line 4). Next, the procedure derives frequent subgraphs of these graphs which are used as contexts where defects are located (Line 6). The last step calculates a likelihood of containing a defect for every method m (Line 7). This facilitates a ranking of the methods, which can be given to software developers. They would then review the suspicious methods manually, starting with the one which is most likely to be defective.

Algorithm 1 Procedure of defect localisation with DEC graphs.

```
Input: a set of programme traces $t \in T$
Output: a method ranking based on each method's likelihood to be defective $P(m)$
1: $G = \emptyset$ // initialise a set of DEC graphs
2: for all traces $t \in T$ do
3:     check if $t$ was a correct execution and assign a $class \in \{correct, failing\}$ to $t$
4:     $G = G \cup \{derive\_dataflow\text{-}enabled\_call\_graph(t)\}$
5: end for
6: $SG = frequent\_subgraph\_mining(G)$
7: calculate $P(m)$ for all methods $m$; based on $SG$
```

4.1 Frequent Subgraph Mining

As shown in Line 6 in Algorithm 1, we use frequent subgraph mining to derive subgraphs which are frequent within the call graphs considered. This particular step mines the pure graph structure only and ignores the edge-weight-tuples for the moment. The subgraphs obtained serve as different contexts, and further analyses are done for every subgraph context separately. This aims at a higher precision than an analysis without such contexts. For example, a failure might occur when method a is called from method b, only when method c is called as well. Then, the defect might be localised only in call graphs containing all methods mentioned, but not in graphs without method c.

We rely on the ParSeMiS implementation [21] of CloseGraph [25] for frequent subgraph mining. CloseGraph has successfully been used in related studies [7, 18]. In a set of graphs G, it discovers subgraphs with a user-defined minimum support. For this value, we use $\min(|G_{\text{corr}}|, |G_{\text{fail}}|)/2$, where G_{corr} and G_{fail} are the sets of call graphs of correct and failing executions, respectively ($G = G_{\text{corr}} \cup G_{\text{fail}}$). This ensures not to miss any structure which occurs in less then half of all executions belonging to the smaller class. Preliminary experiments have shown that this minimum support allows for both short runtimes and good results.

4.2 Entropy-Based Defect Localisation

Next, we calculate the likelihood that a method contains a defect (Line 7 in Algorithm 1). The rationale is to identify methods which call other methods with discriminative parameter values or which have return values discriminating well between correct and failing executions. As mentioned before, we analyse every edge-weight tuple in the DEC graphs in the context of every subgraph mined. This aims at a high probability to actually reveal a defect, as every tuple is typically investigated in many different contexts. We assemble a table which contains the elements of the tuples of all edges in all subgraphs as columns and all programme executions (represented by their DEC graphs) as rows. The table cells contain the tuple values: the total call frequencies t and normalised interval frequencies. More precisely, we divide every interval frequency by the corresponding t in order to obtain the ratio of calls falling into each interval.

Exec.	sg_1									sg_2		*class*
	`main` → `b`	`b` → `c`								`main` → `a`	$\cdots$	
	t	t	$\frac{p_1^{\mathrm{i}1}}{t}$	$\frac{p_1^{\mathrm{i}2}}{t}$	$\frac{p_2^{\mathrm{i}1}}{t}$	$\frac{p_2^{\mathrm{i}2}}{t}$	$\frac{p_2^{\mathrm{i}3}}{t}$	$\frac{r^{\mathrm{i}1}}{t}$	$\frac{r^{\mathrm{i}2}}{t}$	t		
g_1	2	3	1.00	0.00	0.33	0.00	0.67	0.00	1.00	67	$\cdots$	*correct*
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
g_n	2	9	1.00	0.00	0.33	0.00	0.67	0.67	0.33	0	$\cdots$	*failing*

Table 2. Example feature table. g_1 refers to Exec. 1 from Example 1 (Figure 2).

Table 2 is an example table which assumes that two subgraphs were found in the previous graph mining step, sg_1 (`main` → `b` → `c`) and sg_2 (`main` → `a`). The first column lists the call graphs $g \in G$. The second column corresponds to *sg1* and edge `main` → `b` with the total call frequency t. The following eight columns correspond to the second edge in this subgraph. Besides the total call frequency t, these columns represent intervals and are derived from the frequencies of parameter and return values. The very 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_n, which does not contain sg_2.

After assembling the table, we employ the *information-gain-ratio feature-selection* algorithm (*GainRatio*, [22]) in its Weka implementation [23] to calculate the discriminativeness of the columns and thus of the different edge-weight-tuple values. The *GainRatio* is a measure from information theory and builds – similar to information gain – on *entropy*. Values of *GainRatio* are in the interval $[0, 1]$. High values indicate a table column affected by a defect. Previous work [7, 8] has shown that entropy-based measures are well-suited for defect localisation.

So far, we have derived defect likelihoods for every column in the table. However, we are interested in likelihoods for methods m, and every method corresponds to more than one column in general. This is due to the fact that a method can call several other methods and might itself be invoked from various other methods, in the context of different subgraphs. Furthermore, methods might have several parameters and a return value, each with possibly several intervals. To obtain method likelihood $P(m)$, we assign every column containing a total frequency t or a parameter-interval frequency p^{i} to the calling method and every return-value-interval frequency r^{i} to the callee method. We then calculate $P(m)$ as the maximum of the *GainRatio* values of the columns assigned to method m. By doing so, we identify the defect likelihood of a method by its most suspicious invocation and the most suspicious element of its tuple. Other invocations are less important, as they might not be related to a defect. The call context of a likely defective method and suspicious data values are supplementary information which we report to software developers to ease debugging.

Example 2 illustrates how our technique is able to localise *dataflow-affecting bugs* based on the ratios of executions falling into the different intervals of the method parameters and return values. Furthermore, it localises *call-frequency-affecting bugs* based on the call frequencies in the edge-weight tuples. In addition, our technique is able to localise most *structure-affecting bugs* as well: (1) The call structure is implicitly contained in the feature tables (e.g., Table 2) – value 0 indicates subgraphs not supported by an execution. (2) Such defects are frequently caused by control statements (e.g., `if`) evaluating previously wrongly calculated values. Our analysis based on dataflow can detect such situations more directly.

Example 2. The graphs g_1 and g_n in Table 2 display very similar values, but refer to a correct and a failing execution. Assume that method `c` contains a defect which occasionally leads to a wrongly calculated return value. This is reflected in the columns $\frac{r^{\mathrm{i}_1}}{t}$ and $\frac{r^{\mathrm{i}_2}}{t}$ of $\mathtt{b} \to \mathtt{c}$ in sg_1. The *GainRatio* measure will recognise fluctuating values in these columns, leading to a high ranking of method `c`.

4.3 Follow-Up-Infection Detection

Call graphs of failing executions frequently contain infection-like patterns which are caused by a preceding infection (not a defect). We call such patterns *follow-up infections*. We now describe an extension (for Line 7 in Algorithm 1) which detects certain follow-up infections and enhances the method ranking.

A follow-up infection occurs when a defective method m_1 calls another method m_2 which in turn calls method m_3. m_1 can often be localised because of the call frequency associated with edge $m_1 \to m_2$. However, $m_2 \to m_3$ might have a call frequency proportional to $m_1 \to m_2$. Thus, m_2 will have the same *GainRatio* as m_1. We make use of this observation and remove methods within the same subgraph belonging to $m_2 \to m_3$ from the ranking when the following conditions hold: (1) $\mathit{GainRatio}(m_1 \to m_2) = \mathit{GainRatio}(m_2 \to m_3)$ (we consider the *GainRatio* values from columns belonging to total call frequencies and parameters), and (2) $m_1 \to m_2 \to m_3$ is not part of a cycle within any $g \in G$. (2) is necessary as the origin of an infection cannot be determined within a cycle. However, our detection is a heuristic which is helpful in practice (see Section 5). In the presence of noise, this follow-up-infection detection might not work, and – pathologically – two edges might have the same *GainRatio* value by chance.

4.4 Improvements for Structure-Affecting Bugs

The subgraphs mined in Line 6 in Algorithm 1 can be used for an enhanced localisation of structure-affecting bugs. There are two kinds of such bugs: (1) those which lead to additional structures and (2) those leading to missing structures. To deal with both of them, we use the support *supp* of every subgraph *sg* in G_{corr} and G_{fail} separately to define two intermediate rankings. The rationale is that methods in subgraphs having a high support in either correct or failing executions are more likely to be defective. We again use the maximum:

$$P_{\mathrm{corr}}(m) := \max_{m \in sg \in SG} \mathit{supp}(sg, G_{\mathrm{corr}}); \;\; P_{\mathrm{fail}}(m) := \max_{m \in sg \in SG} \mathit{supp}(sg, G_{\mathrm{fail}})$$

With these two values, we define a structural score as follows:

$$P_{\mathrm{struct}}(m) = |P_{\mathrm{corr}}(m) - P_{\mathrm{fail}}(m)|$$

To integrate P_{struct} into our *GainRatio*-based method ranking $P(m)$ (in Line 7 in Algorithm 1), we calculate the average:

$$P_{\text{comb}}(m) = \frac{P(m) + P_{\text{struct}}(m)}{2}$$

5 Experimental Evaluation

To investigate the defect-localisation capabilities of our approach, we use the Weka [23] machine-learning suite, manually add a number of defects to it, instrument the code and execute it using test-input data. Finally, we compare the defect ranking returned by our approach with the de-facto defect locations. Overall, we carry out six experiments[1]:

(E1) Application of the new approach featuring DEC graphs,
(E2) —— with follow-up-infection detection,
(E3) —— with follow-up-infection detection and structural ranking,
(E4) the same approach with call graphs that are not dataflow enabled,
(E5) —— with follow-up-infection detection, and
(E6) —— with follow-up-infection detection and structural ranking.

Experimental Setting. Weka is a data-intensive open-source application with a total of 19,938 methods and 301k lines of code (LOC). We introduce five different kinds of defects. They are of the same types as the defects in related evaluations, e.g., the *Siemens programmes* [10], which are often used [3, 5, 18] to evaluate defect localisation techniques for C programmes. Yet, a single Siemens programme comprises at most 566 LOC, which makes them unrealistically small and makes defect localisation less challenging.

The defect types introduced to Weka are typical programming mistakes, are non-crashing, occasional and dataflow-affecting and/or call-graph-affecting:

- *Variable assignment.* The assigned value of a variable differs from the correct value. An example for such a defect is `counter = a + b` where the correct code is `counter = a`.
- *Off-by-one.* This defect often happens when accessing an array or a collection. For example, `coll.get(i)` is accessed instead of `coll.get(i + 1)`.
- *Return value.* In this case, only the return statement of a method has a defect. For example, `return 0` is used instead of `return bestValue`.
- *Loop iterations.* This kind of defect affects the number of executions of a loop. For example, a `for` loop uses the wrong counter variable or misses an iteration: `for(int i = 0; i < max; i++)` instead of `for(int i = 0; i <= max; i++)`.
- *Branching condition.* This kind of defect covers wrong comparisons like `a > b` instead of `a < b`. Furthermore, the Boolean expressions `and`, `or`, `true` and `false` can be easily misplaced in branching conditions.

[1] (E4–6) essentially are a comparison to [7] (“total reduction”). We use the same localisation technique as with the DEC graphs for a fair comparison.

In total, we evaluate ten separate defects (Defect 1–10) as well as six combinations of two of these defects (Defects 11–16).[2] The defects introduced are 4x variable assignment, 3x return value, 1x off-by-one, 1x loop condition and 1x branch condition. Variable-assignment defects include array manipulations, string operations and inline variable assignments (e.g., `doSth(op(a) + b)`). Return values are manipulated by a value offset, returning a constant instead of a variable and by returning a wrong constant. We introduce some kinds of defects repeatedly to cover different characteristics of each defect. Defects 11–16 mimic typical situations where a programme contains more than one defect.

We introduce all defects in `weka.classifiers.trees.DecisionStump`. This class is the implementation of a decision-tree algorithm which comprises 18 methods. We emphasise that we instrument all 19,938 methods of Weka, and all of them are potential subjects to defect locations. A typical execution of `DecisionStump` involves a total of 30 methods.

We execute each defective version of Weka with 90 sets of sampled data from the UCI machine-learning repository [1] and classify correct and failing executions of the programme. To this end, we first execute a correct reference version of Weka with all 90 UCI data sets. After that, we execute the defective versions with the same data. We then interpret any deviation in the output of the two versions as a failure. The number of correct executions is in the same range as the number of failing ones. They differ by a factor of 2.7 on average and by 5.3 in the worst case.

Experimental Results. We present the results − the ranking position which pinpoints the actual defect − of the six experiments for all sixteen defects in Table 3. This position quantifies the number of methods a software developer has to review in order to find the defect (smaller numbers are preferred). We compare the experimental results pairwise between DEC graphs (E1–3) and non-DEC graphs (E4–6), as indicated by the arcs. A grey-coloured cell means worse results, non-coloured cells mean same or improved results. Bold-face rankings indicate same or improved results compared to the preceding row (separately for DEC/non-DEC graphs). In programmes with more than one defect (i.e., Defects 11–16), we present numbers corresponding to the defect ranked best. This reflects that a developer would first fix one defect, before applying our technique again. Sometimes two or more methods have the same defect likelihood. In this case, we use the worst ranking position for all methods with the same likelihood. This is in line with the methodology of related studies [11].

The experiments clearly show the improved defect-localisation capabilities of the new approach based on DEC graphs. Even without extensions (E1), a top ranking is obtained in 15 out of 16 cases. We consider a method ranked top when a developer has to investigate only 3 methods out of the 30 ones actually executed. With non-DEG graphs (E4), only 6 defects are ranked top. In only 5 out of 48 measurement points, compared to 26 out of 48 ones, the DEC-graph-based approach is worse than the reference. DEC graphs have reached a top

[2] We provide the defective programme versions online: `http://www.ipd.kit.edu/~eichi/papers/eichinger10software-defect/`

Experiment \ Defect	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	∅
(E1) DEC graphs	3	3	1	3	2	2	12	3	1	1	2	2	1	2	1	3	2.3
(E2) DEC graphs	**2**	**2**	**1**	**2**	**2**	**2**	**9**	**2**	**1**	**1**	**2**	**2**	**1**	**2**	**1**	**2**	**1.9**
(E3) DEC graphs	**1**	**1**	**1**	**2**	6	**1**	**1**	**1**	3	5	**1**	**1**	**1**	**1**	**1**	**1**	**1.6**
(E4) Non-DEC graphs	1	1	11	13	10	3	13	10	9	6	3	8	1	3	8	10	6.1
(E5) Non-DEC graphs	**1**	**1**	**4**	**5**	**4**	**2**	**7**	**3**	**5**	**4**	**2**	**4**	**1**	**2**	**3**	**3**	**2.8**
(E6) Non-DEC graphs	**1**	**1**	**1**	**2**	7	**1**	**2**	**1**	8	6	**1**	**1**	**1**	**1**	**1**	**1**	**2.0**

Table 3. Defect-localisation results. (E2/3/5/6) incl. follow-up, (E3/6) incl. struct.

ranking in 44 cases, whereas non-DEC graphs had a top ranking in only 28 cases. When directly comparing DEC graphs (E1) with non-DEC graphs (E4) without extensions, the defect localisation was better in 13 out of 16 cases. Furthermore, looking at the average values ('∅'), the number of methods to be investigated could be reduced by more than half.

Using the follow-up detection (E2/5), the ranking could be improved in all cases or has generated results of the same quality compared to the respective initial approach. This is remarkable, as the follow-up-infection detection is a heuristic approach. The use of both the follow-up and structural extension (E3/6) results in further improvements. For DEC graphs (E3) in comparison to (E2), the extension improves the ranking in 9 cases and lowers the ranking in 3 cases, i.e., better overall results. For non-DEC graphs (E6) in comparison to (E5), the picture is similar: 10 improved cases and 3 worse ones.

Analysing the localisation capabilities per defect class results in an inhomogeneous picture when looking at (E1–3). For the variable-assignment Defects 1, 3, 5, 6, localisation is mostly fine; the off-by-one Defect 2 is well located; return-value Defects 4, 8, 9 can be located with both approaches. Only for Defect 9 we achieve a further improvement compared to non-DEC graphs. The structural extension is misleading for the localisation of branch-condition Defect 10, while it enables the identification of the loop-iteration Defect 7 (a structure-affecting bug; the changed loop condition hinders the loop from being executed and thus other methods from being called).

Regarding the Weka versions with two defects (Defects 11–16), defect localisation always works better on average than for versions with only one defect (E1–6). Our explanation is that defect localisation has a higher chance to be correct when two methods have a defect.

Overall, the experiments show a large improvement of the ranking with the new approach. In combination with follow-up detections and the structural ranking (E3), results are best. Using the structural ranking leads to a slightly worse ranking for some defects. The experiments also show that only 1.6 out of the 19,938 methods of Weka (of which 30 methods are actually executed) must be investigated on average in order to find a defect (E3). The results promise a strong reduction of time spent on defect localisation in software-engineering projects.

Improved Experimental Results using Static Analysis. Despite the good results achieved so far, we investigate further improvements. One starting point is the handling of methods with the same defect likelihood. As in related studies [11], we currently use the worst ranking position for all methods which have the same defect likelihood. A second static ranking criterion helps distinguishing methods with the same defect likelihood: We sort such methods decreasingly by their size in lines of code (LOC)[3]. Research has shown that the size in LOC frequently correlates with the defectiveness likelihood [20]. Applied to our experiments, we can observe an improvement of the average ranking position as follows: 2.3 to 1.9 (E1), 1.9 to 1.7 (E2), 1.6 to 1.5 (E3), 6.1 to 3.6 (E4), 2.8 to 2.6 (E5) and 2.0 to 1.9 (E6). Although the additional static ranking criterion leads to improvements in all experiments, the non-DEC graphs (E4–6) benefit from the improved ranking to a larger extent. As feature selection for non-DEC graphs considers fewer columns, the defect likelihood of methods has fewer different values than for DEC graphs, and this more frequently leads to equal rankings. However, even after the combination with static analysis, defect localisation with DEC graphs is always better on average than with non-DEC graphs. The same observations as described in the preceding paragraphs hold.

6 Related Work

Defect-localisation techniques are *static* or *dynamic*. Dynamic techniques rely on the analysis of programme runs (like our approach) while static techniques do not require any execution and investigate the source code only.

Static Analysis. *Mining software repositories* maps post-release failures from a bug database to defects in static-source code. For example, [20] derives standard code metrics and builds regression models which then predict possible post-release failures. Such approaches require a large collection of defects and extensive version-history data.

FindBugs [2] is an approach complementary to ours. Its static-code analyses for Java generally cannot localise most dataflow-related defects. Regarding the defects in our evaluation (Section 5), FindBugs recognises none of them. Instead, it aims at defects like possible null-pointer accesses due to missing initialisation.

Dynamic Analysis. Approaches in this category are based on instrumentation, like our approach. Such approaches tend to either have a large memory footprint or do not capture all defects due to selective logging of executions.

Statistical defect localisation is a family of dynamic techniques which utilise pattern detection on data values monitored during execution. Liblit et al. [16] analyse monitored data values using regression techniques to identify defective code. Compared to our work, Liblit et al. record only three intervals for return values of methods. We use a variable number of dynamically identified intervals for data characterisations. The approach reduces its footprint by collecting only small samples of executed programmes. A similar approach by Liu et al. [17] focuses on controlflow and instruments variables in condition statements. It then calculates a ranking which yields high values when the evaluation of these state-

[3] Here we use the sum of non-blank and non-comment LOC inside method bodies.

ments differs significantly in correct and failing executions. Opposed to our approach, none of these approaches takes structural properties of call graphs into account. Hence, structure-affecting bugs can be detected less easily.

Analysis of execution traces is the basis for call-graph-based methods. Tarantula [11, 12] is a technique using tracing and visualisation. To localise defects, it utilises a ranking of programme components which are executed more often in failing programme executions. Though this technique is rather simple, it produces good defect-localisation results. Our technique comprises the method-invocation structure and dataflow information, which is not covered by [11, 12].

Masri [19] performs a dynamic information-flow analysis to localise defects in source code. Specifically, sub-paths of information flow of correct and failing executions are compared, to rank defect positions. Information-flow sub-paths comprise frequency, source and target types (e.g., branch, statement), and the length of the information-flow path executed. Opposed to [19], our approach deals with abstractions of data values in the dataflow analysis and not only relies on the relation of correct and failing executions for defect localisation.

7 Conclusions and Future Work

Defect localisation is essential in software engineering, but very time-consuming. (Semi-)Automated localisation therefore is desirable. We have presented an approach based on newly introduced dataflow-enabled call graphs (DEC graphs). It outperforms existing techniques. It targets at defects which affect the dataflow or the controlflow of a programme. Both technical contributions of our approach, the generation and the analysis of DEC graphs, rely on data-mining techniques. Our approach generates DEC graphs by means of meaningful discretisation and derives defect localisations with a weighted graph-mining approach.

Our experiments have shown that the approach may significantly reduce the time required to localise defects in software. On average, only 1.5 out of the 30 methods executed in the case-study system must be investigated to fix a defect.

Future work will extend the approach: (1) Currently, global variables are not handled. Static code analysis might help to identify global variables that are read within a method. They can then be treated like method-call parameters. (2) We plan to investigate an integration with ideas from Masri's approach [19]. Furthermore, as mentioned in Section 3, we will investigate non-primitive data types and 'real defects', originating from open-source software projects [4].

References

1. Asuncion, A., Newman, D.J.: UC Irvine Machine-Learning Repository, available at `http://archive.ics.uci.edu/ml/`
2. Ayewah, N., Hovemeyer, D., Morgenthaler, J.D., Penix, J., Pugh, W.: Using Static Analysis to Find Bugs. IEEE Softw. 25(5), 22–29 (2008)
3. Cheng, H., Lo, D., Zhou, Y., Wang, X., Yan, X.: Identifying Bug Signatures Using Discriminative Graph Mining. In: Proc. Int. Symposium on Software Testing and Analysis (ISSTA) (2009)

4. Dallmeier, V., Zimmermann, T.: Extraction of Bug Localization Benchmarks from History. In: Proc. Int. Conf. on Automated Software Engineering (ASE) (2007)
5. Di Fatta, G., Leue, S., Stegantova, E.: Discriminative Pattern Mining in Software Fault Detection. In: Proc. Int. Workshop on Software Quality Assurance (2006)
6. Eichinger, F., Böhm, K.: Software-Bug Localization with Graph Mining. In: Aggarwal, C.C., Wang, H. (eds.) Managing and Mining Graph Data. Springer (2010)
7. Eichinger, F., Böhm, K., Huber, M.: Mining Edge-Weighted Call Graphs to Localise Software Bugs. In: Proc. ECML PKDD (2008)
8. Eichinger, F., Pankratius, V., Große, P.W.L., Böhm, K.: Localizing Defects in Multithreaded Programs by Mining Dynamic Call Graphs. In: Proc. Testing: Academic and Industrial Conference – Practice and Research Techniques (2010)
9. Han, J., Gao, J.: Research Challenges for Data Mining in Science and Engineering. In: Kargupta, H., Han, J., Yu, P.S., Motwani, R., Kumar, V. (eds.) Next Generation of Data Mining. Chapman & Hall/CRC (2008)
10. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria. In: Proc. Int. Conf. on Software Engineering (ICSE) (1994)
11. Jones, J.A., Harrold, M.J.: Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In: Proc. Int. Conf. on Automated Software Engineering (ASE) (2005)
12. Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of Test Information to Assist Fault Localization. In: Proc. Int. Conf. on Software Engineering (ICSE) (2002)
13. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Proc. European Conf. on Object-Oriented Programming (ECOOP) (2001)
14. Krogmann, K., Kuperberg, M., Reussner, R.: Using Genetic Search for Reverse Engineering of Parametric Behaviour Models for Performance Prediction. IEEE Trans. Softw. Eng. (2010), accepted for publication, to appear
15. Kurgan, L.A., Cios, K.J.: CAIM Discretization Algorithm. IEEE Trans. Knowl. Data Eng. 16(2), 145–153 (2004)
16. Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug Isolation via Remote Program Sampling. SIGPLAN Not. 38(5), 141–154 (2003)
17. Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.P.: SOBER: Statistical Model-Based Bug Localization. SIGSOFT Softw. Eng. Notes 30(5), 286–295 (2005)
18. Liu, C., Yan, X., Yu, H., Han, J., Yu, P.S.: Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs. In: Proc. SDM (2005)
19. Masri, W.: Fault Localization Based on Information Flow Coverage. Softw. Test., Verif. Reliab. 20(2), 121–147 (2009)
20. Nagappan, N., Ball, T., Zeller, A.: Mining Metrics to Predict Component Failures. In: Proc. Int. Conf. on Software Engineering (ICSE) (2006)
21. Philippsen, M., et al.: ParSeMiS: The Parallel and Sequential Mining Suite, available at `http://www2.informatik.uni-erlangen.de/EN/research/ParSeMiS/`
22. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann (1993)
23. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. Morgan Kaufmann (2005)
24. Yan, X., Cheng, H., Han, J., Yu, P.S.: Mining Significant Graph Patterns by Leap Search. In: Proc. SIGMOD (2008)
25. Yan, X., Han, J.: CloseGraph: Mining Closed Frequent Graph Patterns. In: Proc. KDD (2003)
26. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann (2009)