Karlsruher Institut für Technologie

Data-Mining Techniques for Call-Graph-Based Software-Defect Localisation

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der Fakultät für Informatik

des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Frank Eichinger

aus Hannover

Tag der mündlichen Prüfung:	31. Mai 2011
Erster Gutachter:	Prof. Dr.-Ing. Klemens Böhm
Zweiter Gutachter:	Prof. Dr. rer. nat. Ralf H. Reussner

Data-Mining-Techniken für Aufrufgraph-basierte Software-Defekt-Lokalisierung

Motivation und Ziel

Software ist selten gänzlich frei von Fehlern und manuelle Fehlersuche ist eine zeitaufwändige – und damit kostenintensive – Aufgabe. Automatische Fehlerlokalisierungstechniken sind daher überaus wünschenswert. Dies trifft insbesondere für große Softwareprojekte zu, in denen eine manuelle Fehlerlokalisierung nur mit erheblichem Aufwand möglich ist.

Ein noch relativ junger Ansatz zur (halb-)automatischen Fehlerlokalisierung ist die Anwendung von Data-Mining-Techniken auf Aufrufgraphen von Programmausführungen. Solche Graphen stellen üblicherweise Methoden als Knoten und Methodenaufrufe als Kanten dar. Die entsprechenden Fehlerlokalisierungsansätze arbeiten mit Fehlern, die in einigen – aber nicht allen – Programmausführungen auftreten. Konkret können Graph-Mining-Techniken zum Einsatz kommen, die mit Aufrufgraphen arbeiten, die als korrekte oder fehlerhafte Ausführung gekennzeichnet sind. Graph-Mining findet in solchen Graphen Muster, die typisch für fehlerhafte Ausführungen sind. Daraus kann dann – ggf. in Kombination mit weiteren Data-Mining-Techniken – eine Fehlerlokalisierung abgeleitet werden. Eine Softwareentwicklerin bzw. ein Softwareentwickler kann dann mit dieser Information den Fehler deutlich schneller finden und beheben.

Ziel dieser Dissertation im angewandten Data-Mining ist es einerseits, Data-Mining-Techniken für das spezielle Anwendungsproblem – Fehlerlokalisierung in Software – zu entwickeln. Dazu gehört sowohl das Spezifizieren von geeigneten Datenrepräsentationen wie verschiedenartigen Aufrufgraph-Typen, als auch die Entwicklung von darauf abgestimmten Analyseprozessen. Andererseits ist es das Ziel, Graph-Mining-Techniken weiterzuentwickeln. Diese Techniken sollen möglichst über den konkreten Anwendungsfall hinaus, unabhängig von der Anwendungsdomäne, einsetzbar sein.

Beiträge und Vorgehen

Diese Arbeit betrachtet verschiedene Aspekte der Fehlerlokalisierung mit Aufrufgraphen und leistet so vier wesentliche Beiträge. Dabei legt der erste Beitrag die Grundlagen, die weiteren Beiträge erweitern diese indem sie weitere Fehlerarten lokalisieren, skalierbare Ansätze für große Softwareprojekte untersuchen bzw. die Data-Mining-Technik selbst weiterentwickeln:

Fehlerlokalisierung mit gewichteten Aufrufgraphen. Da aus Skalierbarkeitsgründen eine Reduktion von Aufrufgraphen vor der Analyse unabdingbar ist, geht an dieser Stelle viel Information verloren. Um dies zu kompensieren, stellt diese Arbeit einen Ansatz vor, der Aufrufhäufigkeiten von Methoden als Kantengewichte in Aufrufgraphen darstellt. Da bisher keine dedizierte Graph-Mining-Technik für gewichtete Graphen existiert, schlägt diese Arbeit einen kombinierten Ansatz vor: Es kommen herkömmliches Graph-Mining und eine numerische Data-Mining-Technik zum Einsatz. Diese Vorgehensweise erlaubt es insbesondere, solche Fehler zu lokalisieren, die die Aufrufhäufigkeit von Methoden beeinflussen.

Hierarchische Fehlerlokalisierung mit Aufrufgraphen. Graph-Mining-Algorithmen skalieren nicht für große Graphen. Von daher ist es trotz eingesetzter Reduktionen nicht möglich, die bisher entwickelten Techniken unmittelbar auf große Softwareprojekte anzuwenden. Diese Arbeit verfolgt einen anderen Ansatz. Sie beschäftigt sich zunächst mit Graph-Repräsentationen verschiedener Granularitätsstufen (Paket-, Klassen- und Methodenebene) und untersucht erstmalig deren Eignung zur Lokalisierung von Fehlern. Basierend auf solchen Graphen werden dann hierarchische Analyseverfahren entwickelt. Diese lokalisieren Fehler, indem sie auf einer groben Granularitätsstufe beginnen, potentiell fehlerhafte Regionen identifizieren und dann Graphen feinerer Granularität dieser Regionen analysieren. Diese relativ kleinen Ausschnittsgraphen führen deutlich seltener zu Skalierbarkeitsproblemen.

Fehlerlokalisierung mit Datenfluss-annotierten Aufrufgraphen. Mit bisherigen Aufrufgraph-basierten Techniken ist es nicht möglich, Fehler zu lokalisieren, die nur den Datenfluss verändern. Das liegt daran, dass keine bisherige Aufrufgraph-Darstellung Datenflüsse beinhaltet. Eine solche Darstellung zu finden ist allerdings schwierig, da eine einzelne Kante typischerweise sehr viele Methodenaufrufe – und somit Datenflüsse – repräsentiert. Diese Arbeit schlägt eine Aufrufgraph-Repräsentation und Analysetechnik vor, die Abstraktionen von Datenflüssen, generiert durch einen Diskretisierungsansatz, beinhaltet. Dadurch können (neben anderen) vor allem solche Fehler lokalisiert werden, die primär den Datenfluss beeinflussen.

Gewicht-Constraint-basiertes approximatives Graph-Mining. Ein anderer Ansatz Skalierbarkeitsproblemen zu begegnen neben den zuvor beschriebenen hierarchischen Verfahren, ist das Formulieren von Constraints (Bedingungen) oder auch das Zulassen von approximativen Ergebnismengen. Bisherige Constraint-basierte Graph-Mining-Algorithmen geben Garantien bzgl. der Vollständigkeit der Ergebnismengen, betrachten allerdings keine gewichteten Graphen. Dies hängt damit zusammen, dass kein gesetzmäßiger Zusammenhang zwischen Graph-Topologie und Gewichten besteht. Da sich in dieser Arbeit gewichtete Aufrufgraphen jedoch als sinnvolles Konzept erwiesen haben, wird hier dennoch die Verwendung von Gewicht-basierten Constraints untersucht. Es wird gezeigt, dass sie sowohl zu performanteren Algorithmen führen, als auch dass in der praktischen Anwendung auf Garantien bzgl. der Vollständigkeit verzichtet werden kann.

Ergebnisse und Ausblick

In dieser Dissertation werden verschiedene Data-Mining-basierte Techniken zur Fehlerlokalisierung in Software entwickelt, sowie Graph-Mining-Techniken weiterentwickelt. Die Fehlerlokalisierung weist in der Evaluation durchschnittlich doppelt so präzise Ergebnisse auf wie eine verwandte Aufrufgraph-basierte Technik. Die Ergebnisse können durch die Berücksichtigung von Datenflüssen nochmals verbessert werden. Des Weiteren wird in dieser Arbeit erstmalig eine Aufrufgraph-basierte Technik mit Fehlern aus der Praxis eines großen Softwareprojekts erfolgreich evaluiert. Beim Gewicht-Constraint-basierten Graph-Mining wird in dieser Arbeit eine Ausführungsbeschleunigung um den Faktor 3,5 erzielt, bei gleichbleibender Präzision in der Fehlerlokalisierung. Um die Generalität dieses Ansatzes zu zeigen, wird er zusätzlich mit Graph-Daten einer ganz anderen Domäne, der Transportlogistik, evaluiert.

Wie alle Fehlerlokalisierungstechniken sind auch die in dieser Arbeit vorgeschlagen Techniken nicht in der Lage, alle Arten von Fehlern zu lokalisieren. – Ihre Stärken liegen in der Lokalisierung solcher Fehler, die sich auf die Aufrufgraphen bzw. auf die Datenflüsse niederschlagen. Eine Ergänzung durch andere Techniken ist daher sinnvoll, um ein möglichst breites Spektrum an Fehlerarten abzudecken.

Eine immer wichtiger werdende Entwicklung in der Softwaretechnik ist die Entwicklung von mehrfädiger Software für Mehrprozessorsysteme. In solchen Umgebungen treten eigene Arten von Fehlern auf (z.B. Synchronisationsfehler), die besonders schwierig zu lokalisieren sind. Dies ist vor allem darin begründet, dass sie indeterministisch auftreten. Diese Arbeit zeigt unter anderem, dass ein Teil dieser Fehler bereits mit Aufrufgraph-basierten Techniken lokalisiert werden kann. Durch erweiterte Graph-Repräsentationen und Lokalisierungstechniken, die die Spezifika paralleler Ausführungen explizit berücksichtigen, ist die Lokalisierung weiterer Fehlerarten (z.B. bestimmte Wettlaufsituationen) zu erwarten.

Acknowledgements

First of all, I would like to express my deep appreciation to Professor Dr. Klemens Böhm, the supervisor of my dissertation. Throughout the five years at the Karlsruhe Institute of Technology (KIT), Klemens provided me with constructive suggestions and comments, pushed me towards publishing results at highly reputable venues and provided all the assistance I needed to conduct my research. In particular, Klemens taught me how to write research papers. When a deadline was approaching, I received his feedback within hours, no matter at which time of day or night.

At the KIT, I was working with a team of students, who were doing their projects and theses under my supervision or were contributing as working students to my research. I have discussed many of the ideas underlying this dissertation with these team members, and many results have been achieved with the help of these persons.

In particular, I would like to express my gratitude to the following students who provided most valuable help: Matthias Huber contributed many ideas in call-graph-based defect localisation and constraint-based mining and put them into practice. Roland Klug worked with me on the localisation of dataflow-affecting bugs and did the implementation and experiments. Roland also conducted the first experiments on hierarchical defect localisation. Christopher Oßner continued this work. Besides implementation and experiments, Chris contributed important ideas on hierarchical defect localisation. My college Philipp W. L. Große worked on defect localisation in multithreaded programmes and ran first experiments. These were continued by Alexander Bieleš, who provided valuable help and conducted many experiments. Jonas Reinsch contributed by reimplementing approaches from the related work and conducting experiments.

Besides the persons who worked with me on the research directly related to my dissertation, I would like to thank my colleges from Klemens's group at the Institute for Programme Structures and Data Organisation (IPD). All of them have contributed to a pleasant working experience. Furthermore, they provided me with the possibility to discuss questions regarding my research and helped me with technical and organisational issues. In particular, I would like to mention my officemates, Chris and Dr. Mirco Stern, as well as Dr. Erik Buchmann and Jutta Mülle. Further, I would like to express my sincere thanks my collages Matthias Bracht, Dr. Stephan Schosser and, again, Mirco. We gave the lecture on foundations of database systems and the practical course on data warehousing and mining together, and I enjoyed our pleasant cooperation very much. When writing research papers, a number of further colleges provided essential and helpful comments, in particular Dr. Thorben Burghardt,

Dr. Björn-Oliver Hartmann, Martin Heine, Dr. Emmanuel Müller, Heiko Schepperle and Dr. Silvia von Stackelberg.

Not only the members of Klemens's group have contributed to my work, but as well further colleges at the IPD. In particular cooperation with Dr. Klaus Krogmann on localising dataflow-affecting bugs and with Dr. Victor Pankratius on localising defects in multithreaded programmes have been very inspiring and helped me looking at research problems from different perspectives. Further thanks go to a number of scholars I have discussed my work with, in particular to Professor Dr. Xifeng Yan from the University of California, USA and Professor Dr. Andreas Zeller from Saarland University, Germany.

Special thanks go to my second supervisor, Professor Dr. Ralf H. Reussner. He provided valuable insights on localising dataflow-affecting bugs, gave feedback on my research papers and helped me designing experiments.

At the end, I would like to thank my parents and my friends. My parents supported me throughout the years and gave me the possibility to study at the university. Many of my friends supported me all the time and gave me a life outside university. The last credits I would like to dedicate to a very special person. Birte, you are wonderful!

Karlsruhe, May 2011

Contents

1 Introduction

Software is rarely free from defects that cause failing behaviour. On the one side, failures experienced by the users are annoying, and they cost the economy billions of dollars annually [RTI02]. This is in particular severe when failures occur after the software was released. On the other side, manual debugging of software can be extremely expensive, too. More concretely, localising defects is considered to be the most time-consuming and difficult activity in this context [DLZ05, JH05], and studies have shown that 35% of the overall development time is spent for debugging activities [RTI02]. Automated means to localise defects and to guide developers debugging a programme are therefore more than desirable [ZNZ08]. If a developer obtains some hints where defects might be located, debugging becomes more efficient. Certainly, the respective techniques should localise a defect as precisely as possible. More specifically, they should exclude most of the code from being analysed by humans. Furthermore, a defect-localisation technique should be able to deal with a wide range of defects. However, research has shown that none of the existing techniques for defect localisation is perfect, i.e., is able to localise any kind of defect [RAF04, SJYH09]. It is therefore still worthwhile to investigate further directions of defect-localisation techniques that support developers in eliminating failing behaviour.

One way to localise defects in software is to analyse *dynamic call graphs* with *graph-mining techniques* [CLZ+09, DFLS06, LYY+05]. Such graphs are representations of programme executions. Analysing call graphs aims at finding anomalies in failing executions. Graph mining in turn is a general technique for the analysis of graph structures, and it is one of the more recent developments in data mining [AW10c, CH06]. Graph mining bears the potential to produce very precise mining results, in particular compared to more traditional techniques that rely on data representations that are less complex. The rationale is that many real-world artefacts – such as programme executions – can be represented very precisely by means of graph structures. The power of analysing such structures has impressively been demonstrated by the PageRank algorithm [BP98] for ranking results in web search, as well as by many further link-mining applications [YHF10].

Software engineering and defect localisation in particular has been identified as a rewarding and challenging area for applied data mining [DDG+08, HG08, XTLL09]. Further, tackling challenging application problems – such as defect localisation in software – might lead to innovations in the data-analysis domain and in the application domain as well [HCXY07]. In this dissertation, we elaborately investigate

graph-mining techniques for the analysis of dynamic call graphs and ultimately for the localisation of defects in software. This direction of research has been of interest in both scientific communities, data mining [AW10b, LYY+05] and software engineering [CLZ+09, DFLS06]. This dissertation in applied data mining likewise is motivated, solves challenges and contributes in both fields, data mining and software engineering. In particular, this includes advances in defect localisation, problem-oriented graph data representations and analysis techniques.

1.1 Localising Defects in Software

Research in the field of software reliability has been extensive, and various techniques have been developed for defect localisation – some of them building on data mining. Techniques for defect localisation are either *static* or *dynamic*, i.e., they deal with source code only, or they analyse programme executions, respectively.

Static techniques typically rely on code-quality measures or on identifying typical defect-prone programming patterns. Programme components with suspicious values of the measures or patterns identified to be defect prone are then good hints for localising defects. However, static approaches typically lead to many false-positive warnings, and they have difficulties discovering important classes of hard-to-find defects [RAF04].

Dynamic techniques in turn analyse programme executions and typically compare the characteristics from *correct* and *failing* executions. This helps to identify anomalies in the executions, which localisation techniques then suspect to refer to defects. The different approaches use different information derived from executions, as well as different methodologies to derive defect localisations. Two of the best dynamic approaches, which have outperformed a number of competitors, are SOBER [LFY+06] and Tarantula [JH05] with its variations [AZGvG09]. However, even though these techniques have proven to detect certain defects very well, they do not analyse all kind of information that could be obtained from programme executions and is potentially of relevance. To name one example, Tarantula only makes use of the information whether a certain piece of code is executed or not. Certain defects however might alter the number of times a piece of code is executed, which these approaches do not consider.

Example 1.1: The example Java programme given in Listing 1.1 could have a defective loop condition in Line 16. This would be a *call-frequency-affecting bug*, as such a defect would affect the execution frequency of Line 17 and thus the call frequency of method a. Approaches such as Tarantula would not notice this effect.

Analysing dynamic call graphs is a relatively recent dynamic defect-localisation approach [CLZ+09, DFLS06, LYY+05]. It is promising, since such graphs contain much detailed and fine-grained information regarding programme executions, which

```
1  public class Example {
2    static java.util.Random generator;
3
4    public static void main(String[] args) {
5      generator = new java.util.Random();
6      if (generator.nextInt(100) < 99)
7        a(0);
8      b(3);
9    }
10
11   private static void a(int x) {
12     // some application code
13   }
14
15   private static void b(int y) {
16     for (int i = 0; i < y; i++)
17       a(generator.nextInt(100));
18   }
19 }
```

Listing 1.1: An example Java programme.

can hardly be found in any other representation. In particular, call graphs reflect the structure of method invocations of an execution – or the relationship of more fine-grained or more coarse-grained programme components, as we will see. In method-level call graphs, methods are represented as nodes and method calls as edges.

Example 1.2: In a typical execution of the example programme given in Listing 1.1, method $main$ calls method a once, before it calls method b. Method b then calls method a three times. The call graph in Figure 1.1(a) reflects this behaviour.

Besides the advantages of call-graph analysis, mining graphs is much more complex than many other analysis techniques. Therefore, to cope with the size of call graphs, they are typically reduced to compact representations where one edge stands for a number of method calls. However, call-graph-based defect localisation can still be computationally expensive and can lead to scalability problems.

While related work in call-graph-based defect localisation has investigated basic call-graph representations, we extend call graphs with more information relevant for the localisation of defects. In particular, this information refers to the context of method invocations, execution frequencies and dataflows. These extensions aim at broadening the rage of detectable defects.

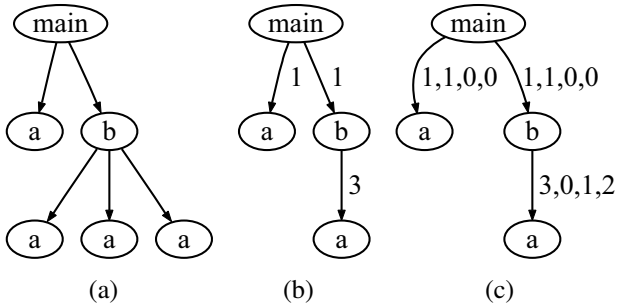


Figure 1.1: Example call graphs referring to the programme given in Listing 1.1.

Example 1.3: Figure 1.1(b) continues Example 1.2. It is an example for a reduced representation of the graph given in Figure 1.1(a). Further, it includes call frequencies as numerical edge weights. Fluctuating frequency values can be identified by analysis techniques and might be a hint for a defect.

The graph in Figure 1.1(c) contains additional information related to the dataflow. It is annotated with tuples of weights at the edges and can be analysed by mining techniques. In this simplified example, the first tuple element is the call frequency, as before. The other three tuple elements stand for the number of method calls with parameter values falling into the intervals 'low', 'medium' and 'high'. Concretely, imagine that method b calls method a with values 98, 83 and 50 for parameter x. The tuple $3, 0, 1, 2$ then stands for three calls in total, zero calls with a low value, one call with a medium value and two calls with a high value.

Until today, call-graph-based defect localisation has not been studied extensively. Therefore, many questions concerning such techniques are currently unanswered. This includes the question what kind of defects can be localised and how well the techniques scale. In this dissertation, we investigate the potential of call-graph-based techniques and different call-graph representations for defect localisation. Therefore, it is not the primary aim to develop a technique which rules out any existing technique, but to comprehensively investigate the usage of call graphs for defect localisation.

1.2 Call-Graph Mining for Defect Localisation

Mining call graphs as described before introduces two main challenges for data analysis, that partly depend on each other:

1. Finding adequate data representations
2. Analysing the resulting graphs

Data Representations. Finding adequate data representations is an inherent part of the knowledge-discovery process [CCK+00, FPSS96]. This non-trivial process is the general data-analysis procedure, aiming at the discovery of "valid, novel, potentially useful and ultimately understandable patterns in data" [FPSS96]. Data mining is only one step within the process; the other steps range from understanding the application domain to the deployment of the analysis technique. Finding an adequate data representation and acquiring this data are the steps preceding the actual data-mining step. In particular, problem-specific data representations have been identified to be key for the success of any applied data-mining problem [HG08]. In the software-engineering application domain of this dissertation, call graphs are the dedicated data representation. However, it is not obvious how exactly to represent the call-graph structure (topology) in order not to lose any important information and to obtain graphs of a manageable size. Other aspects are the granularity of call graphs and the question how to incorporate more domain-specific information such as information regarding calls of methods that belong to the programming language. Graphs at granularities different from the method level have rarely been investigated in a defect-localisation context, and finding representations is challenging as one would like not to lose too much information in coarse graph representations. Further, it is demanding to come up with adequate representations for call frequencies and – more importantly – for dataflows, which we identify to be crucial for the localisation of certain defects. Another challenging area is the definition of call graphs for multithreaded programmes, where parts of the programme are executed in parallel.

Mining Weighted Call Graphs and Localising Defects. Besides the data representation, defining the actual analysis procedure is the other main challenge for call-graph-based defect localisation. It leads to three further subproblems:

- How to mine call graphs that are weighted?
- How to deal with scalability issues caused by large graphs?
- How to derive actual defect localisations?

Weighted subgraph mining. As we will see, we identify different types of weighted graphs to be natural and adequate representations for our mining problem. However, weighted-subgraph mining has not been investigated comprehensively, and there are no obvious ways for the analysis of our weighted call graphs. Most techniques that have been proposed for mining weighted graphs are very specific for the respective mining problem and application domain, and they cannot be applied for defect localisation. It is therefore an unsolved problem how subgraph mining with weighted graphs can be achieved in general. This problem is difficult to solve, since it brings together the domain of graph structures (topologies) and the domain of numerical weights. These two domains are in general not connected by means of a guaranteed

law. This makes it difficult to deal with both kinds of information within the same algorithm.

Scalability issues. Scalability of subgraph-mining algorithms is challenging, since frequent subgraph mining inherently involves *subgraph-isomorphism problems*. This problem is known to be NP-complete [GJ79]. Therefore, finding efficient algorithms is not easy. For instance, approximate and constraint-based algorithms might solve the scalability problem, but bear a trade-off between scalability and possibly worse defect-localisation results. This is as such techniques lead to smaller result sets that might contain less information relevant for defect localisation. Besides adopted mining techniques, the scalability problem can be tackled by a fall back to the data-representation problem. When doing so, the aim is to find suitable graph representations that can be mined more easily. However, this bears a similar trade-off.

Deriving defect localisations. There are many different ways to derive a defect localisation based on results from mining weighted subgraphs. Such a localisation technique should be efficiently computable, should cover a possibly wide range of different types of defects and should ultimately be useful for software developers. Thus, finding a technique that fulfils all these characteristics is difficult.

1.3 Contributions of this Dissertation

In order to solve the challenges mentioned in Sections 1.1 and 1.2, this dissertation features contributions in both domains: in software engineering and at the different stages of the knowledge-discovery process. The contributions described in the following two paragraphs are our basic approach for defect localisation with weighted call graphs. The following paragraphs build on this approach and extend it in order to broaden the range of detectable defects and to scale for larger software projects. These extensions deal with both the data representation and the mining techniques. The last paragraph subsumes the results in defect localisation.

Weighted-Call-Graph Representations. Reducing the size of call graphs as directly obtained from programme executions is mandatory, caused by scalability problems. However, this leads to a loss of information which might be relevant for defect localisation. In this dissertation, we propose an approach that reduces the size of the graphs. It does so to an extent that keeps important structural information. Further, our approach annotates call frequencies as numeric edge weights. This information would be lost otherwise. This call-graph representation allows in particular for the localisation of an important class of defects, *call-frequency-affecting bugs*.

Data-Mining-Based Defect Localisation with Weighted Call Graphs. To analyse the weighted call graphs proposed – in the absence of a suitable out-of-the-box technique for weighted graph mining – we propose a combined approach: It

utilises vanilla frequent-subgraph-mining techniques in a first step and employs a traditional data-mining technique, feature selection, in a subsequent analysis step. To broaden the range of detectable defects, we further propose combination strategies to incorporate the detection of another class of defects, *structure-affecting bugs*. Ultimately, we derive a ranking of methods, ordered by their likelihood to be defective. A software developer can then use this ranking to investigate the methods, starting with the one suspected to be most suspicious.

Hierarchical Defect Localisation with Graphs at Different Granularities. Graph-mining algorithms do not scale well for large graphs, even if tough call-graph-reduction techniques are applied. Therefore, it is not possible to apply existing call-graph-based defect-localisation techniques to large software projects. In order to apply the developed defect-localisation techniques to such large projects, we develop hierarchical procedures in this dissertation. To this end, we firstly propose novel call-graph representations at different levels of granularity, i.e., at the package, class and method level. We then investigate their usefulness for defect localisation and propose various hierarchical analysis procedures. These procedures localise defects starting at the most coarse-grained call-graph representation. There they identify potentially defective regions in the code. Then, they proceed with finer-grained graphs of the previously identified regions etc. Such graphs, representing small regions of the whole graph, lead to scalability issues in much fewer cases.

Defect Localisation with Dataflow-Enabled Call Graphs. Existing call-graph-based defect-localisation techniques do not allow for the localisation of defects that affect the dataflow of a programme execution rather than the method-call structure. This is as such techniques obviously can only detect defects that influence the call graph, which is not the case with such defects. Finding a respective call-graph representation is difficult, as edges in a call graph typically represent huge numbers of method calls and correspondingly huge numbers of dataflows. In this dissertation, we propose *dataflow-enabled call graphs* that extend call graphs with abstractions referring to the dataflow. We derive the graphs using discretisation techniques. Furthermore, we extend the defect-localisation technique to deal with the resulting graphs. With these extensions, we are able to localise defects that primarily affect the dataflow, besides other classes of defects.

Mining Weighted Graphs with Weight-Based Constraints. Besides the aforementioned hierarchical approach, constraint-based mining is a further approach with the potential to increase scalability. Such algorithms lead to smaller result sets and make use of pruning opportunities in the mining algorithms. However, existing constraint-based graph-mining algorithms do not deal with weighted graphs. This is as most weight-based constraints do not fulfil certain properties – most impor-

tantly *anti-monotonicity* – which theoretically forbids their usage as pruning criterion. In this dissertation, we do develop weight-based constraints and integrate them into pattern-growth algorithms for frequent subgraph mining. We do so as weight-based constraints seem to be a well suited general approach for mining weighted graphs. As mentioned, weight-based constraints cannot be employed for pruning – in theory. In this dissertation, we do so nevertheless and study the effects. The rationale for this investigation is that there is evidence that weights and graph structures are frequently correlated in real-world graphs [MAF08]. As mining with such constraints can lead to approximate results, i.e., to incomplete result sets, we study the completeness and the usefulness of such constraints. The result is that weight-based constraints lead to both a better performance of mining algorithms and well results in practice. Concretely, we demonstrate that guaranteeing completeness of mining results in abdicable in the analysis problems investigated – not only in our software-engineering application. Besides defect localisation, we evaluate our approach with datasets from transportation logistics and consider different analysis problems, i.e., graph classification and explorative mining. We do so to demonstrate the broad applicability of the weight-based constraints proposed.

Results in Software-Defect Localisation. The results of defect localisation using the call-graph representations and localisation techniques developed in this dissertation are encouraging: Compared to existing call-graph-based techniques, the approaches developed display on average a doubled localisation precision. These results can be improved when employing dataflow-enabled call graphs. Compared to more established approaches from the software-engineering domain [AZGvG09, JH05, LFY+06], our approach was able to derive better defect-localisation results in 12 out of 14 cases in our test suite. Further, for the first time, we successfully apply call-graph-mining-based defect localisation to real-world defects from a real and relatively large software project (Mozilla Rhino, ≈ 49k lines of code). In our setup, our approach narrows down the amount of code a developer has to examine to about 6% of the whole project on average. In constraint-based mining, we achieve a speed-up of 3.5 while obtaining even slightly better defect-localisation results.

1.4 Outline of this Dissertation

We now describe the contents of the remainder of this dissertation. Chapters 2 and 3 introduce the background and the related work, respectively. Chapters 4 and 5 describe our basic defect-localisation approach, and Chapters 6–8 are extensions thereof. Chapters 5–8 include evaluations of the respective techniques. Chapter 9 concludes.

In Chapter 2, we describe the background of this dissertation. In particular, we introduce the backgrounds from graph theory, software engineering and data mining.

These descriptions are limited to an extent that one can follow the descriptions in the succeeding chapters.

In Chapter 3, we discuss related work. This chapter is divided into two parts, one on defect localisation and one on data mining. In the defect-localisation part, we discuss the different existing approaches for defect localisation (not including call-graph-mining-based techniques) and contrast them to the techniques developed in this dissertation. In the data-mining part, we discuss existing techniques for mining weighted graphs, for mining significant subgraphs (including approximative techniques) and for constraint-based subgraph mining. These techniques are related, as we propose different ways for mining weighted graphs in this dissertation, including an approximate constraint-based technique.

Chapter 4 is about *call-graph representations*. This includes representations proposed by other authors from the closely related work, as well as the call-graph representations that are new in this dissertation. In particular, we introduce different kinds of weighted call graphs. We discuss all these graph representations within the same chapter, as they are closely related to each other, and as this allows for a better comparison. In particular, we focus on call graphs at the method level in this chapter, i.e., one node in a call graph represents a method. Then, we comment on call-graph representations on other levels of granularity, we propose graph representations for multithreaded programmes, and we say how we actually derive call graphs from programme executions.

In Chapter 5, we describe *defect localisation* based on the call-graph representations discussed before. Again, we discuss closely related techniques dealing with traditional graph representations within the same chapter as the techniques newly proposed in this dissertation. This is, we discuss existing structural techniques for defect localisation, followed by novel frequency-based approaches. We also propose possibilities to combine both kinds of approaches in order to be able to detect a broader range of defects. Then, we present an evaluation that compares selected graph representations and mining techniques. Besides the techniques described in this chapter, we also compare our newly proposed approach to established approaches from the related work in software engineering.

Chapter 6 is about *hierarchical defect localisation*. This is, we generalise our call-graph representations to deal with call graphs at several levels of granularity. This allows us to propose hierarchical mining procedures that start with call graphs at coarse levels of granularity, before *zooming-in* into regions of the call graphs suspected to be defective. This aims at a scalable technique for defect localisation. We evaluate this technique with a relatively large software project along with real defects.

In Chapter 7, we deal with the *localisation of dataflow-affecting bugs*. We first introduce *dataflow-enabled call graphs*, which are call graphs incorporating abstractions referring to the dataflow. Then we say how we derive dataflow-enabled call graphs by means of tracing and supervised discretisation. In order to localise data-

flow-affecting bugs along with other types of defects, we adopt our mining technique from the preceding chapters. Finally, we evaluate this new approach.

Chapter 8 is about *constraint-based mining of weighted graphs*. This technique is motivated by our defect-localisation problem, but is actually a more general technique for mining weighted graphs. Concretely, we introduce weight-based constraints, and we explain how to integrate them into pattern-growth-based frequent subgraph mining. In this chapter, we also explain different analysis settings where mining with weight-based constraints is of relevance, including the application to defect localisation. Ultimately, we evaluate the different analysis settings with graph data from software engineering and transportation logistics.

Chapter 9 concludes this dissertation. Besides a short summary, we highlight the lessons learned, and we explain some directions for future work.

Portions of the whole work have been published in [EB10, EBH08a, EBH08b] (weighted call-graph representations and basic defect-localisation techniques, Chapters 4 and 5), [EB09, EOB11] (hierarchical defect localisation, Chapter 6), [EKKB10] (localisation of dataflow-affecting bugs, Chapter 7), [EHB10a, EHB10b] (constraint-based mining of weighted graphs, Chapter 8) and [EPGB10] (multithreading defect localisation, Appendix A).

2 Background

This dissertation is about applied data mining, it has a dedicated field of application, software defect localisation, and it focuses on techniques for the analysis of call graphs. This chapter therefore first introduces the formal graph-theoretic backgrounds (Section 2.1). Then it discusses important concepts from the field of application, software engineering (Section 2.2), and finally, it introduces the relevant data-mining techniques (Section 2.3).

2.1 Graph Theory

We now introduce the basic concepts of graphs and trees from a graph-theoretic point of view that are relevant in this dissertation (Sections 2.1.1 and 2.1.2, respectively).

2.1.1 Graphs

In this dissertation, graphs are typically *labelled*:

Definition 2.1 (Labelled graphs)
A labelled graph *is a four-tuple:* $G := (V, E, L, l)$. V *is the set of vertices*[1]*,* $E \subseteq V \times V$ *the set of edges,* L *a set of categorical labels and* $l : V \cup E \to L$ *a labelling function.* $E(G)$ *denotes the set of edges of* G*,* $V(G)$ *the set of vertices and* $L(G)$ *the set of labels.*

Sometimes we do not explicitly mention the labels of edges. In this case, all edges have the same default label. Further, graphs can be *weighted*:

Definition 2.2 (Labelled weighted graphs)
A labelled weighted graph *is a six-tuple:* $G := (V, E, L, l, W, w)$. V*,* E*,* L *and* l *are as in Definition 2.1,* $W \subseteq \mathbb{R}$ *is the domain of the weights, and* $w : E \to W$ *is a function which assigns weights to edges.*

All techniques discussed in this dissertation can easily be extended to cover nodes that are weighted ($w : V \cup E \to W$). Further, tuples of weights can be handled with the following variation: $W \subseteq \mathbb{R}^n, n \in \mathbb{N}$.

[1] In this dissertation, we use *vertex* and *node* interchangeably.

Notation 2.1 (Properties of graphs)
All graphs can be directed *or* undirected *($e \in E$ is an* ordered *tuple or an* unordered *set, respectively). Further, all graphs can be* connected *(any two nodes are connected by at least one* path*) or* unconnected *(there exists at least one pair of nodes that is not connected by a* path*; the graph consist of several* components*). See [Die06] for further details.*

If not mentioned explicitly, we deal with *directed* and *connected* graphs in this dissertation. To explicitly distinguish graphs from *trees* (see Section 2.1.2), we call graphs that might include cycles also *general graphs*.

Definition 2.3 (Subgraphs (see [Die06]))
A labelled graph G' is a subgraph *of a labelled graph G (and G a* supergraph *of G') if and only if $V(G') \subseteq V(G), E(G') \subseteq E(G), L(G') \subseteq L(G)$, and G' preserves the labelling defined in G. $G' \subseteq G$ denotes such a subgraph-supergraph relationship. If $G' \subseteq G$ and $G' \neq G$, G' is called a* proper subgraph *of G, denoted $G' \subset G$.*

Note that weights are not considered for the definition of subgraphs.

Definition 2.4 (Subgraph-isomorphism problem)
The question whether a given graph G' is a subgraph *from another given graph G ($G' \subseteq G$) is called the* subgraph-isomorphism problem*.*

The *subgraph-isomorphism problem* as defined before for general graphs is known to be *NP-complete* [GJ79].

2.1.2 Trees

Trees are variants of graphs. As they are relevant in the software-engineering domain, too, we now briefly introduce the most important notions.

Definition 2.5 (Trees (see [Die06]))
An acyclic connected graph*, i.e., a connected graph which edges do not form a circle, is called a* tree *(see Definition 2.1 and Notation 2.1 for the definition of connected graphs).*

In this dissertation, trees are (as graphs) always *labelled*, and they can be *weighted*, too (in this case, Definition 2.2 applies accordingly).

Notation 2.2 (Properties of trees)
Nodes having only one outgoing/incoming edge are called leaves*, nodes that are connected to the same node are called* siblings*. When trees are* undirected *(and unordered, see Notation 2.3), they are also called* free trees *[CMNK05]. When trees are* directed *and two nodes are connected by an edge, one calls them* parent *and* child*. A tree T with a dedicated* root node *$r \in V(T)$ is called a* rooted tree*.*

Notation 2.3 (Unordered and ordered trees)
As with graphs, trees are by default unordered*, as V and E are unordered sets.* Rooted trees *can also be* ordered*. In this case, one has to define an order between all siblings that are children from the same parent node.*

In the context of software engineering, we typically deal with both labelled and directed *rooted unordered trees* and labelled and directed *rooted ordered trees*.

Definition 2.6 (Subtrees (see [CMNK05]))
The definition of subtrees *is the same as the one for* subgraphs *given in Definition 2.3. When dealing with* ordered trees*, the ordering among the siblings in the subtree has in addition to be a subordering of the corresponding vertices in the supertree.*

Chi et al. [CMNK05] describe further variations for the definition of subtrees besides the one for *induced subtrees* given in Definition 2.6.

2.2 Software Engineering

As we now know the theoretical background of graphs and trees, we now first discuss the most important graphs in software engineering (Section 2.2.1). We then clarify our notion on failing behaviour in software (Section 2.2.2) and introduce the foundations of software testing and debugging (Section 2.2.3).

2.2.1 Graphs in Software Engineering

Graphs have been used for a long time in different subdisciplines of software engineering. The most important distinction is if the graphs are *static* or *dynamic*, i.e., if they represent aspects from the source code or from programme executions, respectively. In the following, we introduce the graphs that are relevant in this dissertation.

Control-Flow Graphs (CFGs)

Control-flow graphs (CFGs) [All70] are static representations of source code, frequently used in compiler technology. In a CFG, the source code is divided into several so-called *basic blocks*. Each basic block consists of all statements that are always executed conjunctively, i.e., new blocks start when the control flow changes (due to, e.g., an `if` or `for` statement). CFGs are unweighted general graphs:

Notation 2.4 (Control-flow graphs (CFGs, see [All70]))
In a CFG, each basic block *is represented as a node, and* control dependencies *are represented as edges connecting these nodes.*

Example 2.1: Figure 2.1(a) is the control-flow graph (CFG) from the example source code given in Listing 2.1.

Programme-Dependence Graphs (PDGs)

Programme-dependence graphs (PDGs) [OO84] are static graphs as well, and they are typically used in programme slicing [KL88] and optimisation [FOW87]. While CFGs reflect the pure control structure of a programme, PDGs incorporate additionally dataflow-related information. To this end, they require a finer level of granularity than CFGs, as dataflows might occur between the individual statements within a basic block in a CFG. As CFGs, PDGs are unweighted general graphs:

Notation 2.5 (Programme-dependence graphs (PDGs, see [OO84]))
In a PDG, every statement forms its own node (with few exceptions). Further, there is a dedicated entry *node, and there are extra nodes representing every parameter of a method*[2]*. A control edge connects a node a with a node b if and only if the execution of node b depends on node a. Besides control edges, nodes in PDGs are connected by means of edges of another type (technically of another label; say, 'data' instead of 'control') when there is a dataflow between the nodes.*

Example 2.2: Figure 2.1(b) is the programme-dependence graphs (PDG) from the example source code given in Listing 2.1. Control dependencies are displayed by solid lines, data dependencies by dashed lines.

Static and Dynamic Call Graphs

Call graphs can be both *static* or *dynamic* [GKM82]. A *static call graph* [All74] can be obtained from the source code. It represents all methods of a programme as nodes and all possible method invocations as edges. We deal with *dynamic call graphs* (sometimes also called *call trees*) in this dissertation. They represent an execution of a particular programme and reflect the actual invocation structure of the particular execution. Chapter 4 provides detailed definitions for the various variants of call graphs.

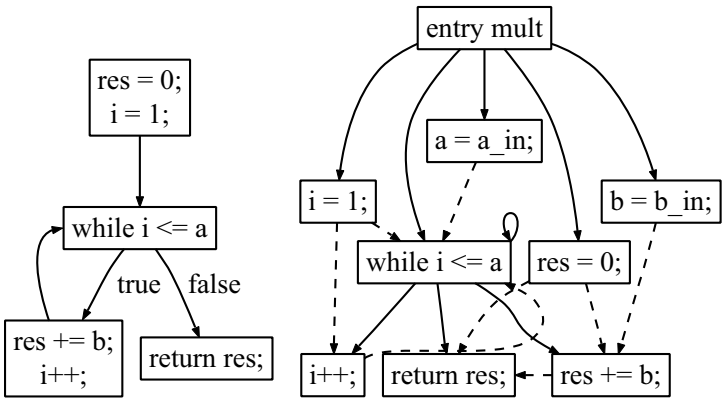
Without any further treatment, an (unreduced) call graph is an unweighted *rooted ordered tree*. The main method of a programme is the root, and the methods invoked directly are its children. Originally, the siblings are ordered by the time of execution. However, unreduced call graphs become very large and are typically reduced to smaller call graphs, which are weighted or unweighted *general graphs*.

In Chapter 4 (and as well in Chapters 6 and 7), we discuss call graphs to a larger extend, including the different reduction techniques, further variants of the graphs and the question how to actually derive such graphs from programme executions. As

[2] In this dissertation, in a software context, we use *method* interchangeably with *function*.

```
1 public static int mult(int a, int b) {
2   int res = 0;
3   int i = 1;
4   while (i <= a) {
5     res += b;
6     i++;
7   }
8   return res;
9 }
```

Listing 2.1: Example Java method performing an integer multiplication.



(a) CFG (b) PDG

Figure 2.1: A control-flow graph (CFG) and a programme-dependence graph (PDG) for the method `int mult(int a, int b)` as given in Listing 2.1.

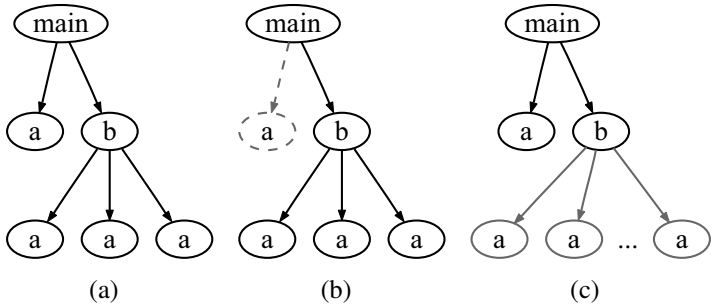


Figure 2.2: (a) An *unreduced dynamic call graph*, (b) a call graph with a *structure-affecting bug* and (c) with a *frequency-affecting bug*.

we will see in Section 4.2 and Chapter 6, call graphs can also be defined at levels of granularity different from the method level. For instance, basic blocks, classes or packages might form the nodes of a call graph.

2.2.2 Bugs, Defects, Infections and Failures in Software

In the field of debugging, one usually avoids the terms *fault*, *bug* and *error*, but distinguishes between *defects*, *infections* and *failures* [Zel09]. In this frequently-cited classification, these terms have the following meaning:

- **Defects** are the places in the source code which cause a problem.
- **Infections** are incorrect programme states (usually triggered by defects).
- **Failures** are an observable incorrect programme behaviour (e.g., a user experiences wrong calculations).

In this dissertation, we use the term *bug* when referring to different types of failing behaviour. We now introduce a more detailed differentiation of our own (unless otherwise stated), which is in particular useful when dealing with call-graph-based defect localisation:

- **Crashing and non-crashing bugs [LYY+05]:** *Crashing bugs* lead to an unexpected termination of the programme. Prominent examples include null-pointer exceptions and divisions by zero. In many cases, e.g., depending on the programming language, the respective defects are not hard to find: A stack trace is usually shown which gives hints where the infection occurred. Harder to cope with are *non-crashing bugs*, i.e., failures which lead to wrong results without any hint that something went wrong during the execution [LYY+05, LCH+09].

  As non-crashing bugs are hard to find, all approaches to localise defects with call-graph mining (including the ones proposed in this dissertation) focus on

them and leave aside crashing bugs. However, when call graphs can be generated from crashing programme executions, there are no obstacles in localising the respective defects with call-graph-based techniques in the same way as non-crashing bugs.

- **Occasional and non-occasional bugs:** *Occasional bugs* are failures which occur with some but not with any input data. In the context of multithreaded programmes, occasional bugs can also arise when the programme input remains the same, but different thread schedules are executed. Finding occasional bugs is particularly difficult, as they are harder to reproduce, and more programme executions 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 call-graph-based defect-localisation techniques (including the ones proposed in this dissertation) rely on comparing call graphs of failing and correct programme executions, they deal with occasional bugs only. In other words, besides examples of failing programme executions, there needs to be a certain number of correct executions.

- **Structure and call-frequency-affecting bugs (call-graph-affecting bugs)**: This distinction is particularly useful when designing call-graph-based defect-localisation techniques. *Structure-affecting bugs* are defects resulting in different structures (topologies) of the call graph where some parts are missing or occur additionally in failing executions. In contrast, *call-frequency-affecting bugs* (*frequency-affecting bugs* for short) are defects which lead to a change in the number of calls of a certain subtree in failing executions, rather than to completely missing or new substructures. In general, it happens frequently that a structure-affecting bug also affects the call frequencies (as a side effect) and vice versa. See Example 2.3 for an illustration of both kinds of defects. We call the class of both kinds of defects, structure and frequency-affecting bugs, also *call-graph-affecting bugs*.

  While the call-graph-based techniques from the related work focus on *structure-affecting bugs*, we develop a defect-localisation technique in Chapter 5 that is able to localise both structure and frequency-affecting bugs.

- **Call-graph and dataflow-affecting bugs**: In contrast to call-graph-affecting bugs, *dataflow-affecting bugs* manifest themselves by infected data exchanged between programme components. In this dissertation, we focus on infected data values exchanged via method-call parameters or return values, e.g., cases where a method returns a wrong value. Dataflow-affecting bugs might affect the call graph as a side effect. Chapter 7 provides more details and examples.

Pure dataflow-affecting bugs are usually not covered by call-graph-based defect localisation, but we present a technique in Chapter 7 which is able to discover both call-graph and dataflow-affecting bugs.

Example 2.3: The graphs in Figure 2.2 are representations from executions of the programme given in Listing 1.1.

Figure 2.2(b) is a call graph where the call of method a from method $main$ is missing, compared to the original graph in Figure 2.2(a). This is an example for a *structure-affecting bug*. The original cause for the infection might be a defective `if`-condition in the main method.

In the graph given in Figure 2.2(c), a defective loop condition or a defective `if`-condition inside a loop in method b are typical causes for the increased number of calls of method a. This is an example for a *frequency-affecting bug*.

2.2.3 Software Testing and Debugging

Software testing is an inherent part of the software development process [Som10]. The overall aim of testing is to ensure that programmes[3] provide the functionality specified before, without eventually leading to any failures. The aim of *debugging* is to find and fix the defects that cause deviations from the specification (failures). *Software quality assurance* (or *validation and verification*), which includes testing and debugging, is a wide field of research of its own. We now briefly explain some fundamental terms and techniques, in order to understand the techniques discussed in this dissertation.

Different Testing Approaches

Testing plays an important role in the whole software-development process, as it takes place at all stages of the process, from coding to final tests before the software is released. In the different stages of the software-development process, one does *unit testing*, *component testing*, *integration testing* and *system testing* [Bei90]. *Unit testing* takes place during coding, and it ensures that the smallest testable pieces of a programme produce the expected results. *Component testing* does the same for larger agglomerations of units. *Integration testing* ensures the correct functionality of several components. *System testing* looks at the functionality of a whole software system. This can consist of one large programme or of a collection of programmes; we consider mainly the first case in this dissertation, as each programme leads to its own call graph.

Regression testing is performed when previous versions of a programme are available, along with their tests [Bei90]. When only small parts from a programme are changed between two versions, one can expect that most tests from the old version

[3]The programme examined is typically called *programme under test*.

pass in the new version as well. Only where functionality was changed between the versions, tests are supposed to fail – all other failing tests can be hints for real failing behaviour in a programme.

In this dissertation, we rely on *system tests* examining the executions of entire programmes. However, when sets of call graphs from a smaller component than the whole programme can be derived, there are no principal obstacles in applying the call-graph-based techniques developed in this dissertation. In real-world software projects, *regression tests* will typically be used to perform system tests (and can be used to drive our defect-localisation techniques), as tests from previous versions are frequently available.

Performing Software Tests

Software tests are formal procedures, consisting of programme inputs and expected outputs [Bei90]. The programme inputs include system configurations, programme parameters and files and user input red by the programme. The expected output includes everything that is produced by the programme, such as files written and output displayed on screen. Designing software tests is its own field of research, typically aiming at covering many different (but non-overlapping) executions of a programme which execute possibly large parts of the source code.

To derive the expected output and to compare it with the actual output, one typically relies on *test oracles* [How78]. Their purpose is to decide whether a certain execution yields any observable problems, i.e., failures. Such an oracle can be a programme that produces the correct result and compares it to the output from the programme under test, or it can be the data itself that the programme under test is supposed to calculate, e.g., calculated manually. Besides unexpected output, other kinds of observable problems such as deadlocks can be considered to be a failure. Test oracles should be able to detect such behaviour as well.

In this dissertation, we assume that both test cases and test oracles are available, as we focus on the later defect-localisation step. This assumption is reasonable, since testing is an inherent part of modern software development [Som10]. Furthermore, in most software projects, (regression) system tests are available, including both test cases and test oracles.

Debugging

Debugging is as well its own field of research [Zel09]. It includes everything from dealing with test cases, observing programme executions, localising defects and ultimately fixing them. It has also been described as the process of relating a failure to an infection to a defect [Zel09].

In this dissertation, we develop techniques for the defect-localisation part of debugging. This is, we aim at helping software developers in localising defects in order to fix them once a failing behaviour has been experienced.

2.3 Data Mining

We now introduce the foundations of data mining that are relevant in this dissertation. We discuss the data-mining process and applied data mining (Section 2.3.1), selected data-mining techniques for tabular data (Section 2.3.2) and finally frequent-pattern-mining techniques, including graph mining (Section 2.3.3).

2.3.1 The Data-Mining Process and Applied Data Mining

The literature has proposed a number of data-mining process models (e.g., [CCK+00, FPSS96]). Sometimes, the term *data mining* stands for a single step within a larger framework, frequently called the process of *knowledge discovery in databases*.

The CRISP-DM Data-Mining Process Model

A well-known representative of data-mining process models is CRISP-DM (CRoss-Industry Standard Process for Data Mining) [CCK+00], which has been proposed by an industry consortium. It describes an iterative process with a number of loops going back to earlier stages: *business understanding*, *data understanding*, *data preparation*, *modelling* (the actual data-mining step), *evaluation* and *deployment* (see Figure 2.3). This process illustrates that data mining or knowledge discovery consists of a number of stages apart from the actual modelling: At first, one needs an understanding of the business or the domain of the application. Then, one needs to understand the data one is working with or one plans to collect. Next, one needs to prepare the data in order to be suited for the data-mining algorithm chosen. Only when the chosen algorithm leads to well evaluation results, the whole process can be deployed.

In this dissertation, the first five steps from the CRISP-DM process model are of relevance. More specifically, the main contribution of this dissertation is not only in the *modelling* part, but also in the *data-preparation* part of the process:

- *Business understanding*: At first, we have to develop an understanding for the principles of software technology and the nature of the various defects, infections and failures.
- *Data understanding*: When we know the domain, we have to understand which data is available and – as we are not faced with an industry project where the aim is to analyse data that is already available – which data we can collect, e.g., from programme executions.

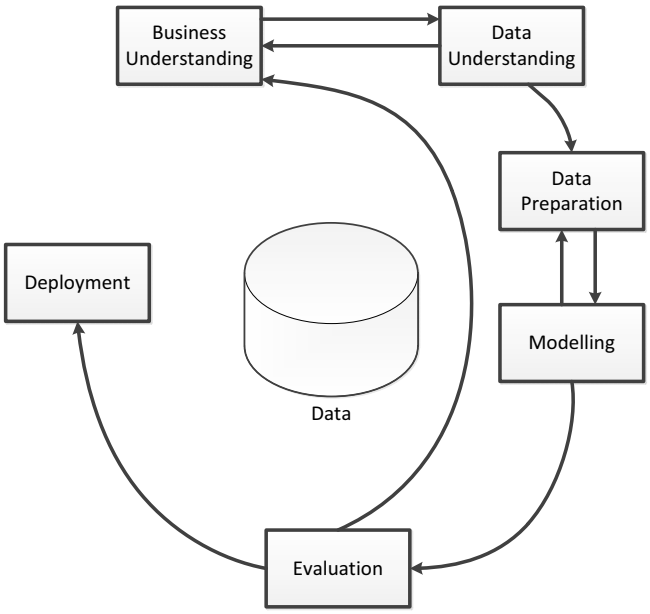


Figure 2.3: The CRISP-DM data-mining process model [CCK+00].

- *Data preparation*: When we know which data is available and can be collected, we have to decide how to represent the data. In this dissertation, we develop a number of call-graph representations.

- *Modelling*: Depending on the data representation chosen, we can chose – or develop – an analysis technique or a combination of different techniques, such as frequent subgraph mining and feature selection.

- *Evaluation*: When all previous steps are done, we have to evaluate our approach consisting of all previous steps.

The next step would be to *deploy* the whole process, possibly in an industrial environment. However, this part of the process is not in the focus of this dissertation.

Applied Data Mining

Besides research on the general data-mining process model, an increasingly important direction of research is *applied data mining*, also called *domain-specific mining* [HG08] or *domain-driven data mining* (D^3M) [CYZZ10]. This direction of research partly builds on the recognition that data-mining research in the past has mainly focussed on building new, faster and more precise techniques, and that relatively little attention has been paid for actual real-world applications [CYZZ10]. Another important recognition is that not much attention has been paid on the integration of

sophisticated scientific and engineering domain knowledge. Thus, specific data representations as well as dedicated analysis techniques are deemed to be essential for the success of applied data mining in any domain [HG08].

This dissertation is in the field of applied data mining in software engineering. The call-graph representations developed in this dissertation are specific data representations that incorporate domain knowledge relevant for the analysis problem. The analysis techniques developed – either as a combination of existing techniques or as a new analysis technique – are specific for the data representations developed beforehand.

2.3.2 Data-Mining Techniques for Tabular Data

Most (conventional) data-mining techniques deal with *tabular data*, i.e., the data to be analysed is stored in tables as in relational databases, and one tuple (a row in the table) refers to one object in the real world. A typical example is customer data, where one tuple refers to one person, and the columns describe numerical or categorical properties such as age, gross income and sex. Based on such data, various data-mining tasks can be defined, such as *classification*, *regression* and *cluster analysis* – for every task there are many different algorithms and implementations available (see, e.g., [BBHK10, HK00, HMS01, Mit97, WF05]). In classification, the task is to predict an unknown categorical class of a tuple, e.g., if a person is creditworthy or not, based on a collection of data from the past. In regression, the task is to predict a numerical attribute. In cluster analysis, the task is to group (or partition) the tuples into previously unknown groups (or partitions) of tuples that share the same properties and have properties different from the other groups (or partitions).

Besides the big success of the data-mining tasks and techniques dealing with tabular data, not every kind of real-world objects can adequately be described using tuples in such a table. As an example, chemical molecules can intuitively be described as a graph structure, where atoms are the nodes, and bindings are the edges of a labelled graph. Of course, based on such a representation, a number of measures can be derived and can be stored in a tuple of numerical and categorical values. As an example, one could derive the number of nodes, the information whether the graph contains cycles and maybe as well the total weight of a molecule. However, despite that such a representation might be suited for certain applications, it does not keep all information encoded in the corresponding graph representation. Therefore, the tabular representation might not be suited for certain applications, or might not allow deriving a certain precision of analyses. The same applies to the software-engineering domain, where call graphs can represent a programme execution more adequately than a table that contains, e.g., different measures corresponding to information such as the number of methods called during an execution.

In this dissertation, we rely on techniques for the analysis of graphs (see Section 2.3.3) and make use of one conventional data-mining technique, *feature selection*, which we describe in the following.

Feature Selection

Many data-mining techniques suffer from the so-called *"curse of dimensionality"*: They either do not scale well for high-dimensional data, or the data becomes less meaningful with an increasing number of dimensions [BGRS99]. *Feature-selection* techniques can help to reduce the number of dimensions in tabular data. They aim at finding subsets of attributes (tuple elements or columns in a table) that still describe the data well, or they aim at scoring these attributes by assigning them with a score that measures their usefulness. In the following, we focus on such usefulness-scoring-based feature-selection techniques, as they are relevant for the analysis techniques developed in this dissertation.

Typically, the usefulness in feature selection is based on the attributes ability to predict another column, e.g., a categorical class attribute. For the categorical case, this ability is also called *discriminativeness*. Respective measures are frequently used internally in decision-tree-induction algorithms, as they have to decide which attribute is best suited to build the next split on, in order to perform a well classification [BK98, Qui93]. Another source of such discriminativeness measures are techniques from statistics that measure the correlation between attributes. In the following, we introduce the *information gain* ($InfoGain$) and *information-gain ratio* ($GainRatio$) discriminativeness measures [Qui93] as two representatives of feature-selection algorithms with a high relevance in practice:

Definition 2.7 (Information Gain and Information-Gain Ratio (see [Qui93]))
Let D be a data table. C is one column in D that associates each row (tuple) to a class. $\mathbb{D}_C$ is the domain of C, and $D_{C=i}$ denotes the set of rows that belong to the i-th class ($i \in \mathbb{D}_C$). Let A denote any other column different from C, consisting of numerical values. The information gain *($InfoGain$) is a measure based on* entropy *($Info$), and the* information-gain ratio *($GainRatio$) in turn is based on $InfoGain$. Both measures measure the discriminativeness of an attribute A when values $v \in A$ partition the dataset D. The partitioning is done in a way that the $InfoGain$ of A is maximised. This requires a discretisation of A's values into n intervals (see, e.g., [ER97] for more information on the discretisation), where $\mathbb{D}_A$ is the domain of the discrete intervals of A ($n = |\mathbb{D}_A|$). $D_{A \in j}$ denotes the partition consisting of the set of rows of D that belong to the j-th interval of A ($j \in \mathbb{D}_A$). The $GainRatio$ normalises the $InfoGain$ value by $SplitInfo$, which is the entropy ($Info$) of the discretisation of the attribute into intervals:*

$$
\begin{aligned}
Info(D) &:= -\sum_{i \in \mathbb{D}_C} \frac{|D_{C=i}|}{|D|} \cdot \log_2\left(\frac{|D_{C=i}|}{|D|}\right) \\
InfoGain(A, D) &:= Info(D) - \sum_{j \in \mathbb{D}_A} \frac{|D_{A \in j}|}{|D|} \cdot Info(D_{A \in j}) \\
SplitInfo(A, D) &:= -\sum_{j \in \mathbb{D}_A} \frac{|D_{A \in j}|}{|D|} \cdot \log_2\left(\frac{|D_{A \in j}|}{|D|}\right) \\
GainRatio(A, D) &:= \frac{InfoGain(A, D)}{SplitInfo(A, D)}
\end{aligned}
$$

Possible values of $InfoGain$ and $GainRatio$ are in the interval $[0, 1]$. Value 1 means that an attribute discriminates perfectly between classes; at 0, an attribute has no influence on class discrimination. Opposed to $GainRatio$, the maximum value of $InfoGain$ can only be 1 if the distribution of classes in C is equal. In skewed class distributions, the maximum value of $InfoGain$ is lower, even if the attribute discriminates perfectly.

2.3.3 Frequent-Pattern-Mining Techniques

Opposed to tabular-data-mining techniques as introduced in Section 2.3.2, *frequent-pattern-mining techniques* [HCXY07] discover frequent or interesting patterns in databases of, e.g., *itemsets*, *sequences*, *trees* and *graphs*. These techniques can be seen as a hierarchy of mining techniques, as sequences generalise itemsets, trees generalise sequences, and graphs generalise trees. In the following, we introduce these techniques and some of its variations, as well as the foundations of *constraint-based mining*.

Itemset Mining

Itemsets and Association Rules. *Itemset mining* has been introduced in the context of *association-rule mining* [AMS+96] – the probably most prominent example for this task is *market-basked analysis*. In itemset mining, one analyses a database of *transactions*, and a transaction consists of one or more binary *items*. As an example, a supermarket transaction consists of a number of products bought. These products are called *items*. The idea of itemset mining is to identify items that were frequently bought together, where the notion of frequency is given by a user-defined *minimum-support* value ($supp_{\text{min}}$, the *support* might be either measured absolutely or as a ratio or percentage): Find all sets of items that are subsets of at least $supp_{\text{min}}$ transactions in a given database. A famous example [SA96a] for the itemset-mining problem is the discovery of a frequent itemset consisting of beer and diapers: $\{beer, diapers\}$.

In *association-rule mining*, one first generates frequent itemsets before these itemsets are split into association rules. As an example, the aforementioned itemset could be split as follows: $\{diapers\} \rightarrow \{beer\}$, saying that people who buy diapers also buy beer. Besides the support value, association rules have a *confidence* value. This is the probability that an association rule holds, i.e., the number of transactions including all items from both sides of the rule divided by the number of transactions including all items from the left side. The legend says [SA96a] that the confidence of the aforementioned rule could be remarkably high, while the overall support of the rule (the support from the union of both sides) would be relatively low.

Itemset-Mining Algorithms. The first and probably easiest algorithm for itemset mining is the *a-priori algorithm* [AMS+96]. It builds on the idea that the support from a subset from some itemset cannot be smaller than the support from its superset. The algorithm uses this idea in a level-wise approach: It first generates all frequent itemsets that consist of a single item (1-itemset). Then, it uses these 1-itemsets to combinatorially generate all 2-itemsets. These 2-itemset candidates are then searched in the database in order to determine their actual support. The remaining 2-itemsets that are actually frequent are then saved and used to generate all potential 3-itemsets etc. Due to the repeated candidate generation and test for frequency, it is also said that the algorithm follows the *generate-and-test paradigm*.

Despite its relatively simple approach for generating all frequent itemsets, the *a-priori algorithm* [AMS+96] has been criticised as it does not scale well. This is due to the potential high number of costly database scans for determining the actual support of the itemsets. A number of further algorithms try to overcome this challenge, by different data representations and algorithm designs: The Eclat algorithm [Zak00] organises the transaction database into a subset lattice and performs a depth-first search in this data structure. The FP-growth algorithm [HPY00] makes use of a prefix-tree structure (*frequent-pattern tree*, *FP-tree*) for the transaction database. The algorithm then follows a divide-and-conquer approach to derive all frequent itemsets. One of the advantages of the FP-growth algorithm is that it relies on the so-called *pattern-growth method*, which replaces the costly generate-and-test approach: Only itemsets that occur at least once in the database are tested if they fulfil the $supp_{\min}$ criterion. This is done efficiently in a subtree of the FP-tree storing the transaction database.

Quantitative Association Rules. Itemset mining and association-rule mining deal by default with binary data. This is, a certain item is part of an itemset or it is not. In market-basked analysis, as an example, it is not considered whether a certain product is contained in a transaction once or hundred times. *Quantitative association rules* [SA96a] introduce numerical weights to items and discretise this information in order to deal with it in an extended association-rule-mining algorithm.

Sequence Mining

Sequence mining is a generalisation of *itemset mining*: Instead of analysing transactions consisting of sets of items, it analyses sequences of such transactions. In more detail, the task is to find all sequences that are subsequences of at least $supp_{\min}$ sequences in a database of sequences [DP07]. One of the first applications was again market-basket analysis: When one is able to track customer purchases over time, the idea is to find frequent sequences of (sets of) items. For instance, it could be a frequent pattern that some customers first buy a digital camera and a camera bag, sometime later a new lens and later on some filters for the new lens. Other applications include DNA-sequence analysis in biology and log-file analysis from web servers.

The first algorithm for sequence mining, AprioriAll [AS95], is a generalisation of the a-priori algorithm and has been proposed by the same authors. The GSP algorithm [SA96b] is an improvement and a generalisation for hierarchies of items, proposed by the same authors as well.

As GSP [SA96b] still follows the generate-and-test paradigm, it bears the same efficiency problems. Therefore, a number of different sequence-mining algorithms has been developed that aim at overcoming this challenge and/or propose further enhancements. One of the well-known variations discovers *frequent episodes* (i.e., partially ordered collections of events occurring together) instead of frequent subsequences [MTV97]. The SPADE algorithm [Zak01] relies on a vertical database format which allows for an optimised lattice-based search space. A sequence-mining algorithm that follows the *pattern-growth approach* is PrefixSpan [PHMA+04]. The authors have shown that their algorithm performs better than all aforementioned algorithms for sequence mining.

Frequent Subtree Mining

Frequent subtree mining is the next generalisation of sequence mining – or a special case of frequent subgraph mining (as introduced in the following). The idea is to discover frequent subtrees in a database of trees. As there are different kinds of trees (see Section 2.1.2), there are different techniques for mining them:

- *Rooted ordered trees* can be mined with the FREQT algorithm [AAK+02].
- *Rooted unordered trees* can be mined with the HybridTreeMiner [CYM04], with the algorithm Unot [AAUN03] and with uFREQT [NK03] (the two last-mentioned algorithms are based on FREQT).
- *Unrooted unordered trees* can be mined with the FreeTreeMiner [CYM03] and with the the HybridTreeMiner [CYM04] as well. Furthermore, such trees can also be mined with arbitrary graph miners, as trees are special cases of

graphs. In particular, Gaston [NK04] is suited for the analysis of trees, as this graph miner internally mines for trees before it extends them to general graphs.

In general, dedicated tree mining algorithms can (but do not necessarily do) decrease the runtime of mining algorithms compared to the usage of general graph-mining algorithms. Algorithms for rooted ordered trees benefit in general most from the specifics of the respective trees. Algorithms for rooted unordered trees benefit less than those for rooted ordered trees and those for unrooted unordered trees less than those for rooted unordered trees. Besides the tree-mining algorithms mentioned, there are more algorithms dedicated for deviating definitions of subgraph relationships and other special cases. Chi et al. present a comprehensive survey of tree-mining algorithms [CMNK05].

Frequent Subgraph Mining

Problem Definition and Algorithms. *Frequent subgraph mining* is the generalisation of all aforementioned frequent-pattern-mining techniques: Roughly speaking, itemsets are graphs without any edges ($E = \varnothing$), sequences are graphs consisting of paths only, and trees are graphs without cycles. Therefore, graph mining can be used for many applications, but suffers from the NP-complete subgraph-isomorphism problem (see Definition 2.4). As we rely on graph-mining techniques in this dissertation, we define the task more formally than the other pattern-mining techniques mentioned before:

Definition 2.8 (Frequent subgraph mining)
Let $D := \{g_1, ..., g_{|D|}\}$ *be a graph database.* Frequent subgraph mining *is the task of finding all subgraph patterns* $f \in F$ *with a* $support$ *of at least* $supp_{min}$ *in* D*. The support of a graph* f *is* $support(f, D) := |\{g | g \in D \wedge f \subseteq g\}|$*. In short,* $f \in F \iff support(f, D) \geq supp_{min}$.

Frequent subgraph mining (and other frequent-pattern-mining algorithms as well) is often used as a building block of some higher-level analysis task such as *cluster analysis* [AW10a] or *graph classification* [CYH10a]. With the latter, frequent subgraph patterns are mined from a set of classified graphs. A standard classifier is then learned on the subgraph features discovered.

Many algorithms have been proposed for *frequent subgraph mining*. The first algorithms, AGM [IWM00] and FSG [KK01], rely on the generate-and-test paradigm known from the *a-priori algorithm*. They therefore follow implicitly a breadth-first-search strategy. All more recent algorithms rely on a depth-first search. These algorithms include FFSM [HWP03], gSpan [YH02] (see also the following paragraph about pattern-growth algorithms) and its extension CloseGraph [YH03] (see also the section on closed mining), Gaston [NK04], MoFa [BB02] and its extension MoSS

[BMB05]. Four of the more recent algorithms mentioned have been compared experimentally by independent scientists using a number of different datasets [WMFP05]. The result is that Gaston and gSpan are mostly the algorithms with the best runtime behaviour, depending on both, the nature of the graph databases analysed and the memory architecture of the machine used for the execution. We focus on gSpan and its variations in the following, as it performs well and is more widely used in the scientific community than Gaston.

The comparison in [WMFP05] and the survey in [YH06] contain more information about the frequent-subgraph-mining algorithms mentioned. We also look at some more recent algorithms in the related-work chapter (Section 3.2.2).

Pattern-Growth Algorithms. Algorithm 2.1 depicts the basic steps of a generic *pattern-growth-based frequent-subgraph-mining algorithm* [YH06]. The idea is that starting from an empty graph-patten p, the current pattern is in each step extended in several ways by exactly one edge, leading to new frequent subgraphs. They are then processed recursively, corresponding to a depth-first search. Concretely, Lines 1–2 check if the current graph pattern is already contained in the result set, Line 4 adds patterns to the result set, and Line 5 extends the current pattern, leading to a set of frequent patterns P. The algorithm then processes them recursively in Lines 6–7 and stops in Line 9 when P is empty.

Algorithm 2.1 *pattern-growth*$(p, D, supp_{\min}, F)$

Input: current pattern p, database D, $supp_{\min}$
Output: result set F

```
1: if p ∈ F then
2:     return
3: end if
4: F = F ∪ {p}
5: P = extend-by-one-edge(p, D, supp_min)
6: for all p' ∈ P do
7:     pattern-growth(p', D, supp_min, F)
8: end for
9: return
```

Algorithm 2.1 performs a depth-first search, which search space is visualised in Figure 2.4. In this search space, the root is the empty graph ($V = \varnothing, E = \varnothing$). Each other node corresponds to a non-empty graph, and Algorithm 2.1 is called once for each node, while it generates the children of a node and calls itself recursively. The leaves are not extended further due to the $supp_{\min}$ criterion. In the generic Algorithm 2.1, the same graph might be generated several times at different places within the search space. We call such graphs *duplicates*.

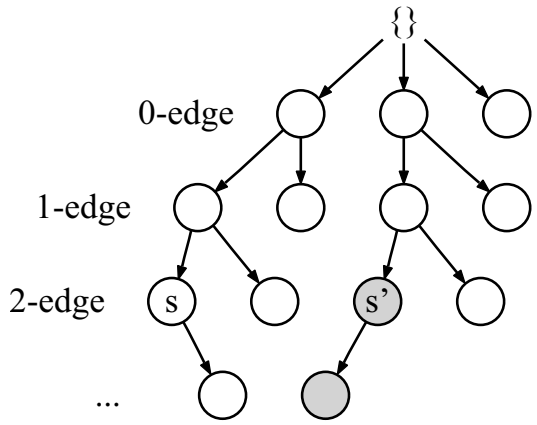


Figure 2.4: A pattern-growth search space.

Example 2.4: Imagine that node s in Figure 2.4 stands for the graph $a \rightarrow b \rightarrow c$ and was generated from the graph $a \rightarrow b$ (its parent node) by extending it with edge $b \rightarrow c$. Node s' stands for graph $a \rightarrow b \rightarrow c$ as well, but was generated from graph $b \rightarrow c$. Node s' is therefore a duplicate. Lines 1–2 in Algorithm 2.1 check for duplicates and prune the search space. Thus, the child from node s' is actually not generated.

Although Lines 1–2 in Algorithm 2.1 identify duplicates which avoids to process the same graphs repeatingly, this check for duplicates is computationally expensive and should be avoided in order to construct a fast algorithm. The extension of graphs should be therefore as conservative as possible, while it still has to guarantee to generate all graphs. Different algorithms use different strategies for this expansion of graphs, and we focus on the strategy from the gSpan algorithm [YH02] in the following, as we use this algorithm in this dissertation.

gSpan [YH02] uses a strategy for generating (extending) graphs that is based on *depth-first search* (*DFS*) in graphs. In general, one can perform different depth-first searches in the same graph, resulting in different *depth-first-search trees* (*DFS trees*). Such a DFS tree can unambiguously be represented as an ordered list of edges (ordered by the discovery time during search). gSpan uses a set of rules for generating (extending) graphs, which relies on the order in a depth-first search and on extending the graph only along the rightmost path in a DFS tree: the *DFS-lexicographic order*. This ensures that one graph is always traversed the same way. When graphs are constructed in this way, it is guaranteed that the frequent-pattern-mining procedure does not extend graphs already discovered.

Closed Mining

Closed mining is an important concept in frequent-pattern mining, as it bears the potential of generating result sets with less redundancy in a faster runtime than non-

closed algorithms. In the following, we define closed mining with graphs, but the concept exists for all frequent-pattern-mining techniques. As two examples, the CloSpan algorithm [YHA03] performs closed sequence mining, and the CMTreeMiner algorithm [XY05] performs closed mining for rooted unordered trees.

Definition 2.9 (Closed Graph Mining)
Closed-frequent-subgraph-mining algorithms *discover only subgraph patterns which are closed. A graph f is closed if and only if no other graph pattern f' is part of the result set F which has exactly the same support and is a proper supergraph of f ($f \subset f'$).*

Closed mining algorithms produce result sets that might be more concise (smaller), for the following reason: The result sets are free of redundancy in the sense that the complete set of frequent subgraphs can be derived from the set of closed subgraphs. In concrete terms, the complete set can be obtained by systematically removing edges (along with the incident nodes when they become unconnected) from all graphs in the closed result set and adding these new graphs to the non-closed result.

The CloseGraph algorithm [YH03] is an extension of gSpan [YH02] that makes use of pruning opportunities and speeds up mining in many situations. However, there are cases where the result set from closed mining is not (or not much) different from the set of all frequent subgraphs. In such cases, the additional effort for checking for closedness and no (or only few) pruning opportunities might slow down the algorithm. In general, the probability for such situations increases with increasing size of the underlying graph database. This is as the probability that two subgraphs have the same support decreases when graph databases increase in size. In this dissertation, we use CloseGraph for mining databases of call graphs. We do so as our graph databases are relatively small, and we therefore do not expect to suffer from the effect described before. Furthermore, in preliminary experiments, CloseGraph has produced defect-localisation results that are not worse than those when employing gSpan, in a runtime that has indeed been faster.

Constraint-Based Mining

Constraint-based mining is another important concept in frequent-pattern mining, as it allows for faster runtimes and result sets focused on the user's needs. However, constraint-based mining requires the user to specify a constraint, and not all constraints can be easily integrated into mining algorithms. Constraint-based mining has originally been introduced for itemset mining [NLHP98], and has been carried forward to sequences (e.g., [GRS99, PHW02]) and graphs (see Section 3.2.3).

Definition 2.10 (Constraint-Based Mining)
A constraint c in constraint-based mining *is a Boolean predicate which any $f \in F$*

must fulfil, where F is the result set. Formally, in constraint-based mining, $f \in F \iff (support(f, D) \geq supp_{min} \wedge c(f) = true)$, where D is the database.

Constraint predicates can be categorised into several classes. We now introduce the most important one, *anti-monotonicity*, and briefly mention some further constraint classes:

Definition 2.11 (Anti-Monotone Constraints (see [NLHP98]))
A constraint c is anti-monotone $\iff (\forall f' \subseteq f : c(f) = true \Rightarrow c(f') = true)$, *where F is the result set.*

Example 2.5: A prominent example of anti-monotone constraints is the frequency criterion: If a graph has a support of at least $supp_{\min}$, all its subgraphs have the same or a larger support. Therefore, anti-monotone constraints are the basis for all a-priori and pattern-growth mining algorithms: They stop extending patterns when the current one does not satisfy the constraint, without missing any patterns.

The class of *monotone constraints* [NLHP98] as a complement to anti-monotone constraints exists as well (but there are constraints that are neither anti-monotone nor monotone). However, monotone constraints are less useful for pruning.

Another class are *succinct constraints* [NLHP98], which are orthogonal to the aforementioned constraint classes. Respective patterns that fulfil such constraints can be enumerated before the support is counted in a graph database.

Example 2.6: In constraint-based *itemset mining*, a *succinct constraint* could be $c(f) := x \in f$, where $f \in F$. This is, only graphs that include item x are supposed to be in the result set. In a-priori-style algorithms, this can be tested before support counting, which speeds up mining significantly.

Another class of constraints are *convertible constraints* [PHL04]. They have been introduced for itemsets, and focus on aggregate constraints that build on functions such as *average*, *median* and *sum*, referring to numeric annotations of the items. These annotations are fixed for every item, no matter in which transaction they occur; an example would be the price of a certain item. Although convertible constraints are less suited to prune the search space, they can be used to speed up the FP-growth algorithm [HPY00] for itemset mining [PHL04].

3 Related Work

This dissertation is about *domain-specific data mining*, in particular about *software defect localisation*. Therefore, we describe related work in the application domain, i.e., various *defect-localisation techniques* (Section 3.1), as well as related data-mining techniques – in particular different approaches for subgraph mining (Section 3.2). We furthermore discuss related work that is closely related to ours in Chapters 4 and 5, i.e., other *call-graph-based defect localisation* techniques.

3.1 Defect Localisation

Defect-localisation techniques are either *static* or *dynamic* [Bin07]. Dynamic techniques rely on the analysis of programme runs while static techniques do not require any execution. An example for a static technique is source-code analysis. It can be based on code metrics or different graphs representing the source code, e.g., *static call graphs*, *control-flow graphs* or *programme-dependence graphs* (see Section 2.2.1). Dynamic techniques usually trace some information during a programme execution which is then analysed. This can be information on the values of variables, branches taken during execution or code segments executed. A further distinction of defect-localisation techniques is the level of granularity: While some techniques identify classes or methods with an increased likelihood to be defective, other techniques identify defects at a finer level of granularity, e.g., statements, lines of code or blocks of statements.

It is worth being mentioned that no defect-localisation technique is perfect in the sense that is is able to localise any kind of defect. A study on comparing different static approaches by Rutar et al. [RAF04] came to the conclusion that none of the tools they have investigated strictly subsumes one of the others. The same applies to dynamic techniques: Santelices et al. [SJYH09] have compared several dynamic approaches and came similarly to the conclusion that no single approach performs best for all kinds of defects. The different defect-localisation techniques described in this section – as well as the ones proposed in this dissertation – can therefore be considered to be orthogonal to each other. A combination of different techniques will probably be the most effective way to do defect localisation in practice.

In the remainder of this section we discuss a selection of different static and dynamic defect-localisation techniques (Sections 3.1.1 and 3.1.2, respectively). We

then briefly introduce some related work on localising defects in multithreaded programmes (Section 3.1.3).

3.1.1 Static Approaches

Mining Software Metrics and Software Repositories

Software-complexity metrics are measures derived from the source code, describing, e.g., the complexity, quality or maintainability of a programme or its methods. The software-engineering community has been very active in defining such metrics [HS95, Jon08], but they are typically not intended to facilitate a defect localisation. However, in many cases, complexity metrics correlate with defects in software [NBZ06, ZNZ08].

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 from a version-management system. Such a mapping has been done, for instance, by Nagappan et al. [NBZ06]. The authors derive standard complexity metrics from source code and build principal-component models based on them and on the information if the software entities considered contain defects. The principal-component models can then predict post-release failures for new pieces of software. However, the authors discover that every project has its specific set of complexity metrics well suited for defect localisation. These sets of metrics can only be used within newer versions of the same project or within very similar projects. In particular, there is no universal set of metrics or even a single metric that is suited for defect predictions for any software project.

Studies related to the one of Nagappan et al. [NBZ06] are, for instance, the ones by Knab et al. and Schröter et al. Knab et al. [KPB06] use decision trees to predict failure probabilities. The approach by Schröter et al. [SZZ06] uses regression techniques to predict the likelihood of defects based on static usage relationships between software components.

All these approaches rather give hints on code quality issues than pinpointing actual defects. Furthermore, they require a large collection of defects and version history. Concerning the level of granularity, different software-repository-mining approaches focus on different levels of abstraction. However, many complexity metrics are defined at the method level, thus deriving defect localisations at this level of granularity.

Syntactic Defect-Pattern Detection

As a number of typical defect-prone programming patterns are known, there is a number of tools that heuristically search for syntactic defect patterns [RAF04]. FindBugs [AHM+08] from Ayewah et al., for instance, is a well-known representative. It anal-

yses static Java source code and generates a number of warnings presented to the user. The tool can seamlessly be deployed within integrated development environments (IDEs) such as eclipse. However, FindBugs typically does not find more sophisticated logical defects and it frequently produces a sheer number of warnings, leading to a high rate of false positives [RAF04]. Nevertheless, FindBugs can help to discipline programmers writing less defect-prone code, and the tool has been successfully used in a large-scale industrial setting [AP10]. FindBugs delivers defect localisations at a very fine granularity, identifying statements or lines which might be defective.

Mining of Programme-Dependence Graphs (PDGs)

The work of Chang et al. [CPY08] focuses on discovering *neglected conditions*. They are a class of defects which are in many cases *non-crashing occasional bugs*. An example of a neglected condition is a forgotten `case` in a `switch` statement. Chang et al. work with static *programme-dependence graphs* (*PDGs*, see Section 2.2.1) and utilise graph-mining techniques. PDGs are graphs describing both control and data dependencies (edges) between elements (nodes) of a method or of an entire programme.

The idea behind [CPY08] 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. To put these ideas into practice, the authors develop a *heuristic frequent subgraph-mining algorithm* and apply it to a database of PDGs. In their approach, an expert has to confirm and possibly edit the 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. This leads to fine-grained defect localisations at the statement-level.

From a technical point of view, it is notable that there are no guarantees for the two 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, but are not compared to related work. Though graph-mining techniques similar to dynamic-call-graph mining as investigated in this dissertation are used in [CPY08], the approaches are not related. The work of Chang et al. relies on static PDGs. They do not require any programme execution, as dynamic call graphs do.

3.1.2 Dynamic Approaches

Many dynamic defect-localisation approaches have been evaluated with a set of small C programmes, ranging from 200 to 700 lines of code (LOC), which were originally introduced by Siemens Corporate Research [HFGO94]. These so-called *Siemens Programmes* provide a number of artificially introduced defects along with a number of test cases. They can be seen as a standard benchmark, although the programmes are rather small and the defects are realistic but artificial.

Delta Debugging

Delta debugging is a general strategy invented by Zeller [Zel99] for systematically searching for causes of failures, following the trial-and-error principle. It does so by determining the relevant difference between two *configurations* with respect to a given test. A *configuration* in this context can be, e.g., a programme input, user interactions, a thread schedule, code changes or programme states.

Looking at delta debugging with programme inputs as an example [ZH02], one searches for the minimal difference between an input that leads to a failure and an input that leads to a correct execution. To this end, one has to provide a test oracle (see Section 2.2.3) that decides whether a programme execution is correct or failing, as well as a failing and a passing programme input. Delta debugging then finds two programme inputs leading to correct and failing results with a minimal difference. This information can be used to ease manual debugging. As an example, when the programme investigated is a compiler and the input data is some source code, the difference in the input source code is probably related to some statement. The defect is then likely to be located in the parts of the compiler handling this kind of statement.

Multithreaded software introduces indeterminism to a programme execution (see Section 3.1.3). This is, there are a huge number of possible thread interleavings, and failures might only occur when internally a certain interleaving is executed. In [CZ02], the authors present a delta-debugging approach which is able to identify failure-inducing thread interleavings. In concrete terms, they use the DEJAVU capture/replay tool [CS98] in order to record the thread interleaving and to replay it deterministically. By systematically varying these replays, delta debugging localises infections, i.e., locations where a thread switch causes the programme to fail. This gives hints where the actual defect might be located within the source code, without directly pinpointing this location. However, Tzoref et al. [TUYT07] have shown that approaches building on varying thread interleavings and delta debugging do not scale for large software projects.

Similarly to programme inputs and thread interleavings, delta debugging has been applied to programme changes [Zel99]: When one version of a programme is available that executes correctly and a version that fails, e.g., from a version-control sys-

tem, delta debugging can reveal the actual change that causes the failing behaviour. This requires the availability of different versions from the same programme.

The delta-debugging technique which setting is probably closest to the defect-localisation approaches developed in this dissertation, is [CZ05]. It does not rely on programme inputs, different thread interleavings or programme versions, but requires a programme with an *occasional bug* along with respective test cases only. The technique extends earlier work of the authors [Zel02]: It applies delta debugging to programme states, represented by the current variable values. To this end, it represents the variable values by means of so-called *memory graphs*. Then, the approach systematically modifies the *memory graphs*, i.e., the programme states of running programmes, using a debugger. To do so, it employs the delta-debugging strategy to compute minimal differences of *memory graphs* of correct and failing executions, i.e., variables. In [CZ05], the authors then investigate *cause transitions*. They provide a means to localise the defect in the source code which later leads to the infected variable identified by delta debugging on *memory graphs*. This leads to defect localisations at the fine-grained granularity level of statements.

The authors evaluate the approach based on cause transitions and delta debugging [CZ05] with the *Siemens Programmes*. It outperforms another more basic defect-localisation approach. However, as we will see in the following, different complemental dynamic approaches outperform delta debugging on the same benchmark programmes.

From Coverage Analysis to Sequence Analysis

Statement-Coverage Analysis. Coverage analysis can be seen as the basis for many dynamic approaches, including the call-graph-based ones discussed in this dissertation. Tarantula from Jones et al. [JHS02] is such a technique, using tracing and visualisation. To localise defects, it utilises a ranking of programme components which are executed more often in failing executions. This is then used to visualise the source code for the programmer, using different colours and intensities. In more detail, a programme component is a *basic block* in a *control-flow graph* (see Section 2.2.1), i.e., a sequence of statements always executed conjunctively. Tarantula calculates the defect-likelihood for a basic block e as follows:

$$P_{\text{Tarantula}}(e) := \frac{\frac{failed(e)}{totalfailed}}{\frac{passed(e)}{totalpassed} + \frac{failed(e)}{totalfailed}}$$

where $passed(e)$ is the number of correct executions that have executed basic block e at least once, and $failed(e)$ similarly refers to failing executions. $totalpassed$ and $totalfailed$ are the total numbers of programme executions that are correct and failing, respectively.

For the source-code visualisation, Tarantula uses different colours depending on the suspiciousness value $P_{\text{Tarantula}}$. For instance, source code with value 1 is visualised in red. For a further differentiation of the visualisation, Tarantula uses an additional $brightness$ score. However, in the experiments by the authors, they only use the suspiciousness value $P_{\text{Tarantula}}$ [JH05, JHS02] to rank the statements. The $brightness$ score is calculated as follows:

$$brightness(e) := \max(\frac{passed(e)}{totalpassed}, \frac{failed(e)}{totalfailed})$$

While the Tarantula technique is relatively simple, it produces good defect-localisation results. In an evaluation conducted by the authors [JH05] based on the *Siemens Programmes*, it has outperformed five competitive approaches, including a *delta-debugging* approach [CZ05]. However, it does not take into account how often a statement is executed within one programme run. This might miss certain defects such as *frequency-affecting bugs*. In general, Tarantula derives defect-localisations at a basic-block level, but the authors also describe how to map these results to the method level [JH05].

Abreu et al. [AZGvG09] aim at improving Tarantula [JHS02] by evaluating different scoring functions besides $P_{\text{Tarantula}}$ within the same framework. Most importantly, they have investigated the *Jaccard coefficient* known from statistics and the *Ochiai coefficient* which is typically used in molecular biology:

$$\begin{aligned} P_{\text{Jaccard}}(e) &:= \frac{failed(e)}{totalfailed + passed(e)} \\ P_{\text{Ochiai}}(e) &:= \frac{failed(e)}{\sqrt{totalfailed \cdot (failed(e) + passed(e))}} \end{aligned}$$

Based on experiments with the *Siemens Programmes*, Abreu et al. [AZGvG09] have found that the *Jaccard coefficient* performs better than Tarantula and the *Ochiai coefficient* performs even better than the *Jaccard coefficient*.

Sequence-Analysis Approaches. Dallmeier et al. present AMPLE [DLZ05]. The technique refines coverage analysis and analyses sequences of method calls. The authors demonstrate that the temporal order of calls is more promising to analyse than statement coverage only. More concretely, AMPLE compares object-specific sequences of incoming and outgoing object calls, using a sliding-window approach. Then, it derives a ranking at the granularity level of classes, which is much coarser than methods, basic blocks or statements. This ranking is based on the information which objects (i.e., instances of classes) differ the most between passing and failing runs regarding their statement sequences.

A fairly recent approach is RAPID from Hsu et al. [HJO08]. It directly extends the Tarantula approach [JHS02]. RAPID first calculates $P_{\text{Tarantula}}$ values for all statements and then filters all statements having a value of less than 0.6. Based on the remaining statements having an increased likelihood to be defective, it derives maximum common subsequences in the programme execution traces. To this end, RAPID utilises the BIDE sequence-mining algorithm [WH04]. Finally, RAPID presents the sequences to the user, starting with those containing the highest ranked statements according to $P_{\text{Tarantula}}$. This aims at providing contextual information referring to execution sequences for the developer, making defect localisation easier than only providing possibly defective statements or lines. However, even though the technique seems to be promising, to our knowledge, it has never been evaluated comprehensively.

Lo et al. [LCH+09] also deal with sequences, but present a *failure-detection approach*. This is, it does not localise defects, but decides whether an execution is correct or not.

Dataflow-Path Analysis. Masri [Mas09] proposes a dataflow-focused approach which has some similarities to sequence analysis. He performs a dynamic analysis of dataflows between statements to detect defects in source code. To this end, he works with dataflow paths, which are similar to sequences. They comprise frequency, source and target types (e.g., branch, statement) and the length of the executed dataflow path. Specifically, Masri compares sub-paths of dataflows of correct and failing executions to rank defect positions at the granularity level of statements, with a mechanism similar to the one in [JHS02]. However, Santelices et al. [SJYH09] describe that the monitoring of dataflows as done by Masri [Mas09] is much more expensive than more lightweight approaches such as Tarantula [JHS02]. Dataflow-enabled call graphs as proposed in Chapter 7 cover dataflow information besides the control-flow-related call-graph structure. However, this is done differently than in [Mas09]. the approach by Masri [Mas09] is therefore complementary to the work presented in this dissertation.

Subsumption. Both approaches, *statement-coverage analysis* and *sequence analysis*, can be seen as a basis for the more sophisticated call-graph-based techniques we focus on in this dissertation: The usage of sequences instead of statement coverage is a generalisation which takes more structural information into account. This structural information, also referred to as a *context*, eases the manual debugging process [HJO08]. Call-graph-based techniques in turn cover more complex structural information (encoded in the graphs) than sequences. Likewise, a *subgraph context* provided besides a more fine-grained defect localisation, likely eases the manual debugging activities.

Statistical Defect Localisation

Statistical defect localisation is a family of dynamic techniques which make use of more detailed information than *coverage-analysis techniques*. Such techniques are based on instrumentation of the source code, which allows capturing the values of predicates or other execution-specific information, so that patterns can be detected among the variable values. Daikon from Ernst et al. [ECGN01] uses such an approach to discover programme invariants. This problem is somewhat different from defect localisation and can therefore hardly be compared to such techniques. However, the authors claim that defects can be detected when unexpected invariants appear in failing executions or when expected invariants do not appear.

The Approach from Liblit et al. Liblit et al. [LNZ+05] rely on the statistical analysis of programme predicates, building on earlier work from the authors [LAZJ03], which uses predicates and regression techniques. The more recent approach considers a large number of Boolean programme predicates, most importantly predicates that are evaluated within condition statements (e.g., `if`, `for`, `while`) and predicates referring to return values of functions. Concretely, return-value predicates indicate whether the returned value is < 0, ≤ 0, > 0, ≥ 0, $= 0$ or $\neq 0$. For each predicate in a programme, the authors calculate the likelihood that its evaluation to `true` correlates with failing executions. This is used to rank predicates. Predicates are typically used to perform decisions relevant for control-flow branches within a programme. Therefore, predicates can be mapped to basic blocks [SJYH09], and predicate-based analysis is a fine-grained defect-localisation technique.

Liu et al. [LFY+06] have shown with experiments using the *Siemens Programmes* that [LNZ+05] performs constantly better than *delta debugging* [CZ05], and that it performs similarly as Tarantula [JHS02]. Despite these good results, the approach from Liblit et al. [LNZ+05] inherently bears the risk to miss certain defects: As the likelihood calculation only considers whether a predicate has at least once been evaluated as `true` in a programme execution, it might not localise *frequency-affecting bugs*. In particular, if a predicate is evaluated to `true` at least once in every execution, the method considers the predicate to be completely unsuspicious.

The SOBER Method. Liu et al. propose a similar approach, called SOBER [LFY+06], that overcomes some problems of [LNZ+05]. It makes use of a subset of the predicates analysed in [LNZ+05], which includes the condition-statement and return-value predicates discussed before. It then uses a more sophisticated statistical method to calculate defect likelihoods: It models the predicate evaluations to `true` and `false` in both correct and failing programme executions and uses the model difference as the defect likelihood of predicate p as follows:

$$P_{\text{SOBER}}(p) := -\log(L(p))$$

where L is a function which calculates the similarity of the predicate evaluation models. See [LFY+06] for all details on how the similarity functions are chosen and P_{SOBER} is calculated exactly.

The authors show that the SOBER method is able to detect the infections identified by suspicious invariants mentioned before [ECGN01] as well. As SOBER uses predicate analysis, the granularity is as in [LNZ+05] the level of predicates or basic blocks. The evaluation conducted by the authors [LFY+06] based on the *Siemens Programmes* has shown that SOBER performs almost constantly better than Tarantula [JHS02] and the approach from Liblit et al. [LNZ+05] and as well constantly better than *delta debugging* [CZ05].

Subsumption. Opposed to the call-graph-based techniques discussed in this dissertation, the *statistical-defect-localisation* approaches do not take structural properties of call graphs into account. Hence, detecting *structure-affecting bugs* is more difficult.

Another known issue is that statistical defect localisation might possibly miss some defects. This is caused by the usual practice (e.g., as done in [LAZJ03, LNZ+05]) not to observe every value during an execution, but to consider sampled values. [LAZJ03] partly overcomes this issue by collecting information from productive code on large numbers of machines via the Internet. However, this does not facilitate the discovery of defects before the software is released.

Compared to the dataflow-analysis approach in call graphs proposed in Chapter 7, Liblit et al. [LNZ+05] and SOBER by Liu et al. [LFY+06] consider only three intervals for return values of methods, i.e., $(-\infty, 0)$, $[0, 0]$ and $(0, \infty)$. A variable number of dynamically identified intervals might be better suited to capture defects that do not manifest themself in the fixed intervals given. Furthermore, [LNZ+05, LFY+06] do not consider dataflows in the method-call parameters, which might contain important defect-related information, too.

Defect Localisation with Graphical Models

A fairly recent technique is the application of *graphical models* to defect localisation. Graphical models are a machine-learning technique relying on statistics, bringing together concepts from graph theory and probability theory [Jor99]. Well-known representatives of graphical models are *Bayesian networks*, also known as *directed acyclic graphical model* or *belief network* [Jen09].

Dietz et al. [DDZS09] make use of graphical models and apply them for defect localisation. They train so-called *Bernoulli graph models*, with data obtained from programme executions. More concretely, the authors generate models for every method of a programme execution, where nodes refer to statements within the methods. Once the models are generated, the authors use them for Bayesian inference to calculate

probabilities of transitions between the nodes in a new programme execution. Based on these probabilities, they derive defect localisations at the statement level.

The authors evaluate their technique with real defects from large software programmes, originating from an early version of the iBUGS project [DZ09]. In the experiments, they outperform Tarantula [JHS02] almost consistently. However, the evaluation covers only situations where a software developer has to investigate up to 1% of the source code in order to find the defect. – The study does not cover the localisation of defects that are harder to detect, i.e., where one has to investigate more than 1% of the code.

Dynamic Programme Slicing

Dynamic programme slicing [KL88] can be very useful for debugging although it is not exactly a defect-localisation technique. It helps searching for the exact cause of a failure, i.e., the defect, if the programmer already has some clue which parts of the programme state are infected or knows where the failure appears, e.g., if a stack trace is available. Programme slicing gives hints which parts of a programme 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 failure appeared.

3.1.3 Defect Localisation in Multithreaded Programmes

Multicore computers with several cores on a single chip have become ubiquitous. They provide developers with new opportunities to increase performance, but applications need to be multithreaded to exploit the hardware potential [Pan10]. One drawback of multithreaded software development, compared to the sequential case, is that programmers are additionally confronted with non-determinism and parallel-programming errors. Non-determinism arises as the operating system might assign different thread schedules to different executions of the same programme [CS98]. Parallel-programming errors such as atomicity violations, race conditions (i.e., uncontrolled concurrent access of memory objects from different threads) and deadlocks [FNU03, LPSZ08] are frequently triggered by this effect.

In this dissertation, our focus is on the localisation of defects in sequential programmes. However, as described above, multithreaded programming leads to new kinds of defects, and multithreaded programmes seem to be more defect-prone than sequential software. Therefore, we briefly summarise the most important research directions in the field of defect localisation in multithreaded programmes in the following. Concretely, we comment on a selection of static and dynamic approaches and conclude this section with a brief subsumption.

Static Approaches

Tools employing static analysis, such as ESC/Java by Flanagan et al. [FLL+02] or RacerX by Engler and Ashcraft [EA03] investigate the source code without execution, but – similarly to the single-threaded case – might produce large numbers of false-positive warnings. Furthermore, some tools such as ESC/Java require programmer annotations to reduce the number of warnings, which are tedious to create.

The intuition behind many static approaches is to discover situations in which variables or objects are accessed concurrently without explicit monitoring. This possibly leads to race conditions. ESC/Java, for instance, lets the user specify which data objects should always be accessed in a controlled way. The tool then generates predicates from the annotations, and a theorem proofer derives whether these predicates hold for the entire source code. This information then is used to derive warnings. RacerX does not rely on annotations, but employs a set of heuristics to decide whether a memory access might be accidentally unmonitored. While such an approach is more convenient to use, it also tends to produce more false positive warnings.

Dynamic Approaches

Dynamic race detectors such as Eraser by Savage at al. [SBN+97] instrument programmes and analyse the runtime behaviour of the memory access of each thread. Eraser, for instance, monitors the locks each thread currently holds. Based on observed sets of locks per variable, it identifies situations that possibly lead to race conditions. Other dynamic approaches rely on the analysis of *happens-before relations:* When no synchronisation constructs protect a read and a write access or two write accesses from two threads, a race condition is likely. To derive such happens-before relations, logical *Lamport clocks* [Lam78] and *vector clocks* [Mat89] have been used. Hybrid race detectors such as the one by O'Callahan and Choi [OC03] combine different dynamic techniques to improve race detection. The IBM MulticoreSDK by Qi et al. [QDLT09] is an implementation of this approach. It also makes use of some static analysis: It analyses the source code to improve the runtime of dynamic analysis by identifying variables that can safely be excluded from further consideration. However, deriving the information needed for the dynamic approaches at runtime implies a possibly huge overhead. Further, dynamic approaches can influence a programme under test and change its timing, which can make a race condition disappear. This effect is known as the *probe effect* [Gai86].

Another general problem of dynamic race detectors is that a race might manifest itself only when certain thread schedules occur. As scheduling is done by the operating system, developers have limited influence on reproducing a race. Addressing this problem, ConTest by Farchi et al. [FNU03] executes a multithreaded Java programme several times and influences thread schedules by inserting certain statements

(e.g., `sleep()`) into a programme. Chess, developed by Musuvathi et al. [MQB07] for C#, has an additional refinement: a modified thread scheduler exhaustively tries out every possible thread interleaving. On top of that, a *delta-debugging* strategy [Zel99] as described in Section 3.1.2 can be used to automatically localise a defect. Such an approach has been followed, e.g., in [CZ02]. However, as mentioned before, Tzoref et al. [TUYT07] have shown that such approaches do not scale well.

Subsumption

All of the tools mentioned in this section on defect-localisation in multithreaded programmes focus on identifying atomicity violations, race conditions or deadlocks. These tools are specialised on a particular class of parallel programming errors that are due to wrong or missing usage of synchronization constructs in parallel programming languages. However, failures of multithreaded programmes might have other causes, too. For instance, they might originate from *non-parallel* constructs that trigger wrong *parallel* programme behaviour.

Example 3.1: Suppose that a programmer forgets or incorrectly specifies a condition when she or he writes the code creating threads in a thread pool. This slip affects parallel behaviour and might lead to an unbounded creation of threads, wrong control flow and incorrect programme outputs.

Such situations might be better tackled by analysing anomalies of executions such as differences between call graphs from correct and failing executions of a programme. In this dissertation, we discuss some ideas concerning call-graph representations for multithreaded programmes in Section 4.3, and we present the results from a first study on defect localisation with such graphs in Appendix A. We furthermore come to the conclusion that the field of defect localisation with multithreaded programmes bears much potential for future investigations. We present some ideas in Section 4.3 and Chapter 9.

3.2 Data Mining

In this section, we discuss related data-mining techniques. In particular, we consider *weighted subgraph mining* (Section 3.2.1), *mining significant subgraphs* (Section 3.2.2) and *constraint-based subgraph mining* (Section 3.2.3).

3.2.1 Weighted Subgraph Mining

Weighted graphs are ubiquitous in the real world. For instance, think of transportation networks, where numerical weights attached to edges might stand for the load, the average speed, the time, the distance etc. As well software call graphs as investigated

in this dissertation for defect localisation can be attached with weights: Edge weights might represent call frequencies or abstractions of the dataflow. However, we are only aware of a few studies analysing weighted graphs with frequent subgraph mining. Most studies focus on the specific analysis problem, rather than proposing general weighted-subgraph-mining techniques. In the following, we review some work based on discretisation, and we discuss approaches building on the concept of *weighted support*.

Discretisation-Based Approaches

Logistic Networks. Jiang et al. [JVB+05] investigate frequent subgraph mining in logistic networks where edges represent single transports and are annotated with several weights such as distance between two nodes and the weight of the load. With each weight, a different weighted graph can be constructed. In order to derive labels which are suited for graph mining from the edge weights, the authors use a binning strategy. Each weight is partitioned into ranges of the same size, giving a few (7 to 10) distinct labels. The binning strategy for discretisation may curb result accuracy, for two reasons: (1) The particular scheme does not take the distribution of values into account. Thus, close values may be assigned to different bins. (2) The discretisation leads to a number of ordered (ordinal) intervals, but the authors treat them as unordered categorical values. For example, the information that 'medium' is between 'low' and 'high' is lost.

Image Analysis. Nowozin et al. [NTU+07] do discretisation as well before it comes to frequent subgraph mining. They study image-analysis problems, and images are represented as weighted graphs. The authors represent each point of interest by one vertex and connect all vertices. They assign each edge a vector consisting of image-analysis-specific measures. Then they discretise the weights, but with a method more sophisticated than binning. The weight vectors are clustered, resulting in categorical labels of edges with similar weight vectors. However, the risk of losing potentially important information by discretisation is not eliminated: (1) It might still happen that close points in an n-dimensional space fall into different clusters. (2) Even when value distributions are considered, the authors do so in the context of the original graphs. When frequent subgraph mining is applied afterwards, the distributions within the different subgraphs can be very different, and other discretisations could be more appropriate.

Subsumption. In this dissertation, we deal with software call graphs that are weighted. For the analysis of these graphs, we propose two kinds of approaches that are different from discretisation: In Chapters 5–7 we investigate a postprocessing approach, in Chapter 8 a constraint-based mining approach. Both proposals avoid the

shortcomings of discretisation mentioned. They analyse numerical weights instead of discrete intervals.

Weighted-Frequent Subgraph Mining

The Approaches by Jiang et al. Jiang et al. [JCSZ10] deal with a text-classification task, formulated as a *weighted-frequent-subgraph-mining problem*. This is based on the concept of *weighted support* formulated by the authors. This concept builds on the assumption that certain edges within a graph are considered to be more significant than others, and that the significance is reflected in the edge-weight values (i.e., a significant edge displays a high value)[1]. Concretely, the authors calculate the *weighted support* $wsup$ of a subgraph g as follows:

$$wsup(g) := sup(g) \cdot \sum_{e \in E(g)} w(e)$$

This is, the *weighted support* of a certain subgraph is high when it has a high support and contains edges having high weight values. Correspondingly, *weighted-frequent-subgraph mining* as defined by the authors discovers subgraphs satisfying a certain user-defined *minimum-weighted-support threshold*. However, the *minimum weighted support criterion* is not *anti-monotone* and can therefore not be used to prune the search space in pattern-growth-based frequent-subgraph-mining algorithms. The authors therefore make use of an alternative but weaker concept to prune the search space and implement their technique as an extension of gSpan [YH02] (see Section 2.3.3). In [JCZ10], the authors present variations of the approach, including two further weight-based criteria that are *anti-monotone*.

Using their approaches, the authors achieve well results not only in the text-classification application [JCSZ10], but also applied to (medical) image-analysis problems [ECJ+10, JCSZ08] and certain problems from logistics [JCZ10].

The Approaches from Shinoda et al. Shinoda et al. [SOO09] present an approach similar to the ones from Jiang et al. [JCSZ10, JCZ10]. They consider graphs with weighted nodes and edges (referred to as *internal weights*), and their graphs themself are assigned with a weight as well (referred to as *external weights*). They define the *internal weighted support* $wsup_{\text{int}}$ similar to Jiang et al., but they consider the total internal weight of the graph database D (in the denominator):

$$wsup_{\text{int}}(g) := \frac{\text{sum of all internal weights of } g \text{ in all graphs } d \in D \text{ where } g \subseteq d}{\text{sum of all internal weights of all graphs in } D}$$

If there are several embeddings of $g \in d$, the one with the maximum weight is chosen.

[1] Jiang et al. consider only weighted edges, but claim that their concepts can be easily transferred to weighted nodes.

The authors define the *external weighted support* $wsup_{\text{ext}}$ similarly as follows:

$$wsup_{\text{ext}}(g) := \frac{\text{sum of the external weights of all graphs } d \in D \text{ where } g \subseteq d}{\text{sum of all external weights of all graphs in } D}$$

Finally, they define a *general weighted support* $wsup_{\text{gen}}$, based on a user-defined parameter λ ($0 \leq \lambda \leq 1$):

$$wsup_{\text{gen}}(g) := \lambda \cdot wsup_{\text{ext}}(g) + (1 - \lambda) \cdot wsup_{\text{int}}(g)$$

Based on a user-defined *minimum general-weighted-support* value and parameter λ, the authors define the *general-weighted-subgraph-mining* problem. Their solution to this problem is similar to the one of Jiang et al. [JCSZ10]: As the *minimum general-weighted-support* criterion is not anti-monotone, they rely on a weaker pruning criterion for mining with a pattern-growth-based subgraph-mining algorithm. The authors also propose a related problem, *mining external weighted subgraphs under internal weight constraints*, which is solved similarly within the same framework. In their experiments, Shinoda et al. [SOO09] achieve well results with synthetic data, communication graphs and chemical compound graphs.

Subsumption. While mining for *weighted frequent subgraphs* (or mining using the variations from Shinoda et al.) is adequate for certain applications, it relies on the assumption that high weight values identify significant components. This does not hold in every domain. For instance, in software-defect localisation, high (or low) edge-weight values are in general not related to defects. Therefore, *weighted-frequent-subgraph mining* cannot be used for every problem and offers less flexibility than constraints on arbitrary measures as investigated in Chapter 8 of this dissertation. Furthermore, to our knowledge, the *weighted-frequent-subgraph-mining techniques* presented in this section have never been evaluated systematically nor compared to alternative approaches.

3.2.2 Mining Significant Subgraphs

In many settings, frequent subgraph mining is followed by a feature-selection step. This is to ease subsequent processes such as *graph classification* [CYH10a] and to identify the most significant features. The different proposals use various objective functions for feature selection. Besides others, Cheng et al. [CYH10b] have identified this two-step approach of mining and selecting to be the computational bottleneck in many graph-mining applications: On the one side, generating large numbers of frequent subgraphs to choose from is expensive and in certain applications even infeasible. On the other side, the selection process can be expensive as well.

A number of studies investigate scalable subgraph-mining algorithms [CHS+08, RS09, SKT08, SNK+09, TCG+10, YCHY08]. They deal with the direct mining of subgraphs satisfying an objective function, instead of following the two-step approach. In other words, the subgraph sets mined might be incomplete with regard to the frequency criterion, but contain all (or most) graphs with regard to some other objective function. One can consider these functions to be constraints, as they narrow down the mining results. However, they do not necessarily fall into any of the constraint classes introduced in Section 2.3.3. Objective functions are either based on their ability to discriminate between classes or numerical values associated with the graphs [SKT08, SNK+09, TCG+10], on some other measure of significance [CHS+08, RS09] or leave this choice to the user by allowing for interchangeable measures [YCHY08]. In the following, we look at the approaches mentioned in a little more detail.

Boosting-Based Approaches

The approach from Saigo et al. [SNK+09], gBoost, builds on a boosting technique with decision-stump classifiers. In each iteration, they search for the most promising classifier, consisting of a single discriminative subgraph. These promising subgraphs are found by repeatedly calculating structural objective functions measuring the discriminativeness. They do so in a pattern-growth search space similar to the one from gSpan [YH02] (see Section 2.3.3). The authors use their discriminativeness measure to refine pruning bounds in the search space in each iteration.

Saigo et al. [SKT08] refine their approach in the gPLS algorithm. It makes use of the same boosting technique and pattern search space, but relies on partial least-squares regression (PLS) to prune the search space and to select the most promising subgraphs.

A Leap-Search-Based Approach

Yan et al. [YCHY08] present the LEAP algorithm. It allows for the integration of different kinds of objective functions that are not anti-monotone. The idea of the algorithm is not to prune the search space, but to leap in this space. This is in contrast to performing a (pruned) stringent depth-first search as done by algorithms such as gSpan [YH02] (see Section 2.3.3). Thereby it makes use of the observation that structurally similar subgraphs tend to have similar support values and statistical significance scores. Therefore, the authors rely on a strategy that mines with an exponentially decreasing minimum support threshold. This leads to a fast discovery of (near-)optimal subgraphs. In the evaluation, the authors use the *G-test* as well as *information gain* as objective functions. The G-test is a measure of statistical significance, and the information gain measures the discriminativeness of a subgraph (see Definition 2.7). They successfully apply their technique to several datasets from the

chemical domain. Furthermore, Cheng et al. [CLZ+09] employ the LEAP algorithm for call-graph-based defect localisation (see Chapter 5).

Mining with Optimality Guarantees

Both boosting-based approaches [SNK+09, SKT08] as well as the LEAP algorithm [YCHY08] have proven to work well in the respective settings and evaluations. However, they do not provide optimality guarantees. Thoma et al. [TCG+10] present an approach, CORK, which integrates an objective function into the pattern-growth-based frequent subgraph miner gSpan [YH02] (see Section 2.3.3). They use this function to greedily prune the search space. The distinctiveness of their approach is that the objective function has the *submodularity* property, and the authors show that such functions used for pruning ensure near-optimal results. This is, CORK provides the optimality guarantee that *almost all* discriminative subgraphs useful for classification are found.

A Partitioning-Based Approach

Ranu and Singh [RS09] investigate a setting that relies on significance (with respect to the statistical *p-value* measure) rather than on the ability to discriminate between classes. They observe that significant subgraphs might have any support value. In particular, significant subgraphs might have a support that is too low to be mined efficiently. This is as frequent-subgraph-mining algorithms roughly scale exponentially with decreasing minimum support values. Based on this observation, they develop the GraphSig technique which builds on two main steps: *In the first step*, they partition all graphs into sets such that all graphs in a set are likely to contain a common significant subgraph with a high support. They do so by using a technique similar to a sliding-window approach on the graphs, based on random walks. This generates a set of feature vectors for each graph. The authors then mine closed subfeature vectors which are significant and use them to group all graphs containing a subfeature vector into a group. *In the second step*, the authors make use of these groups of graphs. As these groups are relatively small, they apply a frequent-subgraph-mining technique on every set of graphs with a very small minimum support value. This procedure allows for finding significant subgraphs with a low support which cannot be discovered by traditional techniques due to scalability issues. In the evaluation, the authors demonstrate that their significant subgraphs are well-suited for graph-classification applications.

Mining Representative Subgraphs

Chaoji et al. [CHS+08] do not measure the significance of subgraphs nor their discriminativeness. They are concerned about finding subgraphs that are representative

for the complete set of frequent subgraphs (i.e., not similar to the graphs in the result set) with regard to the graph structure. To this end, the authors introduce parameter $\alpha \in [0, 1]$: Frequent subgraphs have to have a similarity to graphs in the result set below value α. Furthermore, they introduce parameter $\beta \in [0, 1]$: For every frequent subgraph that is not part of the result set, there has to be at lest one subgraph in the result set having a similarity of at least value β. In the ORIGAMI algorithm, the authors measure the similarity between two graphs by calculating the relative size of their maximum common subgraph. For mining frequent subgraphs which comply with the restrictions defined by the two parameters α, β, the authors mine a set of subgraphs *in a first step*. Instead of enumerating the complete set of such graphs, they adopt a random-walk approach which enumerates a subset of diverse subgraphs. *In a second step*, they extract the result set complying with the parameters. They do so by mapping the problem to a maximum-clique problem which they again solve with a randomised algorithm.

Subsumption

Various researchers have studied scalable mining of subgraph patterns, with much success. However, they have not taken weights into account. In this dissertation, in particular in Chapter 8, we use measures building on edge weights as objective functions, to decide which graphs are significant. The usage of weights allows for a more detailed analysis as compared to the graph structure only. Like the previous approaches, ours does not necessarily produce graph sets which are complete with regard to frequency or some other hard constraint.

3.2.3 Constraint-Based Subgraph Mining

Constraint-based mining allows the user to formulate constraints describing the patterns she or he is interested in. The mining algorithms in turn may make use of these constraints by narrowing down their internal search space and thus speeding up the algorithm. In Section 2.3.3, we have presented the constraint classes *anti-monotonicity*, *monotonicity* and *succinctness*, as originally introduced by Ng et al. [NLHP98].

More recently, constraint-based graph mining has been proposed. Wang et al. [WZW+05] build on the constraint classes introduced in [NLHP98] and categorise various graph-based constraints into these classes. Then the authors develop a framework to integrate the different constraint classes into a pattern-growth-based graph-mining algorithm. They use *anti-monotone constraints* to prune the search space and *monotone constraints* to speed up the evaluation of further constraints. Further, they use the *succinctness* property to reduce the size of the graph database. Wang et al. also propose a way to deal with some weight-based constraints. For the average-weight constraint, they propose to omit nodes and edges with outlier values from the graphs in the database. They do so to shrink the graph size and to avoid the evaluation of

such ‘unfavourable’ elements. This can lead to incomplete result sets. Furthermore, situations where such constraints lead to significant speedups are rare, according to the evaluation of the authors with one artificial dataset, and they do not make any statements regarding result quality.

In [ZYHY07], Zhu et al. extend [WZW+05] by refining the classes of constraints, and they integrate them into mining algorithms. However, they do not consider weights, too.

Although the techniques proposed work well with *monotone*, *anti-monotone* or *succinct* constraints and their derivations, most weight-based constraints do not fall into these categories [WZW+05]. They are not *convertible* (see Section 2.3.3) as well, even if such constraints might seem to be similar. The weights considered in convertible constraints stay the same for every item in all transactions, while weights in graphs can be different in every graph in D. Therefore, the established constraint-based-mining schemes cannot use weight-based constraints for pruning while guaranteeing completeness.

4 Call-Graph Representations

Call-graph-based defect localisation naturally relies on call graphs. Such graphs are representations of programme executions. Raw call graphs typically become much too large for graph-mining algorithms, as programmes might be executed for a long period and frequently call other parts of the programme, which adds information to the graph. Therefore, it is essential to compress the graphs – we call this process also *reduction*. It is usually done by a lossy compression technique. This involves the trade-off between keeping as much information as possible and a strong compression. The literature has proposed a number of different call-graph representations [CLZ+09, DFLS06, LYY+05], standing for different degrees of reduction and different types and amounts of information encoded in the graphs. In this dissertation, we make further proposals for call graph compressions and for encoding additional information by means of numerical annotations at the edges. To ease presentation, we discuss the related approaches from the literature (in particular the call-graph representations $R^{\text{tmp}}_{\text{total}}$, $R^{\text{ord}}_{\text{01m}}$ and $R^{\text{block}}_{\text{total}}$; R_{total} and $R^{\text{unord}}_{\text{01m}}$ are simplified variants thereof) along with the new proposals (R_{subtree}) or variations ($R^{\text{w}}_{\text{total}}$, $R^{\text{mult}}_{\text{total}}$) in this dissertation. Besides the graph representations discussed in this chapter, we introduce further graph representations in Chapters 6 and 7. They focus on specific graph representations for call graphs at different levels of abstraction and on the incorporation of dataflow-related information, respectively.

In Section 4.1, we discuss call-graph representations at the method level. In Section 4.2, we briefly explain call graphs at other levels of granularity than the method level. In Section 4.3, we present call-graph representations for multithreaded programmes. In Section 4.4, we explain how we technically derive call graphs from Java programme executions. Section 4.5 subsumes this chapter.

4.1 Call Graphs at the Method Level

We now discuss call-graph representations at the method level. The basis for all such representations are unreduced call graphs, sometimes also called *call trees*, as obtained from tracing programme executions (in Section 4.4 we give some details on tracing):

Notation 4.1 (Unreduced call graphs)
Unreduced call graphs can be obtained by tracing a programme execution. They are rooted ordered trees*. Nodes stand for methods and one edge stands for each method invocation. The order of the nodes is the temporal order in which the methods were executed.*

Example 4.1: Figure 4.1(a) is an example of such a graph. Even if not depicted in the figure, the siblings in the graph are ordered by execution time from left to right. When we want to emphasise the temporal order, we express the order by increasing integers attached to the nodes. Figure 4.4(a) is the same graph featuring this representation.

In Section 4.1.1, we describe the *total reduction* scheme. In Section 4.1.2, 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 Section 4.1.3. We provide some ideas on the reduction of recursion in Section 4.1.4 and conclude with a brief comparison in Section 4.1.5.

4.1.1 Total Reduction

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

Notation 4.2 (Total reduction, R_{total})
In totally reduced graphs at the method level, every distinct method is represented by exactly one node. When one method has called another method at least once in an execution, a directed edge connects the corresponding nodes.

Note that the total reduction may give way to the existence of loops in R_{total} graphs (i.e., the output is a regular graph), and it limits the size of the graph (in terms of nodes) to the number of methods of the programme. In defect localisation, Liu et al. [LYY+05] have introduced this technique, along with a temporal extension (see Section 4.1.3).

In this dissertation, we extend the plain total-reduction scheme (R_{total}) to include call frequencies. We do so as this eases the discovery of frequency-affecting bugs, as we will see.

Notation 4.3 (Total reduction with edge weights, $R^{\text{w}}_{\text{total}}$)
Building on R_{total} graphs as defined in Notation 4.2, every edge is annotated with a numerical weight. It represents the total number of calls of the callee method from the caller method.

Even though the extension in the $R^{\text{w}}_{\text{total}}$ graphs is quite simple, we are not aware of any studies using weighted call graphs for defect localisation. Furthermore, these

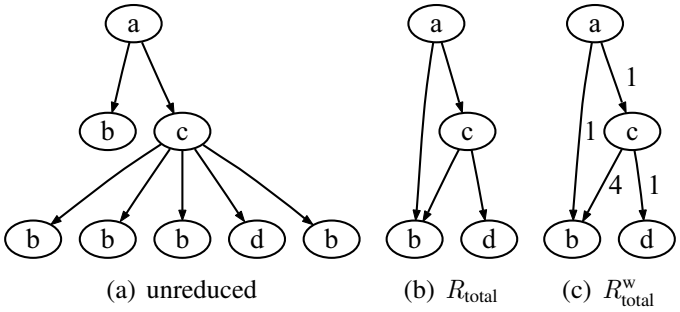


Figure 4.1: Total reduction techniques.

weights allow for more detailed analyses, in particular regarding the localisation of *frequency-affecting bugs*.

Example 4.2: Figure 4.1 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 ($R^{\text{w}}_{\text{total}}$).

In general, total reduction (R_{total} and $R^{\text{w}}_{\text{total}}$) reduces the graphs quite significantly. Therefore, it allows graph-mining-based defect localisation with software projects larger than other reduction techniques. On the other hand, much information on the programme 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 $R^{\text{w}}_{\text{total}}$). In particular, the information is lost in which context (at which position within a graph) a certain substructure is executed.

4.1.2 Reduction of 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. Furthermore, as described before, the relatively severe total-reduction techniques give way to the assumption that they lose much information originally available in unreduced call graphs. In the following, we introduce two variants that encode more structural information than totally reduced graphs.

Notation 4.4 (Unordered zero-one-many reduction, $R^{\text{unord}}_{\text{01m}}$)
Unordered zero-one-many reduced graphs are rooted (unordered) trees where nodes represent methods and edges method invocations. In contrast to unreduced call graphs (see Notation 4.1), such graphs ignore the order and omit isomorph substructures which occur more than twice below the same parent node.

The $R^{\text{unord}}_{\text{01m}}$ reduction 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

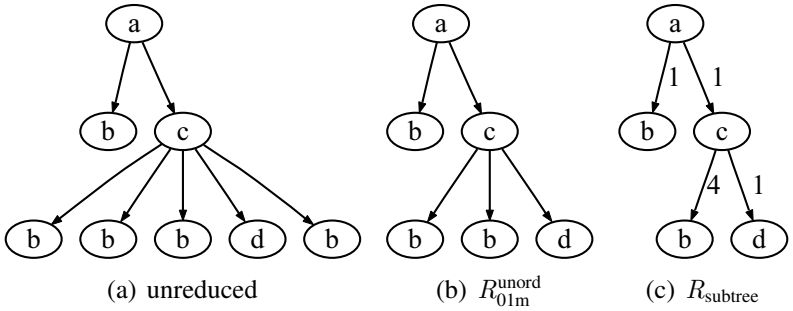


Figure 4.2: Reduction techniques based on iterations.

substructure is executed several times is still encoded in the graph structure, but without exact numbers. This is indicated by doubled substructures within the call graph (only substructures occurring more than twice are not included). Compared to total reduction (R_{total}), more information on a programme execution is kept. The downside is that R_{01m}^{unord} call graphs generally are much larger.

The R_{01m}^{unord} reduction is a simplified variant of the one from Di Fatta et al. [DFLS06] (see R_{01m}^{ord} in Section 4.1.3). The difference is that R_{01m}^{unord} graphs do not take the temporal order of the method executions into account. We use this representation in this dissertation for comparisons with other techniques which do not make use of temporal information.

Notation 4.5 (Subtree reduction, R_{subtree})
Subtree-reduced graphs are rooted (unordered) trees where nodes represent methods and edges method invocations. This reduction ignores the order and reduces subtrees executed iteratively by deleting all but one isomorph subtree below the same parent node in an unreduced call tree (see Notation 4.1). The edges are weighted and numerical weights represent call frequencies. Algorithm 4.1 describes the reduction procedure in detail.

The R_{subtree} reduction is newly proposed in this dissertation. It leads to smaller graphs than R_{01m}^{unord}. The edge weights allow for a detailed analysis; they serve as the basis of our analysis technique described in Chapter 5. We discuss details of the reduction technique in the remainder of this section.

Example 4.3: Figure 4.2 illustrates the two iteration-based reduction techniques: (a) is an unreduced call graph, (b) its zero-one-many reduction without temporal order (R_{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_{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_{total}^{w} in Figure 4.1(c), but the same number of edges.

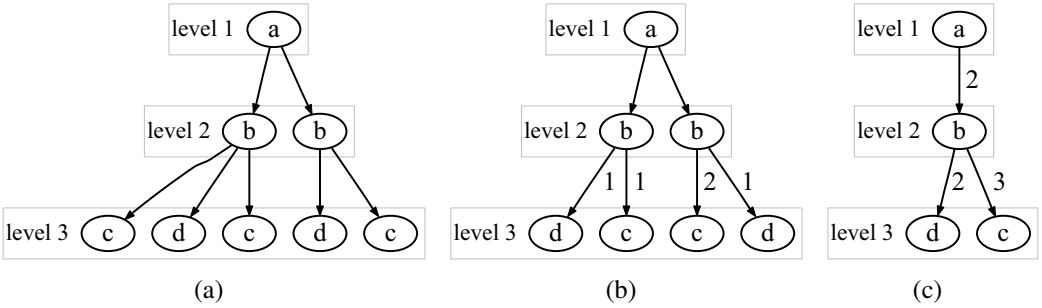


Figure 4.3: A raw call tree, its first and second transformation step.

Note that with R_{total}, and with $R_{01\text{m}}^{\text{unord}}$ in most cases as well, the graphs of a correct and a failing execution with a *frequency-affecting bug* are reduced to exactly the same graph. With R_{subtree} (and with $R_{\text{total}}^{\text{w}}$), the edge weights would be different when call frequency-affecting bugs occur. Analysis techniques can discover this (see Chapter 5).

The Subtree-Reduction Procedure

For the subtree reduction (R_{subtree}), we organise 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 naïve approach to reduce the example call tree in Figure 4.3(a) 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, we proceed level by level, starting from $level\ n-1$, as described in Algorithm 4.1.

Algorithm 4.1 Subtree reduction algorithm.

```
1: Input: a call tree organised 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
```

Example 4.4: Suppose we want to reduce the graph given in Figure 4.3(a). We start 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 4.3(b). In the next level, $level$ 1, we process 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 4.3(c).

4.1.3 Temporal Order in Call Graphs

So far, the call graphs described just represent the occurrence of method calls. Even though, say, Figure 4.2(c) 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 programme executions, the defect-localisation techniques proposed in [DFLS06, LYY+05] take the temporal order of method calls within one call graph into account. In the following, we introduce the corresponding reductions.

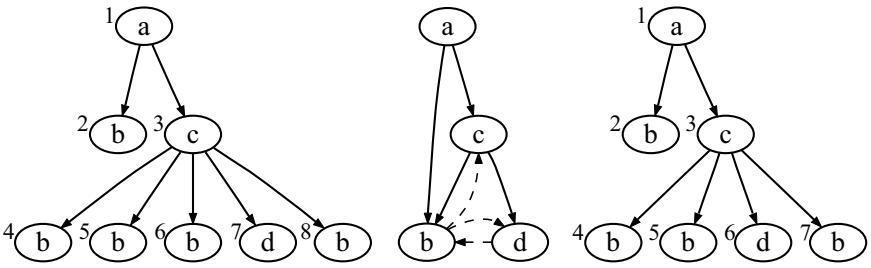
Notation 4.6 (Total reduction with temporal edges, $R^{\text{tmp}}_{\text{total}}$)
In addition to the total reduction (R_{total}, see Notation 4.2), totally reduced graphs with temporal edges have so-called temporal edges *that are directed. Such an edge connects two methods which are executed consecutively and are invoked from the same method. Technically, temporal edges are directed edges with another label, e.g., 'temp', compared to other edges which are labelled, say, 'call'.*

The $R^{\text{tmp}}_{\text{total}}$ reduction has been introduced by Liu et al. [LYY+05], and the resulting graphs are also known as *software-behaviour graphs*. As the graph-mining algorithms used for further analysis can handle edges labelled differently, the analysis of $R^{\text{tmp}}_{\text{total}}$ graphs does not give way to any special challenges, except for an increased number of edges. In consequence, the totally reduced graphs lose their main advantage, their small size. However, taking the temporal order into account might help discovering certain defects.

Notation 4.7 (Ordered zero-one-many reduction, $R^{\text{ord}}_{\text{01m}}$)
Ordered zero-one-many-reduced graphs are as unreduced call graphs (see Notation 4.1) rooted ordered trees. *To include the temporal order, the reduction technique differs to the R^{unord}_{01m} reduction (see Notation 4.4) as follows: While R^{unord}_{01m} omits any isomorph substructure which is invoked more than twice from the same node, only substructures are removed which are executed more than twice in direct sequence.*

The $R^{\text{ord}}_{\text{01m}}$ reduction has been introduced by Di Fatta et al. [DFLS06]. As the resulting graphs are rooted ordered trees, they can be analysed with an order-aware tree mining algorithm. The fact that substructures are only removed when they occur in direct sequence facilitates that all temporal relationships are retained. For instance, in the reduction of the sequence b, b, b, d, b (see Figure 4.4) only the third b is removed, and it is still encoded that b is called after d once.

Depending on the actual execution, the $R^{\text{ord}}_{\text{01m}}$ 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 [DFLS06] partly compensates the additional effort for mining algorithms caused by such sizes, which are huge compared to $R^{\text{unord}}_{\text{01m}}$. Rooted ordered tree mining algorithms scale significantly better than usual graph-mining algorithms [CMNK05], as they make use of the order.



(a) unreduced (b) $R^{\text{tmp}}_{\text{total}}$ (c) $R^{\text{ord}}_{\text{01m}}$

Figure 4.4: Temporal information in call-graph reductions.

Example 4.5: Figure 4.4 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, $R^{\text{tmp}}_{\text{total}}$) and (c) is its ordered zero-one-many reduction ($R^{\text{ord}}_{\text{01m}}$). Note that, compared to $R^{\text{unord}}_{\text{01m}}$, $R^{\text{ord}}_{\text{01m}}$ keeps a third node b called from c, as the direct sequence of nodes labelled b is interrupted.

4.1.4 Reduction of Recursions

Another challenge with the potential to reduce the size of call graphs is recursion. The total reductions (R_{total}, $R^{\text{w}}_{\text{total}}$ and $R^{\text{tmp}}_{\text{total}}$) implicitly handle recursion as they reduce both iteration and recursion. For instance, 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 4.5(a) where method b calls itself. Figure 4.5(b) presents a possible reduction represented with a self-loop at node b. In Figure 4.5(b), 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 4.5(c) where b calls c which again calls b etc. Such

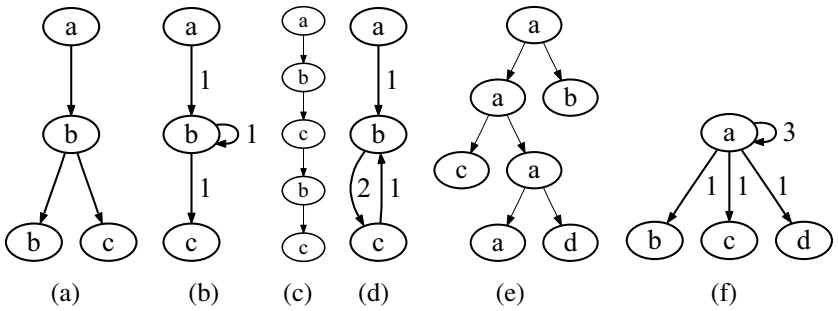


Figure 4.5: Examples for reduction based on recursion.

chains can be of arbitrary length. Obviously, such *indirect recursions* can be reduced as shown in Figures 4.5(c) and 4.5(d). This leads to the existence of loops.

Both types of recursion are challenging when it comes to reduction. Figures 4.5(e) and 4.5(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 defect localisation, one might want to avoid such artefacts. For instance, d called from exactly the same method as b could be a *structure-affecting bug* which is not found when such artefacts 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 artefacts may occur.

In this dissertation, we focus on the reduction of iterations (R_{subtree}) or fall back to total reduction with weights ($R^{\text{w}}_{\text{total}}$). This fall back has the advantage that we deal with smaller graphs making graph mining easier and that recursions are treated without any extra effort.

4.1.5 Comparison

To compare reduction techniques, we must look at the level of compression they achieve on call graphs. Table 4.1 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 [Dar04] used in the evaluation in Chapter 5. Clearly, the effect of the reduction techniques varies extremely depending on the kind of programme and the data processed. However, the small programme 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}, $R^{\text{w}}_{\text{total}}$	22	30
R_{subtree}	36	35
$R^{\text{tmp}}_{\text{total}}$	22	47
$R^{\text{unord}}_{\text{01m}}$	62	61
$R^{\text{ord}}_{\text{01m}}$	117	116
unreduced	2,199	2,198

Table 4.1: Examples for the effect of call-graph-reduction techniques.

Obviously, the total reduction (R_{total} and $R^{\text{w}}_{\text{total}}$) achieves the strongest compression and yields a reduction by two orders of magnitude. As 22 nodes remain, the programme 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 ($R^{\text{unord}}_{\text{01m}}$) 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 ($R^{\text{tmp}}_{\text{total}}$), the number of edges increases by roughly 50% due to the temporal information, while the ordered zero-one-many reduction ($R^{\text{ord}}_{\text{01m}}$) almost doubles this number. Chapter 5 assesses the effectiveness of defect localisation with the different reduction techniques along with the localisation 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 summarise, different authors have proposed different reduction techniques, each one together with a localisation technique (see Chapter 5): the total reduction ($R^{\text{tmp}}_{\text{total}}$) in [LYY+05], the zero-one-many reduction ($R^{\text{ord}}_{\text{01m}}$) in [DFLS06] and the subtree reduction (R_{subtree}) proposed in this dissertation. Some of the reductions can be used or at least be varied in order to work together with a defect-localisation technique different from the original one. In Chapter 5, we present original and varied combinations.

4.2 Call Graphs at Different Levels of Granularity

So far, we have considered call graphs at the method level. However, call graphs can be more fine grained or more coarse grained. Finer levels of granularity allow for more detailed defect localisations, but the graphs are typically much larger. Coarser granularities are less detailed, but lead to smaller graphs. In the following, we look a finer levels of granularity, in particular at basic-block-level call graphs as defined by

Cheng et al. [CLZ+09]. In Chapter 6, we look at coarser call-graph representations, i.e., at the class level and at the package level.

Cheng et al. [CLZ+09] rely on the method-level call graphs with total reduction and temporal edges as introduced by Liu et al. [LYY+05] ($R^{\text{tmp}}_{\text{total}}$, see Notation 4.6). Besides these graphs, they also introduce basic-block-level call graphs ($R^{\text{block}}_{\text{total}}$), aiming at more fine-grained defect localisations:

Notation 4.8 (Basic-block-level call graphs, $R^{\text{block}}_{\text{total}}$)
Each basic block *as known from static* control-flow graphs *(see Section 2.2.1) forms a node in the dynamic R^{block}_{total} call graph. Three kinds of differently labelled directed edges connect these nodes. Edges of the type 'call' correspond to method calls, edges of the type 'trans' to transitions between two basic blocks and edges of the type 'return' to method returns.*

Example 4.6: Listing 4.1 is an example Java source code of a programme containing a simple integer-multiplication method (known from Examples 2.1 and 2.2) and a main method that calls the multiplication method once. Figure 4.6 is the corresponding basic-block-level call graph ($R^{\text{block}}_{\text{total}}$), representing a single execution of the programme.

Note that Cheng et al. [CLZ+09] do not make use of any weights in their $R^{\text{block}}_{\text{total}}$ graphs. However, introducing weights corresponding to call/transition/return frequencies would be easy.

4.3 Call Graphs of Multithreaded Programmes

So far, we have considered call graphs from single-threaded programmes. However, as motivated in Section 3.1.3, defect localisation in multithreaded programmes is a challenging field. Although multithreaded programmes are not in the focus of this dissertation, we discuss some specialities of such programmes and possible call-graph representations in this section. We limit these discussions to the method-level case, although respective graphs can be defined similarly for other levels of granularity. In Appendix A, we evaluate the usefulness of the graphs developed here for defect localisation in multithreaded programmes.

Unreduced and Totally-Reduced Multithreaded Call Graphs. In the multithreaded case, every method can be executed several times in more than one thread. Therefore, in unreduced call graphs, nodes are initially labelled with a prefix consisting of the respective thread ID and method name. Figure 4.7(a) illustrates an example of such a call graph. This example represents the method calls of one programme execution, without any reductions. To achieve a strong reduction of call graphs from potentially large multithreaded programmes, we consider a total reduction of the graphs

```
1 public static void main(String[] args) {
2   System.out.println(mult(3, 4));
3 }
4
5 public static int mult(int a, int b) {
6   int res = 0;
7   int i = 1;
8   while (i <= a) {
9     res += b;
10    i++;
11  }
12  return res;
13 }
```

Listing 4.1: Example Java programme performing an integer multiplication.

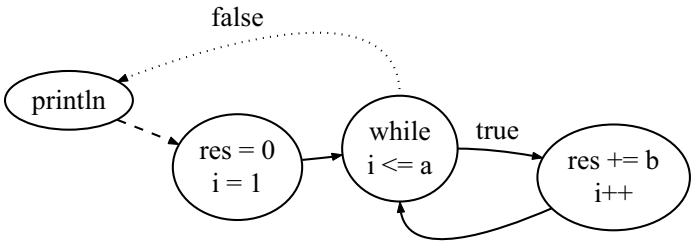


Figure 4.6: A basic-block-level call graph ($R^{\text{block}}_{\text{total}}$) [CLZ+09], representing the execution of the programme from Listing 4.1. Dashed lines stand for 'call' edges, solid lines for 'trans' edges and dotted lines for 'return' edges.

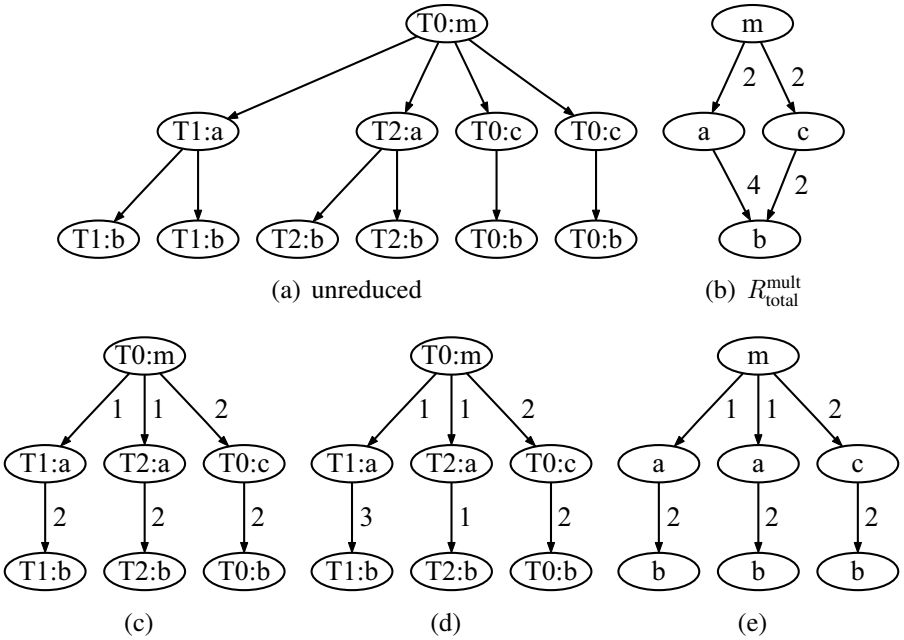


Figure 4.7: Example graphs illustrating alternative choices for call-graph representations for multithreaded applications.

in the following (see Section 4.1.1). As in all total reduction variants, each method is uniquely represented by exactly one node, for the moment identified by the method name prefixed with the thread ID. Two nodes are connected by an edge if the corresponding methods call each other at least once. Furthermore, we use edge weights as in the $R^{\text{w}}_{\text{total}}$ graph representations to represent call frequencies. Figure 4.7(c) is an example for such a totally reduced graph representation, it is the reduced version from the call graph in Figure 4.7(a).

Temporal Relationships. For the localisation of defects in multithreaded software, it seems to be natural to encode temporal information in call graphs, e.g., to tackle race conditions caused by varying thread schedules. The call graphs such as the one in Figure 4.7(a) do not encode any order of execution of the different threads and methods. One straight-forward approach to include such information uses temporal edges (see Section 4.1.3). The problem with this idea, however, is that the overhead to obtain such information might be large and requires sophisticated tracing techniques. Furthermore, it may significantly influence programme behaviour – possibly making a failure disappear. In addition, increasing the amount of information in the call graph makes the mining process more difficult and time-consuming. We therefore propose a more lightweight approach without temporal information encoded in the graphs.

Non-Deterministic Thread Names. Figure 4.7(c) illustrates our totally reduced call-graph representation that contains the thread IDs in the node labels. This is awkward, as threads are allocated dynamically by the runtime environment or the operating system. Therefore, various correct executions could lead to threads with different IDs for the same method call, even for a programme using the same parameters and input data. We therefore would not be able to compare several programme executions based on the node labels. Omitting this information would result in the graph shown in Figure 4.7(e), which is directly derived from the one in Figure 4.7(c).

Replicated Tasks and Varying Thread Interleavings. Graphs such as the ones in Figures 4.7(c), (d) and (e) suffer from two problems: (1) They might contain a high degree of redundancy that does not help finding defects. For example, a programme using thread pools could have a large number of threads with similar calls due to the execution of replicated tasks (and therefore similar method calls). This typically produces a call graph with several identical and large subtrees, which contain no meaningful information for defect localisation. (2) The call frequencies (i.e., the edge weights) might not be useful for defect localisation, either. Different execution schedules of the same programme can lead to graphs with widely differing edge weights. Example 4.7 illustrates how this effect can disturb data-mining analyses, as such differences are not related to infections.

Example 4.7: Think of method a in Figure 4.7(c) as the `run()` method, calling the worker task method b, which takes work from a task pool. Sometimes, thread 1 and thread 2 would both call method b twice, as in Figure 4.7(c). In other cases as in Figure 4.7(d), depending on the scheduling, thread 1 could call method b three times, while thread 2 would only call it once or vice versa.

Proposed Graph Representation. Based on the observations discussed so far, we propose a graph representation that avoids repeated substructures as follows:

Notation 4.9 (Total reduction for multithreaded programmes, $R^{\text{mult}}_{\text{total}}$)
Similar to R_{total} graphs as described in Notation 4.2 for single-threaded programmes, R^{mult}_{total} graphs do not consider the thread names or IDs either. This is, all nodes referring to the same method are merged into a single node, even if it is called within different threads.

Example 4.8: Figure 4.7(b) is an example for $R^{\text{mult}}_{\text{total}}$ graphs. It is the reduced version of the multithreaded call graph in Figure 4.7(a).

The representation proposed is robust in the sense that different schedules do not influence the graph structure. The reason is that methods executed in different threads are mapped to the same nodes. The downside of this representation is that graph structures from different executions rarely differ. Consequently, a structural analysis

of the call graphs as in other approaches (e.g., [LYY+05, DFLS06]) is less promising. However, the edge weights introduced aim to compensate for this effect and allow for detailed analyses.

Possible Extensions. As mentioned, our proposal for reduced call graphs of multithreaded programmes as described in the previous paragraph does not lead to many differences in the graph structure. We therefore present some ideas for possible extensions in the following. Clearly, they should deal with issues such as those related to non-deterministic thread names, varying thread interleavings and replicated tasks as discussed before. As one example, graph representations can have distinct substructures for the different types of threads. A possible solution for the problem of indeterministic thread IDs is the introduction of *thread classes*. Each of these classes stands for a source-code context, i.e., a position in the source code where new threads are created. As an example, one class could stand for GUI-related threads and one for database-access-related threads. Further information to enhance the expressiveness of call graphs could be information on locks on certain objects. This information could be included as an annotation of nodes or edges.

4.4 Derivation of Call Graphs

In order to derive call graphs from programme executions, we have to trace the executions and to store the relevant information. As we rely on Java in this dissertation, we employ AspectJ [KHH+01] to weave tracing functionality into the Java programmes considered. AspectJ is an *aspect-oriented programming* (AOP) language [KLM+97] which provides cross-cutting concern functionality for Java. The basic functionality of AspectJ is to define so-called *pointcuts* which allow for the addition of extra functionality at certain points of a programme execution. Listing 4.2 contains the essence of the *aspects* we use to generate call graphs.

```
1 public aspect tracing {
2   pointcut getMethod() : execution(* *(..)) && !execution(*
       AspectJ..*(..));
3   before(): getMethod() {
4     //derive callingMethod and calleeMethod
5     edgeName = callingMethod + " -> " + calleeMethod;
6     callGraph.add(edgeName);
7   }
8 }
```

Listing 4.2: AspectJ code.

As usual in AspectJ, we first declare an *aspect*, `tracing` in our case, in Line 1 of Listing 4.2. We then define a *pointcut* which catches all method invocations (`execution(* *(..))`) in Line 2. The second part of this line (the part behind `&&`) avoids that AspectJ-specific methods will be traced and become part of our call graphs. What follows is the definition of an *advice*, starting in Line 3. An advice describes what has to be done within a pointcut and at which exact point of the execution. In this case, Lines 4 to 6 are executed before a method matched by the pointcut is actually invoked (this is controlled by the keyword `before()` in Line 3). In the body of the advice, we first derive the names of the methods involved (Line 4). We derive the callee method by accessing the special variable `thisJoinPointStaticPart` in AspectJ (which involves reflection) and the calling method with a stack we maintain by ourself. We then assemble an edge name based on the two method names (Line 5) and store it in an internal data structure (Line 6). This data structure counts the occurrences of all edges in an edge list and it can easily be used to derive call graphs in arbitrary representations. We use a dedicated pointcut at the end of a programme execution to write the call graph into a file.

4.5 Subsumption

In this chapter, we have introduced various call-graph representations that are the basis for the defect-localisation techniques we introduce in the following Chapter 5. We have focused on method-level call graphs with its different variants, and we have compared these graphs from a descriptive point of view. In Chapter 5, we will shed light on their usefulness for defect localisation. Besides method-level graphs, we have discussed some graph representations at different levels of granularity (and will do so more extensively in Chapter 6) as well as representations for multithreaded programmes. Further, we have explained how we actually derive call graphs from programme executions.

5 Call-Graph-Based Defect Localisation

This chapter focuses on the actual defect-localisation process. The related work [CLZ+09, DFLS06, LYY+05] and as well this dissertation suggest a number of different approaches for this process, relying on various call-graph representations (see Chapter 4). In this dissertation, we distinguish between various structural approaches [CLZ+09, DFLS06, LYY+05] and novel frequency-based and combined approaches.

In this chapter, we first present an overview in Section 5.1. To ease presentation, we then discuss existing related approaches in Section 5.2, directly followed by the novel approaches in Section 5.3. We then present an experimental evaluation in Section 5.4 and a subsumption in Section 5.5.

The approach presented in this chapter (in particular in Section 5.3) serves as a basis for the more sophisticated approaches in Chapters 6 and 7 focusing on scalability issues and defects that affect the dataflow, respectively. In Chapter 8, we present an approach for constraint-based subgraph mining which is a further development of the approach presented in Section 5.3.

5.1 Overview

We now give an overview of the procedure of call-graph-based defect localisation. This is a generic procedure which applies to the techniques that are new in this dissertation (Section 5.3) as well as to most related studies (Section 5.2). Algorithm 5.1 first assigns a class ($correct$, $failing$) to every programme trace (Line 3), using a test oracle (see Section 2.2.3). The approaches discussed in this dissertation require such an oracle, and they are typically available in the software development process [JH05]. Then every trace is reduced (Line 4), which leads to smaller call graphs (see Chapter 4). Now frequent subgraphs are mined (Line 6). For this step, several algorithms, e.g., tree mining or graph mining in different variants, can be used. The last step calculates a likelihood of containing a defect. This can be at different levels of granularity, typically at the method level (as shown in Line 7). The calculation of the likelihood is based on the frequent subgraphs mined and facilitates a ranking of the methods, which can then be given to the software developer.

Algorithm 5.1 Generic graph-mining-based defect-localisation procedure.

Input: a collection of programme traces $t \in T$

1: $G = \varnothing$ // initialise a collection of reduced graphs
2: **for all** traces $t \in T$ **do**
3: assign a $class \in \{correct, failing\}$ to t
4: $G = G \cup \{reduce(t)\}$
5: **end for**
6: $SG = frequent\_subgraph\_mining(G)$
7: calculate $P(m)$ for all methods m, based on SG

5.2 Existing Structural Approaches

Structural approaches for defect localisation can localise *structure-affecting bugs* in particular. In some cases, a likelihood $P(m)$ that method m contains a defect is calculated, for every method. This likelihood is then used to rank the methods. In the following, we refer to it as a *score*. In Sections 5.2.1–5.2.3 we introduce and discuss the different structural scoring approaches.

5.2.1 The Approach from Di Fatta et al.

Di Fatta et al. [DFLS06] use the $R^{\text{ord}}_{01\text{m}}$ call-graph reduction (see Chapter 4) and the rooted ordered tree miner FREQT [AAK+02] to find frequent subtrees (Line 6 in Algorithm 5.1). The call trees analysed 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 neighbourhood* (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:

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

where $support(g, D)$ 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. [DFLS06] uses a minimum support $supp_{\text{min}}$ of 85%.

Based on the *specific neighbourhood*, Di Fatta et al. define a *structural score* P_{SN} which can be used to guide the following manual debugging process:

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

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

5.2.2 The Approach from Liu et al.

Although [LYY+05] is the first study which applies graph-mining techniques to dynamic call graphs to localise non-crashing bugs, this work from Liu et al. is not directly compatible to the approach from Di Fatta et al. [DFLS06]. In [LYY+05], defect localisation is achieved by a rather complex classification process, and it does not generate a ranking of methods suspected to contain a defect, but a set of such methods.

The work is based on the $R^{\text{tmp}}_{\text{total}}$ reduction technique and works with total reduced graphs with temporal edges (see Chapter 4). The call graphs are mined with a variant of the CloseGraph algorithm [YH03] (see Section 2.3.3). This step results in frequent subgraphs which are turned into binary features characterising a programme execution: A binary 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) classifier [Vap95] is learned which decides if a programme execution is $correct$ or $failing$. More precisely, for every method, two classifiers are learned: one based on call graphs including the respective method and 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 defect. 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 programme 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 localise defects more easily. However, the approach does not provide any ranking, which makes it hard to compare the results to other works.

5.2.3 The Approach from Cheng et al.

The study from Cheng et al. [CLZ+09] builds on the same graphs as used by Liu et al. [LYY+05]: totally reduced graphs with temporal edges ($R^{\text{tmp}}_{\text{total}}$). However, it relies on discriminative subgraph mining with the LEAP algorithm [YCHY08] (see Section 3.2.2). Cheng et al. first apply a heuristic graph filtering procedure to classified $R^{\text{tmp}}_{\text{total}}$ call graphs. This aims at shrinking the graph sizes by removing edges with a lower likelihood to be related to a defect. However, the authors do not provide any guarantees that this does not lose parts of the graphs that are actually relevant for defect localisation. Then, the authors apply the LEAP algorithm to the filtered graphs, resulting in the top-k discriminative subgraphs (discriminative with respect to $correct$, $failing$), i.e., subgraphs having an increased likelihood to be related to defects. The authors then report these subgraphs to the user to ease the manual debugging process. As with the approach from Liu et al. [LYY+05], the results cannot directly be compared to other approaches, as no method ranking is generated. Be-

sides the $R^{\text{tmp}}_{\text{total}}$ graph representation, Cheng et al. also apply their approach to more fine-grained basic-block-level call graphs ($R^{\text{block}}_{\text{total}}$, see Section 4.2).

5.3 Frequency-Based and Combined Approaches

As mentioned before, the structural approaches for defect localisation have their strengths in localising *structure-affecting bugs* (see Section 5.2). In particular the totally reduced graphs used in [CLZ+09, LYY+05] lose all information about the frequency of method calls (except the information whether a certain method is called or not). This makes it hard to impossible to localise frequency-affecting bugs. However, these techniques might find such defects when the infection leads so side effects that change the structure of the call graphs.

We now develop a novel technique that specialises on the localisation of *frequency-affecting bugs* in Section 5.3.1. In order to be able to localise a possibly broad range of defects, we then present novel approaches for the combination of structural and frequency-based techniques in Section 5.3.2.

5.3.1 Frequency-Based Approach

We now develop a technique that is able to localise *frequency-affecting bugs*. To do so, it is natural to analyse the call frequencies that are included as edge weights in some of the call-graph representations proposed in Chapter 4. As discussed before (see Section 3.2.1), there are no weighted subgraph-mining approaches that can be used directly for defect localisation. We therefore present a postprocessing approach in the following. It builds on frequent subgraph mining and feature selection to analyse the edge weights. Similarly to the structural approaches (see Section 5.2), the aim is to calculate a score, i.e., a likelihood to contain a defect, for every method. In the following we describe the individual steps.

Graph Mining

After having reduced the call graphs gained from correct and failing programme executions using the R_{subtree} technique (see Chapter 4), we search for frequent closed subgraphs SG in the graph dataset G using the CloseGraph algorithm [YH03] (Line 6 in Algorithm 5.1; see Section 2.3.3). For this step, we employ the ParSeMiS graph mining suite [PWDW09]. Closed mining reduces the number of graphs in the result set significantly and increases the performance of the mining algorithm. Furthermore, the usage of a general subgraph-mining algorithm instead of a tree miner allows for comparative experiments with other graph-reduction techniques such as $R^{\text{w}}_{\text{total}}$ (see Section 5.4). We use the subgraphs obtained from this frequent-subgraph-mining step as different *contexts* and perform all further analyses for every subgraph context

separately. This aims at a higher precision than an analysis without such contexts and allows to localise defects that only occur in a certain context.

Example 5.1: 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 the context of call graphs containing all methods mentioned, but not in graphs without method c.

Analysis of Weights

We now consider the edge weights. As an example, a frequency-affecting bug increases the frequency of a certain method invocation and therefore the weight of the corresponding edge. To find the defect, one has to search for edge weights which are increased in failing executions. To do so, we 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 programme are most significant to discriminate between $correct$ and $failing$.

To identify discriminative edges, one possibility is to consider different *edge types*, e.g., edges having the same calling method m_{s} (start) and the same callee method m_{e} (end). However, edges of one type can appear more than once within one subgraph and, of course, in several different subgraphs. Therefore, we analyse every edge in every such location, which we refer to as a *context*. This aims at a high probability to reveal a defect. As doing so, we typically investigate every edge weight in many different contexts. To specify the exact location of an edge in its context within a certain subgraph, we do not use the method names, as they may occur more than once. Instead, we use a unique id for the calling node (id_{s}) and another one for the callee method (id_{e}). All ids are valid within their subgraph. To sum up, we reference edges in its context in a certain subgraph sg with the following tuple: $(sg, id_{\text{s}}, id_{\text{e}})$.

A certain defect does not affect all method calls (edges) of the same type, but method calls of the same type in the same context. To allow for a more detailed analysis, we take this information into account, and we assemble a comprehensive feature table as follows:

Notation 5.1 (Feature tables for defect localisation with R_{subtree} graphs)
The feature tables have the following structure: The rows stand for all programme executions, represented by their call graphs. For every edge in every frequent subgraph, there is one column. The table cells contain the edge weights, except for the very last column, which contains the $class \in \{correct, failing\}$. *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. If a subgraph is not contained in a call graph, the corresponding cells have value 0.*

exec.	$a \to b$ (sg_1,id_1,id_2)	$a \to b$ (sg_1,id_1,id_3)	$a \to b$ (sg_2,id_1,id_2)	$a \to c$ (sg_2,id_1,id_3)	$\cdots$	class
g_1	0	0	13	6513	$\cdots$	*correct*
g_2	512	41	8	12479	$\cdots$	*failing*
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$

Table 5.1: Example table used as input for feature-selection algorithms.

Example 5.2: Table 5.1 serves as an example. The first column contains a reference to the programme 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*. g_1 does not contain sg_1, and the respective cells have value 0.

The table structure described allows for a detailed analysis of edge weights in different contexts within a subgraph. Algorithm 5.2 describes all subsequent steps in this section. After putting together the table, we deploy a standard feature-selection algorithm, information gain ($InfoGain$, see Definition 2.7), to calculate the discriminativeness of the columns in the table and thus the different edges. We use the implementation from the Weka data-mining suite [HFH+09] to calculate the $InfoGain$ with respect to the class of the executions (*correct* or *failing*) for every column (Line 1 in Algorithm 5.2). We interpret the values as a likelihood of being responsible for defects. Columns with an $InfoGain$ of 0, i.e., the edges always have the same weights in both classes, are discarded immediately (Line 2 in Algorithm 5.2).

Algorithm 5.2 Procedure to calculate $P_{\text{freq}}(m_s, m_e)$ and $P_{\text{freq}}(m)$.

Input: a set of edges $e \in E$, $e = (sg, id_s, id_e)$

1: assign every $e \in E$ its information gain $InfoGain$
2: $E = E \setminus \{e \mid e.InfoGain = 0\}$
3: // remove follow-up infections:
$E = E \setminus \{e \mid \exists p : p \in E, p.sg = e.sg, p.id_e = e.id_s, p.InfoGain = e.InfoGain\}$
4: $E_{(m_s,m_e)} = \{e \mid e \in E \wedge e.id_s.label = m_s \wedge e.id_e.label = m_e\}$
5: $P_{\text{freq}}(m_s, m_e) = \max_{e \in E_{(m_s,m_e)}} (e.InfoGain)$
6: $E_m = \{e \mid e \in E \wedge e.id_s.label = m\}$
7: $P_{\text{freq}}(m) = \max_{e \in E_m} (e.InfoGain)$

Besides the information gain ($InfoGain$, see Definition 2.7), we could have chosen various different algorithms originally designed for feature selection. In preliminary experiments, we have evaluated a number of such techniques with the result that those based on entropy are best suited for defect localisation, and that information gain produces the best results for our particular dataset we use in Section 5.4. Concretely, we have run experiments with the following feature-selection algorithms besides information gain (1–3 are based on entropy, too):

1. *Information-gain ratio* ($GainRatio$, see Definition 2.7)
2. *Symmetrical uncertainty* [WF05]
3. The OneR decision-stump classifier [WF05]
4. The *chi-squared statistic* (see, e.g., [WF05])
5. Relief [Kon94]
6. An *support vector machine (SVM) based algorithm* [GWBV02]

Follow-Up Infections

Call graphs of failing executions frequently contain infection-like patterns which are caused by a preceding infection. We call such patterns *follow-up infections* and remove them from our ranked list of features. Figure 5.1 illustrates a follow-up infection: (a) represents a defect-free version, (b) contains a defect in method a where it calls method d. Here, this method is called 20 times instead of twice. Following our reduction technique, this leads to the same (or a proportional) increase in the number of calls in method d. In our entropy-based ranking, the edges $d \to e$ and $d \to f$ inherit the score from $a \to d$ if the scaling of the weights is proportional. Thus, we interpret these two edges as follow-up infections and remove them from our ranking. More formally, we remove edges if the edge leading to its direct parent within the same subgraph has the same entropy score (Line 3 in Algorithm 5.2). In case of more than one defect in a programme, this way of follow-up infection detection might not find all such infections, but preliminary experiments have shown that it does detect common cases efficiently. We leave aside the pathological case that this technique classifies a real infection as follow-up infection. This is acceptable, since the probability of a certain entropy value is the same for every defect. Therefore, it is very unlikely that two unrelated defects lead to exactly the same entropy value, which would lead to a *false positive* classification.

From the Invocation-Level to the Method-Level

Until now, we calculate likelihoods of method invocations to be defective for every invocation (described by a calling method m_s and a method called m_e). We call this score $P_\mathrm{freq}(m_\mathrm{s}, m_\mathrm{e})$, as it is based on the call frequencies. To do the calculation, we first determine sets $E_{(m_\mathrm{s},m_\mathrm{e})}$ of edges $e \in E$ for every method invocation in Line 4

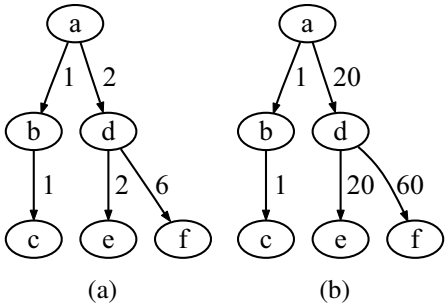


Figure 5.1: Follow-up infections.

of Algorithm 5.2. In Line 5, we 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, we assign every invocation the score from the context ranked highest. Other invocations with lower values might not be related to the defect.

Example 5.3: An edge from a to b is contained in two subgraphs. In one subgraph, this edge $\mathrm{a} \rightarrow \mathrm{b}$ 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 defect 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. We report this information additionally. To ease comparison with other approaches not providing this information, we also calculate $P_{\text{freq}}(m)$ for every calling method m in Lines 6 and 7 of Algorithm 5.2. The explanation is analogous to the one of the calculation of $P_{\text{freq}}(m_{\text{s}}, m_{\text{e}})$ in Lines 4 and 5.

5.3.2 Combined Approaches

As discussed before, structural approaches are well suited for the localisation of *structure-affecting bugs*, while frequency-based approaches focus on *call frequency-affecting bugs*. To be able to localise a broader range of defects, it seems to be promising to combine both approaches. In the following we first introduce a new structural score for combinations before we discuss combination strategies.

A Structural Score for Combination

The notion of the *specific neighbourhood* (SN) as introduced by Di Fatta et al. [DFLS06] (see Section 5.2.1) has the problem that no support can be calculated when the SN is empty.[1] Furthermore, preliminary 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 (see Section 5.3.1), we define another structural score. It is based on the set of frequent subgraphs which occur in failing executions only, SG_{fail}. We calculate the structural score P_{fail} as the support of m in SG_{fail}:

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

This is the support of all graphs containing method m in SG_{fail}.

Combination Strategies

As a first combination strategy, we combine the frequency-based approach with the P_{SN} score (see Section 5.2.1). In order to calculate the resulting score, we use the approach from Di Fatta et al. [DFLS06] without temporal order: We use the $R_{01\text{m}}^{\text{unord}}$ reduction with a general graph miner, gSpan [YH02] (see Section 2.3.3), in order to calculate the structural P_{SN} score. We derive 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 [YH03] (as described before). In order to combine the two scores derived from the results of two graph-mining runs, we calculate the arithmetic mean of the normalised scores:

$$P_{\text{comb}}^{\text{SN}}(m) := \frac{P_{\text{freq}}(m)}{2 \max\limits_{n \in V(sg), sg \subseteq g \in D}(P_{\text{freq}}(n))} + \frac{P_{\text{SN}}(m)}{2 \max\limits_{n \in V(sg), sg \subseteq g \in D}(P_{\text{SN}}(n))}$$

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

As this combined approach requires two costly graph-mining executions, we have introduced the structural score P_{fail} as a basis for a simpler combined defect-localisation approach. It requires only one graph-mining execution: We combine the frequency-based score with the P_{fail} score, both based on the results from one CloseGraph execution. Concretely, we combine the results with the arithmetic mean, as before:

$$P_{\text{comb}}^{\text{subtree}}(m) := \frac{P_{\text{freq}}(m)}{2 \max\limits_{n \in V(sg), sg \subseteq g \in D}(P_{\text{freq}}(n))} + \frac{P_{\text{fail}}(m)}{2 \max\limits_{n \in V(sg), sg \subseteq g \in D}(P_{\text{fail}}(n))}$$

[1] [DFLS06] uses a simplistic fall-back approach to deal with this effect.

5.4 Experimental Evaluation

We now evaluate the different proposals for call-graph reductions (see Chapter 4) and localisation techniques introduced in this section. In Section 5.4.1, we describe the experimental setup, and in Section 5.4.2 we present the experimental comparison of call-graph-based techniques. In Section 5.4.3, we compare these techniques to related work from software engineering.

5.4.1 Experimental Setup

Methodology

Many of the defect-localisation 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 defect 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 localise a defect within a method, this measure linearly quantifies the intellectual effort to find a defect. 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. This is in-line with the methodology of related studies (e.g., [JH05]). If the first defect 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 programme.

Programme under Test and Defects

As we rely on Java and AspectJ instrumentations in this dissertation, our experiments feature a Java programme. Concretely, we use a well-known diff tool taken from [Dar04], consisting of 25 methods and 706 lines of code (LOC). We instrumented this programme with 14 different defects which are artificial, but mimic defects which occur in reality and are similar to the defects used in related work. In particular, we have examined the *Siemens Programmes* [HFGO94] which are used in many related publications on dynamic defect localisation (see Section 3.1.2) and have identified five types of defects which are most frequent within them:

1. Wrong variable used
2. Off-by-one (e.g., `i+1` instead of `i` or vice versa)
3. Wrong comparison operator (e.g., >= instead of >)
4. Additional conditions
5. Missing conditions

Our programme versions contain these five types of defects. The *Siemens Programmes* mostly contain defects in single lines and just a few programmes with more than one defect. To mimic the *Siemens Programmes* as close as possible, we have instrumented only two out of 14 versions (defects 7 and 8) with more than one defect. We give an overview of the kinds of defects used in Table 5.2.

We have executed each version of the programme 100 times with different input data. Then we have classified the executions as $correct$ or $failing$ with a test oracle based on a defect-free reference programme.

Design of the Experiments

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 Section 4.1.5, we have compared *reduction techniques* based on the compression ratio achieved. How do the different reduction techniques perform in terms of defect-localisation 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?

version	description
defect 1, defect 10	wrong variable used
defect 2, defect 11	additional or-condition
defect 3	>= instead of !=
defect 4, defect 12	i+1 instead of i in array access
defect 5, defect 13	>= instead of >
defect 6	> instead of <
defect 7	a combination of defect 2 and defect 4 (in the same line)
defect 8	i+1 instead of i in array access + additional or condition
defect 9, defect 14	missing condition

Table 5.2: Defects used in the evaluation.

In concrete terms, we compare the following five alternatives:

$E_{\mathbf{01m}}$ The structural P_{SN}-scoring approach [DFLS06] (see Section 5.2), based on the unordered $R_{\text{01m}}^{\text{unord}}$ reduction.

$E_{\mathbf{subtree}}$ Our frequency-based P_{freq}-scoring approach (see Section 5.3.1) based on the R_{subtree} reduction.

$E_{\mathbf{comb}}^{\mathbf{SN}}$ Our combined approach with the $P_{\text{comb}}^{\text{SN}}$ scoring (see Section 5.3.2), based on the $R_{\text{01m}}^{\text{unord}}$ and R_{subtree} reductions.

$E_{\mathbf{comb}}^{\mathbf{subtree}}$ Our combined approach with the $P_{\text{comb}}^{\text{subtree}}$ scoring (see Section 5.3.2), solely based on the R_{subtree} reduction.

$E_{\mathbf{total}}$ The combined approach as before, but with the $R_{\text{total}}^{\text{w}}$ reduction [LYY+05] (with weights but without temporal edges, see Section 5.2).

For all experiments relying on the CloseGraph algorithm we use a minimum support $supp_{\text{min}}$ of 3. This allows for relatively large result sets, even when the graph database is relatively small. Large result sets prevent the approaches relying on the P_{fail} score (experiments $E_{\text{comb}}^{\text{subtree}}$ and E_{total}) to have the same score for many methods, which would lower the quality of the ranking.

5.4.2 Experimental Results

We present the results (the number of the first position in which a defect is found) of the five experiments for all 14 defects in Table 5.3. We represent a defect which is not discovered with the respective approach with '-'. Note that with the frequency-based and the combined method rankings, there usually is additional information available where a defect is located within a method, and in the context of which subgraph it appears. The following comparisons leave aside this additional information.

Structural, Frequency-Based and Combined Approaches

When 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 analysing numerical call frequencies is adequate to localise defects. Defects 1, 9 and 13 illustrate that both approaches alone cannot find certain defects. Defect 9 cannot be found by comparing call frequencies (E_{subtree}). This is because defect 9 is a modified condition which always leads to the invocation of a certain method. In consequence, the call frequency is always the same. Defects 1 and 13 are not found with the purely structural approach (E_{01m}). Both are typical call-frequency-affecting defects: Defect 1 is in an `if`-condition inside a loop and leads to more invocations of a certain method. In defect 13, a modified `for`-condition

exp. \ defect	1	2	3	4	5	6	7	8	9	10	11	12	13	14
E_{01m}	-	3	1	3	2	4	3	1	1	6	4	4	-	4
E_{subtree}	3	3	1	1	1	3	3	1	-	2	3	3	3	3
$E^{\text{SN}}_{\text{comb}}$	1	3	1	2	2	1	2	1	3	1	2	4	8	5
$E^{\text{subtree}}_{\text{comb}}$	3	2	1	1	1	2	2	1	18	2	2	3	3	3
E_{total}	1	5	1	4	3	5	5	2	-	2	5	4	6	3

Table 5.3: Experimental results.

slightly changes the call frequency of a method inside the loop. With the $R^{\text{unord}}_{\text{01m}}$ reduction technique used in E_{01m}, defect 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 $E^{\text{SN}}_{\text{comb}}$ and $E^{\text{subtree}}_{\text{comb}}$ 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 $E^{\text{subtree}}_{\text{comb}}$, 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. $E^{\text{SN}}_{\text{comb}}$ turns out to be better in four cases while $E^{\text{subtree}}_{\text{comb}}$ is better in six ones. Thus, the technique in $E^{\text{subtree}}_{\text{comb}}$ is slightly better, but not with every defect. Furthermore, the technique in $E^{\text{SN}}_{\text{comb}}$ 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 ($R^{\text{unord}}_{\text{01m}}$). Besides the increased precision of the localisation techniques based on the reduction, R_{subtree} also produces smaller graphs than $R^{\text{unord}}_{\text{01m}}$, which is good for scalability and runtime (see Section 4.1.5).

E_{total} evaluates the total reduction technique. We use $R^{\text{w}}_{\text{total}}$ as an instance of the total reduction family. The rationale is that this one can be used in the same setup as $E^{\text{subtree}}_{\text{comb}}$. In most cases, the total reduction (E_{total}) performs worse than the subtree reduction ($E^{\text{subtree}}_{\text{comb}}$). 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, $R^{\text{w}}_{\text{total}}$ can be an alternative to R_{subtree}.

Temporal Order

The experimental results listed in Table 5.3 do not shed any light on the influence of the temporal order. When applied to the defective programmes used in our comparisons, the total reduction with temporal edges ($R^{\text{tmp}}_{\text{total}}$) 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 $E_{01\text{m}}$ with the $R^{\text{ord}}_{01\text{m}}$ reduction and the FREQT [AAK+02] 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 $E_{01\text{m}}$. Only in two of the 14 defects the temporal-aware approach has performed better than $E_{01\text{m}}$, in the other cases it has performed worse. In a comparison with the R_{subtree} reduction and the gSpan algorithm [YH02] (see Section 2.3.3), the $R^{\text{ord}}_{01\text{m}}$ reduction with the ordered tree miner displayed a significantly increased runtime by a factor of 4.8 on average.[2] Therefore, our preliminary result based on the defects used in this section is that the incorporation of the temporal order does not increase the precision of defect localisations.

5.4.3 Comparison to Related Work

So far, the existing call-graph-based techniques [CLZ+09, DFLS06, LYY+05] have not been compared to the well-known techniques from software engineering discussed in Chapter 3.[3] We now compare our best-performing approach, $E^{\text{subtree}}_{\text{comb}}$, to the Tarantula technique [JHS02], to two of its improvements [AZGvG09] and to the SOBER method [LFY+06] (see Section 3.1.2 for details). These techniques can be seen as established defect-localisation techniques as they have outperformed a number of competitive approaches (see Section 3.1.2).

For the experiments in this section, we have implemented Tarantula, its improvements and SOBER for our programme used in the evaluations in this chapter. We have done so as no complete implementations are publically available. For SOBER, there is MATLAB source code available from the authors that performs the statistical calculations. However, there is no tool for the instrumentation of predicates available. For SOBER we have therefore implemented an automatic instrumentation, and we have reimplemented the statistical calculations in Java. For Tarantula and its improvements, we have implemented both steps, automated instrumentation and the calculations.

[2] In this comparison, FREQT was restricted as in [DFLS06] 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 is used instead of gSpan.

[3] Only [CLZ+09] has been compared to the sequence-mining-based approach RAPID [HJO08] which party builds on the well-known Tarantula technique [JHS02].

Both techniques, Tarantula and SOBER, work on granularities that are finer than the method level used by our approach (see Section 3.1.2). However, the Tarantula authors describe a means of mapping the results to the method level [JH05]. Concretely, the authors assign the score from its highest ranked basic block to the method. For our comparisons we rely on this mapping, and we do the same for SOBER, which originally works on the predicate level.

In our experiments with Tarantula, we have noticed that it happens frequently that methods have the same likelihood score, which worsens the results from the approach. Although the authors have not applied this technique in the original evaluations [JH05, JHS02], we use the $brightness$ score from Tarantula (see Section 3.1.2) as a secondary ranking criterion. This is, we let this criterion decide the ranking position in case the original score is the same for some methods. This approach seems to be natural, as the brightness would be a secondary source of information for a developer who uses the original visualisation from Tarantula.

When looking at Tarantula and our approach from a theoretical perspective, our approach considers more data than code coverage as utilised by Tarantula, but at the coarser method level. The information analysed by our approach additionally to the information analysed by Tarantula includes (1) call frequencies, (2) subgraph contexts and (3) the information which method has called another one. This data is potentially relevant for defect localisation, e.g., to localise *frequency-affecting bugs* (1) and *structure-affecting bugs* (2, 3). We therefore expect well results from our approach in comparison to Tarantula.

As discussed before (see Section 3.1.2), SOBER overcomes some of the shortcomings of previous approaches mentioned. It analyses the frequencies of predicate evaluations and is therefore better suited than Tarantula to localise *frequency-affecting bugs*. However, it does not analyse subgraph contexts as our approach does, but its predicate analysis takes information into account that we do not consider (e.g., return-value predicates). It is therefore hard to formulate theoretical expectations whether SOBER or our approach will perform better.

In the following, we compare our $E_{\text{comb}}^{\text{subtree}}$ approach (values taken from Table 5.4) to Tarantula in experiments $E_{\text{Tarantula}}$ and $E_{\text{Tarantula}}^{\text{b}}$ (with and without the brightness score), to the *Jaccard coefficient* variations in experiments E_{Jaccard} and $E_{\text{Jaccard}}^{\text{b}}$, to the *Ochiai coefficient* variations in experiments E_{Ochiai} and $E_{\text{Ochiai}}^{\text{b}}$ and to SOBER in experiment E_{SOBER}.

Table 5.4 contains the results from the comparison. The table clearly shows that our approach ($E_{\text{comb}}^{\text{subtree}}$) performs best in 12 out of the 14 defects and as well best on average. Only for defect 1 some of the other approaches perform a little better, and for defect 9 all other approaches perform better than $E_{\text{comb}}^{\text{subtree}}$. The explanation for the latter is as before: Defect 9 does not affect the call frequencies at all which are the most important evidence for our approach. The other comparisons are as expected: $E_{\text{Tarantula}}^{\text{b}}$ leads to better results than $E_{\text{Tarantula}}$, and E_{Jaccard} and E_{Ochiai} perform better than $E_{\text{Tarantula}}$ (as in [AZGvG09]). In our case, the $brightness$ extension does not

exp. \ defect	1	2	3	4	5	6	7	8	9	10	11	12	13	14	∅
$E_{\text{comb}}^{\text{subtree}}$	3	**2**	**1**	**1**	**1**	**2**	**2**	**1**	18	**2**	**2**	**3**	**3**	**3**	**3.1**
$E_{\text{Tarantula}}$	6	8	5	6	7	9	8	6	5	6	9	9	11	9	7.6
$E_{\text{Tarantula}}^{\text{b}}$	**1**	6	**1**	3	7	7	6	3	5	6	7	7	11	7	5.5
E_{Jaccard}	**1**	6	**1**	3	6	7	6	4	**4**	5	7	7	11	7	5.4
$E_{\text{Jaccard}}^{\text{b}}$	**1**	6	**1**	3	6	7	6	4	**4**	5	7	7	11	7	5.4
E_{Ochiai}	**1**	6	**1**	3	6	7	6	4	**4**	5	7	7	11	7	5.4
$E_{\text{Ochiai}}^{\text{b}}$	**1**	6	**1**	3	6	7	6	4	**4**	5	7	7	11	7	5.4
E_{SOBER}	3	3	6	3	3	4	3	3	**4**	5	4	4	9	4	4.1

Table 5.4: Comparison to related work (bold face indicates the best experiments).

improve the results of E_{Jaccard} and E_{Ochiai}. This can be explained by the fact that E_{Jaccard} and E_{Ochiai} are initially a lot better than $E_{\text{Tarantula}}$, which has more potential for improvements. In our experiments, the results of E_{Jaccard} and E_{Ochiai} do not differ. This is as well not unexpected, as in [AZGvG09] the *Ochiai coefficient* is only in very few cases better than the *Jaccard coefficient*. E_{SOBER} in turn performs on average better than all Tarantula variations, although there are few defects where E_{SOBER} performs worse. This is consistent to the results in [LFY+06].

Looking at the average values, a developer has to consider 3.1 methods when using our approach ($E_{\text{comb}}^{\text{subtree}}$). In contrast, when using the best performing approach from the related work considered in this comparison, SOBER (E_{SOBER}), the developer would have to consider 4.1 methods. Based on the benchmark defects used in these experiments, our approach therefore reduces the effort for defect localisation by 24% compared to SOBER.

Besides Tarantula and SOBER, we have also experimented with the static FindBugs [AHM+08] defect-localisation tool (see Section 3.1.1). However, FindBugs was not able lo localise any defect in our 14 defective programme versions. This is not surprising, as the defects of our benchmark (listed in Table 5.2) mostly represent defects affecting the programme logic rather than defect-prone programming patterns that can be identified by FindBugs.

5.5 Subsumption

The experiments in this chapter have shown that our approach performs well (regarding localisation precision) compared to both related approaches based on call-graph mining and established approaches from software engineering. However, as in the related work [CLZ+09, DFLS06, LYY+05], our evaluation is based on a relatively small number of defects and it is hard to draw conclusions for arbitrary defects in arbitrary programmes. Nonetheless, the defects in our evaluation serve as a bench-

mark. According to Zeller [Zel09], it is likely that a technique that performs better than another one on a benchmark will perform better on other programmes, too.

More concretely, our experiments presented in this chapter as well as the ones in the closely related work [CLZ+09, DFLS06, LYY+05] suffer from two issues related to the question if the results can be generalised:

- The experiments are based on artificially seeded defects. Although these defects mimic typical defects as they occur in reality, a study with real defects from an existing software project would emphasise the validity of the techniques. (This also applies to most of the techniques described in Section 3.1.2, as the evaluations rely on the *Siemens Programmes* [HFGO94] featuring artificial defects.)

- All experiments feature rather small programmes containing the defects (i.e., roughly ranging from 200 to 700 LOC). The programmes rarely consist of more than one class and represent situations where defects could be found relatively easy by a manual investigation as well. (This also applies to most of the techniques described in Section 3.1.2.) The approaches considered here will probably not scale without any further effort for programmes that are much larger than the programmes considered currently.

In the remainder of this dissertation we tackle these two issues. In Chapter 6, we investigate a scalable solution that builds on the techniques proposed in this chapter. We evaluate the approach with real defects from an open-source software project that is two orders of magnitude larger than the programmes in the evaluations considered so far. Furthermore, we present a constraint-based approach in Chapter 8, which leads to better scalability of the underlying graph-mining algorithms.

So far, we have not considered multithreaded programmes in our evaluations. In Appendix A, we present and evaluate a variation of the technique presented in this chapter for the localisation of defects in such programmes.

6 Hierarchical Defect Localisation

In the previous chapter, we have presented our approach for defect localisation (Section 5.3). Despite good results, we have identified two issues of this approach, namely *poor scalability* with the size of the software project and a desired *evaluation with real defects* (see Section 5.5). Both issues as well apply to the closely related work [CLZ+09, DFLS06, LYY+05]. In this chapter, we aim at generalising our approach for defect localisation to scale for larger software projects. To this end, we propose a hierarchical procedure that works with call graphs at different levels of granularity. We furthermore evaluate our new approach with real defects from a real-world software project.

We first present an introductory overview in Section 6.1. Sections 6.2 and 6.3 explain the call-graph representations we use in this chapter and defect localisation based on them, respectively. Section 6.4 contains the evaluation, and Section 6.5 is a subsumption of this chapter.

6.1 Overview

In this chapter, we aim at a scalable method for call-graph-mining-based defect localisation and at an evaluation with real defects. Solving the scalability issues is challenging, as seemingly possible solutions have issues: (1) Using increased computing capabilities or distributed algorithms is not feasible due to exploding computational costs. We have experienced this effect in preliminary experiments as well. Further, spending a lot of computing time for graph mining might be inappropriate for defect localisation. (2) Solving the scalability issue with approximate graph-mining algorithms might be a solution, but might miss patterns which are important for defect localisation. For instance, [CLZ+09] (see Section 5.2.3) does not report any problems caused by the better scaling LEAP algorithm [YCHY08] (see Section 3.2.2), but does not analyse large programmes either.

A different starting point to deal with the scalability problem in call-graph-based defect localisation is the graph representation. In this chapter, we investigate graph representations at coarser abstractions than the method level (see Section 4.1), i.e., the package level and the class level, and we start at such a coarse abstraction before *zooming-in* into a suspicious region of the call graphs. These graphs are a lot smaller than conventional method-level call graphs, and they cause scalability problems in much fewer cases. However, this idea leads to new challenges:

1. Call-graph representations have not yet been studied for levels of abstraction higher than the method level. How do representations well-suited for defect localisation look like?

2. When *zooming-in* into defect-free regions by accident, the following question arises: How to design hierarchical defect localisation in a way that minimises the amount of source code to be inspected by humans?

3. It is unclear which defects can indeed be localised in coarse graph representations.

Our approach for hierarchical defect localisation builds on the *zoom-in* idea and solves these challenges. It relies on weighted call graphs, making the localisation of certain defects a lot easier. In more detail, this chapter makes the following contributions:

Granularities of Call Graphs. We define *call graphs at different levels of granularity*, featuring edge-weight tuples that provide further information besides the graph structure (challenge 1). We do so by taking the specifics of defect localisation into account: We explicitly consider API calls as well as inter-/intra-package and inter-/intra-class method calls.

Hierarchical Defect Localisation. We describe the *zoom-in operation* for call graphs, present a *methodology for defect localisation* for the graphs at each level and describe *hierarchical procedures for defect localisation* (challenge 2). In concrete terms, we present different variants of a depth-first search strategy to hierarchically mine a software project.

Evaluation with a Large Software Project. An essential part of this chapter is the evaluation featuring *real programming defects* in Mozilla Rhino (challenge 3). To this end we use the iBUGS repository [DZ09] and the original test suite. Rhino consists of ≈ 49k LOC, and the defects in the repository were obtained by joining information from a bug-tracking system with data and source code from a revision-control system.

Ideas related to *zooming-in* into call graphs, namely *Graph OLAP*, have been described in [CYZ+09]. The authors propose data-warehousing operations to analyse graphs, e.g., *drill-down* and *roll-up* operations, similar to our *zoom-in* proposal. However, [CYZ+09] does not help in defect localisation, as it aims at interactive analyses, and it does not consider specific requirements (e.g., API calls).

6.2 Dynamic Call Graphs at Different Levels

In this section, we propose and define totally-reduced call-graph representations for the method, class and package level (Sections 6.2.1–6.2.3). These representations build on the totally-reduced weighted call graphs ($R^{\text{w}}_{\text{total}}$) we have introduced in Section 4.1.1. Then we introduce the *zoom-in* operation for call graphs (Section 6.2.4).

The call graphs introduced in this section can easily be extended in either direction: More coarse-grained *meta-package-level call graphs* could rely on the hierarchical organisation of packages and would allow to analyse even larger projects. Graphs more detailed than the method level, e.g., at the level of basic blocks (see Section 4.2), would allow for a finer defect localisation.

As in Chapter 4, we rely on AspectJ [KHH+01] to weave tracing functionality into Java programmes and to derive call graphs from programme executions. This yields an unreduced call-graph representation at the method level. (Figure 6.1(a) is an example.) This is the basis for all reduced representations we discuss in the following. In concrete terms, our tracing functionality internally stores unreduced call graphs in a pre-aggregated space-efficient manner. This lets us derive call-graph representations at any levels of granularity.

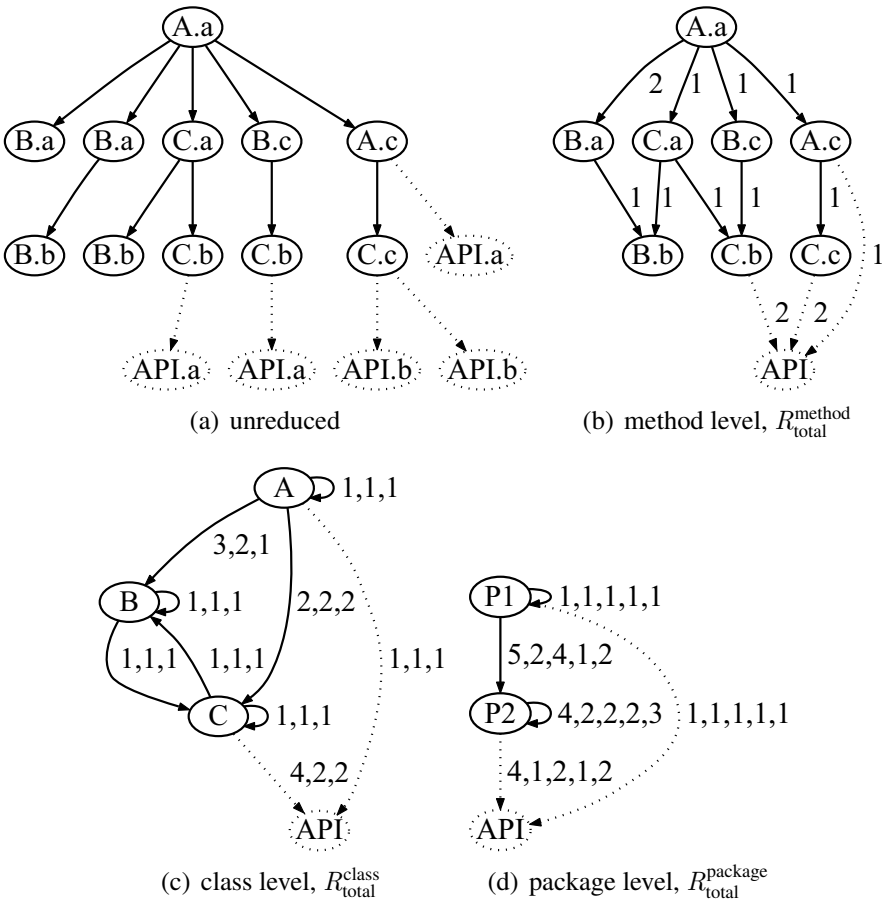
6.2.1 Call Graphs at the Method Level

We now propose total graph reductions that are weighted, where exactly one node represents a method. Furthermore, we do not make use of any temporal information. All this leads to a compact graph representation (see Section 4.1).

As an innovation, we consider calls of methods belonging to the Java class library (API) in all graphs. We do so as we believe that some defects might affect the calls of such methods. To our knowledge, no previous study has considered such method calls. However, to keep the instrumentation overhead to a minimum, we do not consider API-internal method calls. In the graph representation, we use one node (API) to represent all methods belonging to the class library.

Notation 6.1 (Method-level call graphs, $R^{\text{method}}_{\text{total}}$)
In method-level call graphs, every method is represented by exactly one node, directed edges represent method invocations, and edge weights stand for the frequencies of the calls represented by the edges. The API node represents all methods of the class library and does not have any outgoing edges.

Example 6.1: Figure 6.1(b) is a method-level call graph. It is the reduced version of the graph in Figure 6.1(a). The API nodes in Figure 6.1(a) represent two API methods, a and b, represented by one node in Figure 6.1(b). Both graphs do not have any self-loops, as the corresponding programme execution does not involve any recursive method calls.



(a) unreduced (b) method level, $R^{\text{method}}_{\text{total}}$

(c) class level, $R^{\text{class}}_{\text{total}}$ (d) package level, $R^{\text{package}}_{\text{total}}$

Figure 6.1: An unreduced call graph and its total reduced representations at the method level, class level and package level. Notation: $class.method$; class A forms package $P1$, classes B and C form package $P2$.

6.2.2 Call Graphs at the Class Level

We now propose class-level call graphs with tuples of weights. The rationale is to include some more information, which would otherwise be lost by the more rigorous compression.

Notation 6.2 (Class-level call graphs, $R^{\text{class}}_{\text{total}}$)
In class-level call graphs, every class is represented by exactly one node, and edges represent inter-class method calls (or intra-class calls in case of self-loops). The API node is as in R^{method}_{total} graphs. An edge is annotated with a tuple of weights: (t, u, v). t refers to the total number of method calls represented by the edge (as in R^{method}_{total} graphs), u is the number of different methods invoked, and v is the number of different methods that invoke methods.

Example 6.2: Figure 6.1(c) is a class-level call graph, it is the compression of the graphs in Figures 6.1(a) and (b). Class-level call graphs may include self-loops (except for the API node), even if there is no recursion.

6.2.3 Call Graphs at the Package Level

The reduction for this level is analogous to the previous ones, but to capture more information, we extend the edge-weight tuples by two elements:

Notation 6.3 (Package-level call graphs, $R^{\text{package}}_{\text{total}}$)
In package-level call graphs, there is one node for each package, and there is an additional API node. The edge-weight tuples are as follows:

$$(t_m, u_c, u_m, v_c, v_m)$$

where u_c is the number of different classes called, v_c the number of different classes calling, and t_m, u_m, v_m are as t, u, v in Notation 6.2. ('m' stands for method, 'c' for class.)

Example 6.3: We assume that class A in Figure 6.1 forms package $P1$, that classes B and C are in package $P2$, and that methods $API.a$ and $API.b$ belong to the same class. Figure 6.1(d) then is a package-level call graph, representing the call graphs from Figures 6.1(a)–(c).

6.2.4 The *Zoom-In* Operation for Call Graphs

Before we discuss the *zoom-in* operation for call graphs, we first define an auxiliary function:

Definition 6.1 (The generate function)
The generate function is of the following type:

$$generate_{level} : (\mathbb{G}_{unreduced}, \mathbb{V}) \to \mathbb{G}_{level}$$

where $\mathbb{G}_{unreduced}$ stands for unreduced call graphs, $\mathbb{G}_{level}$ for call graphs of the level specified by $level \in \{method, class, package\}$ and $\mathbb{V}$ for sets of vertices. $A \in \mathbb{V}$ specifies the area to be included in the graph to be generated, by means of a set of vertices of the package level (package names) in case '$level = class$' or of the class level (class names) in case '$level = method$'. In case '$level = package$', $A = \star$ selects all packages.

From a given unreduced graph, the function generates a subgraph at the level specified, containing all nodes contained in A (all nodes if $A = \star$) and edges connecting these nodes. If $A \neq \star$, the function introduces a new node labelled '$Dummy$' in the subgraph generated that stands for all nodes not selected by A.

In the *generate function*, we treat the API nodes separately from other nodes. They do not have to be explicitly contained in A, but are contained in the resulting graphs by default, as described in Notations 6.2 and 6.3. As the *generate function* selects certain areas of the graph, it obviously omits other areas. This is a conscious decision, as small graphs tend to make graph mining scalable. As calls of methods in the omitted areas might indicate defects nevertheless, the *generate function* introduces the $Dummy$ nodes to keep some information about these methods.

To *zoom-in* to a finer level of granularity, say into a certain package $p \in V(G_\mathrm{p})$ of a package-level call graph G_p to obtain a class-level call graph G_c, one calls the *generate function* as follows: $G_\mathrm{c} := generate_{class}(G_\mathrm{u}, \{p\})$, where G_u is the unreduced call graph of G_p. Zooming from a class-level call graph to a method-level call graph is analogous.

6.3 Hierarchical Defect Localisation

We now describe our hierarchical approach for defect localisation. At first, we introduce defect localisation without considering the hierarchical procedure, i.e., we describe how defect localisation works for call graphs at any selected level of granularity (Section 6.3.1). Note that this is a generalisation of the procedure described in Section 5.3.1. We then present different approaches for turning this technique into a hierarchical procedure (Section 6.3.2), which are further generalisations.

6.3.1 Defect Localisation in General

We now discuss defect localisation with call graphs at arbitrary levels of granularity. This is in principle a synopsis of our approach in Section 5.3.1 with generalisations

for arbitrary levels of abstraction and with adjustments for the graphs introduced in Section 6.2. After a short overview of the approach, we describe subgraph mining and defect localisation based on edge-weight tuples. Finally, we discuss the incorporation of information from static source-code analysis.

Overview

Algorithm 6.1 works with unreduced call graphs U (traces), representing programme executions. More specifically, it deals with graphs at a user-defined $level$, describing a certain subgraph of the graphs (parameter A). For the time being, we consider the package level ($A = \star$), i.e., without restricting the area. The algorithm first assigns a $class \in \{correct, failing\}$ to every graph $u \in U$ (Line 3), using a test oracle. Such oracles are typically available [JH05]. Then the procedure generates reduced call graphs, from every graph u (Line 4). Next, the procedure derives frequent subgraphs of these graphs, which provide different contexts (Line 6). The last step calculates a likelihood of containing a defect, for every software entity e at the $level$ specified (i.e., a package, class or method; Line 7). We do so by deriving a discriminativeness measure for the edge-weight-tuple values, in each context separately. The P values for all entities of a certain $level$ form a ranking of the entities, which can be given to software developers. They would then review the suspicious entities manually, starting with the one which is most likely to be defective. Alternatively, this result can be the basis for a *zoom-in* into a finer level of granularity, as described in Section 6.3.2.

Algorithm 6.1 Procedure of defect localisation.

Input: a set of unreduced call graphs U, a $level \in \{package, class, method\}$, an area A

Output: a ranking based on each software entity's likelihood to be defective $P(e)$

1: $G = \varnothing$ // initialise a set of call graphs
2: **for all** graphs $u \in U$ **do**
3: check if u refers to a correct execution, and assign a $class \in \{correct, failing\}$ to u
4: $G = G \cup \{generate_{level}(u, A)\}$
5: **end for**
6: $SG = frequent\_subgraph\_mining(G)$
7: calculate $P(e)$ for all software entities e at the $level$ specified, based on SG

In this chapter focussing on hierarchical mining, we do not rely on any structural score nor combinations as we have done in Section 5.3.2. We do so as preliminary experiments have revealed that structural scores do not work so well with totally reduced graphs from the particular software project used in the evaluation of this chapter. This is as the call graphs from several executions of the same programme tend to frequently have the same topology. Compared to the graphs we have used

before (e.g., in the R_{subtree} representation), the graphs we use now are less interesting from a structural point of view, but encode relevant information in the edge weight tuples.

Subgraph Mining

As in our approach in Section 5.3.1, the frequent-subgraph-mining step (Line 6 in Algorithm 6.1) mines the pure graph structure and ignores the edge-weight tuples for the moment. Later steps will make use of them. As before, we use the subgraphs obtained as different *contexts* and perform all further analyses for every subgraph context separately.

For subgraph mining, we rely on the ParSeMiS implementation [PWDW09] of the CloseGraph algorithm [YH03], which we have already used in Section 5.3.1. We now use a minimum support value of $\min(|G_{\text{corr}}|, |G_{\text{fail}}|)/2$, 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 that no structure occurring in at least half of all executions belonging to the smaller class is missed. Preliminary experiments have shown that this minimum support allows for both short runtimes and good results.

The API and $Dummy$ nodes as well as *self-loops* ($\circlearrowright$) require a special treatment during subgraph mining:

- *API nodes:* As almost all methods call API methods, almost every node in a call graph has a connection to the API node. This increases the number of edges in a call graph significantly, compared to a graph without API nodes, possibly leading to scalability issues. At the same time, as almost every node has an edge to an API node, these edges usually are not interesting for defect localisation. We therefore omit these edges during graph mining, but keep the edge-weight tuples for the subsequent analysis step. This is, only nodes and edges drawn with solid lines in Figure 6.1 are considered.

- *Dummy nodes:* We treat $Dummy$ nodes in the same way as we treat API nodes, as their structural analysis with subgraph mining does not seem to be promising. $Dummy$ nodes tend to be connected to many other nodes as well, leading to unnecessarily large graphs.

- *Self-loops ($\circlearrowright$):* Such edges result from recursion at the method level. However, at the package and class level, a self-loop represents calls within the same entity, which happens frequently. Therefore, self-loops enlarge the graph significantly while not bearing much information. We therefore treat self-loops at the package and class level as API and $Dummy$ nodes: We omit them during graph mining and keep the edge-weight tuples for subsequent analysis.

exec.	sg_1																					sg_2					class
	$A{\to}B$			$A{\to}C$			$A\circlearrowleft$			$B\circlearrowleft$			$C\circlearrowleft$			$A{\to}API$			$C{\to}API$			$B{\to}C$			... ($\circlearrowleft$,API)	⋯	
	t	u	v	t	u	v	t	u	v	t	u	v	t	u	v	t	u	v	t	u	v	t	u	v			
g_1	3	2	1	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	4	2	2	1	1	1	⋯	⋯	*correct*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋱	⋮
g_n	9	1	1	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	4	2	2	-	-	-	⋯	⋯	*failing*

Table 6.1: Example feature table for class-level call graphs.

Edge-Weight-Based Defect Localisation

When graph mining is completed, we calculate the likelihood that a method contains a defect (Line 7 in Algorithm 6.1). This is analogous to our approach in Section 5.3.1. Note that the description of an edge is now easier (i.e., without node ids), as we deal with totally reduced graphs where node names are unique. This also leads to the effect that each subgraph has maximally one embedding in a graph. We therefore do not have to use any average values. Concretely, we assemble a feature table as follows:

Notation 6.4 (Feature tables for defect localisation at arbitrary levels of abstraction) *Our feature tables have the following structure: The rows stand for all programme executions, represented by their call graphs. For every edge in every frequent subgraph, there is one column for every edge-weight-tuple element (i.e., a single call frequency t or tuples of values, depending on the granularity level of the call graph considered, see Section 6.2). For all edges leading to API and $Dummy$ nodes as well as for all self-loops ($\circlearrowleft$), there are further columns for the edge-weight-tuple elements; again, for each subgraph separately. The table cells contain the edge-weight-tuple values, except for the very last column, which contains the $class \in \{correct, failing\}$. If a subgraph is not contained in a call graph, the corresponding cells have a null value ('-').*

We do not include $Dummy$ nodes in the tables when considering the method level, as preliminary experiments have shown that this does not lead to any benefit. However, we include API nodes and *self-loops* at all levels.

Example 6.4: Table 6.1 is a feature table corresponding to class-level call graphs, such as the one in Figure 6.1(c). (This graph is execution g_1 in the table.) Suppose that the preceding graph-mining step has found two subgraphs, sg_1 ($B \leftarrow A \rightarrow C$) and sg_2 ($B \rightarrow C$). The very first column lists the call graphs $g \in G$. The next column corresponds to sg_1 and edge $A \rightarrow B$ with the total call frequency t. The following two columns correspond to the remaining two edge-weight tuple elements u and v (see Notation 6.2). Then follows the second edge in the same subgraph ($A \rightarrow C$) with its edge-weight tuple (t, u, v). Next, all self-loops ($A\circlearrowleft$, $B\circlearrowleft$, $C\circlearrowleft$) and API calls ($A \rightarrow API$, $C \rightarrow API$) in sg_1 are listed. ($Dummy$ nodes would be listed here

as well, but do not exist in this example.) The same columns for subgraph sg_2 and finally the class of the execution follow. Graph g_n does not contain sg_2, which is indicated by '-'.

After assembling the feature table, we employ the *information-gain* feature-selection algorithm ($InfoGain$, see Definition 2.7) in its Weka implementation [HFH+09] to calculate the discriminativeness of the columns and thus of the different edge-weight-tuple values. This is again analogous to our approach in Section 5.3.1.

So far, we have derived defect likelihoods for every column in the table. However, we are interested in likelihoods for software entities (i.e., packages, classes or methods), and every software entity corresponds to more than one column in general. To obtain the defect likelihood $P(e)$ of software entity e, we assign every column to the calling software entity. We then calculate $P(e)$ as the maximum of the $InfoGain$ values of the columns assigned to e. By doing so, we identify the defect likelihood of a software entity by its most suspicious invocation. The call context of a likely defective software entity and suspicious columns are supplementary information which we report to software developers to ease debugging.

Example 6.5: The graphs g_1 (see Figure 6.1) and g_n in Table 6.1 display similar values, but refer to a correct and a failing execution. Suppose that method $A.a$ contains a defect with the implications that (1) method $B.c$ will not be called at all, and (2) that method $B.a$ will be called nine times instead of twice. This is reflected in columns 2–4, referring to (t, u, v) of $A \to B$ in sg_1. t increases from three ($1 \times B.c$, $2 \times B.a$) to nine ($9 \times B.a$), u decreases from two ($B.c$, $B.a$) to one ($B.a$), and v stays the same – in class A, only method a invokes other methods. The $InfoGain$ measure will recognise fluctuating values of t and u, leading to a high ranking of class A.

Incorporation of Static Information

The edge-weight and $InfoGain$-based ranking procedure sometimes has the minor drawback that two or more entities (i.e., packages, classes or methods) have the same ranking position. In such cases, we fall back to a second ranking criterion: We sort such entities decreasingly by their size in (normalised) lines of code (LOC) derived with LOCC [Joh00]. The rationale is that the size frequently correlates with the defectiveness likelihood [NBZ06] (see Section 3.1.1). This is, large methods tend to be more defective.

6.3.2 Hierarchical Procedures

The defect-localisation procedure described in Section 6.3.1 can already guide a manual debugging process: A developer can first do defect localisation at the package level. She or he can then decide to *zoom-in* into certain suspicious packages. The developer would continue with our defect-localisation technique at the class level,

proceeding with the method level etc. However, it might happen that the developer *zooms-in* into an area where no defect is located. In this case, the developer would backtrack and *zoom-in* elsewhere etc. This manual process, guided by our technique, bears the potential that important background knowledge known to the developer can be easily included.

In this section, we say how to turn the manually-guided debugging process into semi-automatic procedures for defect localisation. We present a depth-first-search-based (DFS-based) procedure, a so-called merge-based variant and a parameter-free variant. We also propose a technique that partitions large packages and classes.

DFS-Based Defect Localisation

Our DFS-based procedure follows the idea to manually investigate the most suspicious method in the most suspicious class in the most suspicious package first. If this first method turns out to not be defective, we go to the second most suspicious method in the same class. If all methods in this class are investigated, we backtrack to the next class etc. We further propose the parameters k, l, m. They limit the number of software entities to be investigated at each stage, to k packages, l classes and m methods. Algorithm 6.2 formalises this approach. The parameters k, l, m can be set to infinity in order to obtain a parameter-free algorithm; at the end of this section we also present a means to set these parameters.

Algorithm 6.2 iterates through three loops, one for packages, one for classes and one for methods (Lines 3, 6 and 9). In each loop, the algorithm calculates a defectiveness likelihood P for the respective software entities. This is, Lines 1–2, 4–5 and 7–8 comprise the graph-mining step (Line 6 in Algorithm 6.1) and the step that calculates P (Line 7 in Algorithm 6.1), as described in Section 6.3.1. These lines make use of the *generate function* (see Definition 6.1), each with the area selection based on the currently selected software entity at the respective coarser level. Ultimately, the algorithm presents suspected methods to the user and terminates in case the user has identified a defect (Lines 10–12).

The DFS-based procedure described works interactively. This is, the potentially expensive graph-mining step as well as the calculation of P are done only when needed – the algorithm might terminate before all packages and classes have been analysed. The suspected methods are presented to the user in an on-line manner. This avoids long runtimes before a developer actually can start debugging. However, it is of course possible to skip Lines 10–12 in Algorithm 6.2 and to save the current $method$ to an ordered list of suspected methods. This leads to a ranking as described in Section 6.3.1. To ease experiments, we follow this approach in our evaluation.

The proposed approach obviously has the drawback that the user has to set the parameters k, l, m. When the values are too low, the technique might miss a defect. Based on our experience, it is not hard to set appropriate parameters based on empirical values derived from debugging other defects in the same project. Furthermore,

Algorithm 6.2 DFS-Based Defect Localisation.

Input: a set of classified ($correct$, $failing$) unreduced call graphs U,
parameters k, l, m

Output: a defective $method$

1: SG = *frequent_subgraph_mining*$(\{generate_{package}(u, \star) \mid u \in U\})$
2: calculate $P(package)$, based on SG
3: **for all** $package \in top_k(P(package))$, ordered decreasingly by $P(package)$ **do**
4: SG = *frequent_subgraph_mining*$(\{generate_{class}(u, \{package\}) \mid u \in U\})$
5: calculate $P(class)$, based on SG
6: **for all** $class \in top_l(P(class))$, ordered decreasingly by $P(class)$ **do**
7: SG = *frequent_subgraph_mining*$(\{generate_{method}(u, \{class\}) \mid u \in U\})$
8: calculate $P(method)$, based on SG
9: **for all** $method \in top_m(P(method))$,
ordered decreasingly by $P(method)$ **do**
10: present $method$ to the user
11: **if** $method$ is defective **then**
12: **return** $method$
13: **end if**
14: **end for**
15: **end for**
16: **end for**

we will present an automated choice of optimal parameter values in the following paragraphs.

Merge-Based Variant of DFS-Based Defect Localisation

This technique is an alternative to the DFS-based one. Instead of presenting the results to the user in an on-line manner, it replaces Lines 10–12 in Algorithm 6.2 with code that saves all methods processed (along with their likelihood P) in a result set. Then, right after Line 12 in Algorithm 6.2, it sorts all methods decreasingly by their defect likelihood.

The drawback of this procedure is that the algorithm has to terminate before one can actually start debugging. On the other side, we hypothesise that the defect localisations obtained by this merge-based variant are better than the ones with the first approach. We evaluate this hypothesis in Section 6.4.

Concerning the parameters k, l, m, the merge-based variant is more robust. As the merged result set is sorted at the very end, large parameter values usually do not lead to worse localisation results. They only affect the runtime.

Parameter-Free Variant of Merge-Based Defect Localisation

As yet another variant, we propose *parameter-free defect localisation*. Here we set the parameters k, l, m in the merge-based variant to infinity. This promises to not miss any defective method. In addition, if one uses this variant several times with a certain software project, one can use it to empirically set the parameter values. This allows for an efficient usage of the interactive (on-line) DFS-based procedure without parameters that are too high or to speed up the regular merge-based variant.

Partitioning Approach

The hierarchical procedures investigated in this dissertation analyse small *zoomed-in* call graphs at several granularities. However, a number of software projects – especially large ones and those with a long history – have imbalanced sizes of packages and classes. This might lead to large graphs that cause scalability issues, even if we are considering a *zoomed-in* subgraph only. It is an open research question how to overcome such situations. For now, we present a sampling-based partitioning approach for such cases.

Whenever a certain call graph at the package or class level is too large to be handled, we partition the graph into two (or, if needed more) partitions. We do so by randomly sampling nodes from the graph. We keep the edges connecting two nodes within the same partition. As not all edges connect nodes belonging to the same partition, we would lose a lot of information. To compensate for this effect, we introduce a dummy node $Dummy_{\text{part}}$ in each partition, representing all nodes in other

partitions. We treat $Dummy_{\text{part}}$ nodes in exactly the same way as $Dummy$ nodes, i.e., we omit them during graph mining and include the edge-weight-tuple values in the feature tables.

When graph partitions are generated and $Dummy_{\text{part}}$ nodes are inserted, we do defect localisation as described before with each partition separately. Then, similarly to the merge-based variant, we merge the rankings obtained from the different partitions and obtain a defectiveness ranking ordered by the P values of the software entities. This lets us proceed with any manual or automated hierarchical defect localisation procedure, as described before.

This partitioning approach for large packages and classes has worked well in preliminary experiments. However, there might be cases where a loss of relevant information exists, and defect localisation might not work. For instance, think of a defect which occurs in a certain subgraph context that is distributed over several partitions. In such situations, the defect-localisation procedure can be repeated with a different partitioning, either based on the expertise of a software developer or by using another seed for random partitioning.

6.4 Evaluation with Real Software Defects

We now evaluate our defect-localisation techniques in order to demonstrate their effectiveness and usefulness for large software projects. After a description of the target programme and the defects (Section 6.4.1) we explain the evaluation measures used (Section 6.4.2). Then we focus on defect localisation at the different levels in isolation (Section 6.4.3). Finally, we evaluate the hierarchical defect-localisation approaches (Section 6.4.4).

6.4.1 Target Programme and Defects: Mozilla Rhino

For our evaluation we rely on Mozilla Rhino, as published in the iBUGS project [DZ09]. Rhino is an open-source JavaScript interpreter, consisting of nine packages, 146 classes and 1,561 methods or ≈ 49k LOC (normalised 37k LOC). iBUGS provides a number of original defects that were obtained by joining information from the bug-tracking system of the project with data and source code from its revision-control system. Furthermore, it contains the original test cases along with the test oracles. See [DZ07] for details on how the data was obtained. All in all, Rhino from the iBUGS repository provides a realistic test scenario for defect localisation in a large software project. At least compared to programmes used in related evaluations [CLZ+09, DFLS06, LYY+05] and in Section 5.4 of this dissertation that are two orders of magnitude smaller, Rhino can be considered to be a relatively large software project.

level \ defect	85880	114491	114493	137181	157509	159334	177314	179068	181654	181834	184107	185165	191668	194364	∅
package	2	1	1	4	2	1	1	1	3	1	3	1	3	2	1.9
class	1	8	8	16	20	1	2	15	5	1	1	2	3	3	6.1
method	1	2	2	-	1	1	1	10	3	3	9	10	1	2	3.5

Table 6.2: Defect-ranking positions for the three levels separately.

Concretely, we make use of 14 defects (Table 6.2 lists the defect numbers) from the iBUGS Rhino repository which have associated test cases and represent *occasional bugs*. These defects represent different real programming errors, and they are hard to localise: They occur occasionally and have been checked-in into the revision-control system before a failing behaviour has been discovered. See the iBUGS repository [DZ09] for more details. In addition, iBUGS provides about 1,200 test cases consisting of some JavaScript code to be executed by Rhino, together with the corresponding oracles. As in many software projects, there are only a few failing test cases for each defect, besides many passing cases. To obtain a sufficient number of failing cases, we have generated new ones by varying existing ones. In concrete terms, we have merged JavaScript code from correct and failing test cases.

6.4.2 Evaluation Measures

In order to assess the precision of our techniques, we consider the ranking positions of the actual defects. These positions quantify the number of software entities (i.e., packages, classes and methods) a software developer has to investigate in order to find the defect. As the sizes of methods can vary significantly, we deem it more adequate to assess the hierarchical approaches by considering the normalised LOC rather than only the number of methods involved. We therefore provide the percentage of LOC to examine in addition to the ranking position. We calculate the percentage as the ratio of methods that has to be examined in the software project, i.e., the sum of LOC of all methods with a ranking position smaller than or equal to the position reported, divided by the total LOC.

6.4.3 Experimental Results (Different Levels)

We now present the defect-localisation results for the three different levels. This is, we consider complete package-level call graphs and call graphs at the class and method level, *zoomed-in* into the correct package (and class). We do so in order to assess the defect-localisation abilities for every level in isolation.

Table 6.2 contains the experimental results, the ranking positions for all defects investigated, separately for the three levels. Figure 6.2 provides a graphical representation of the same data. It plots the number of defects localised when a developer

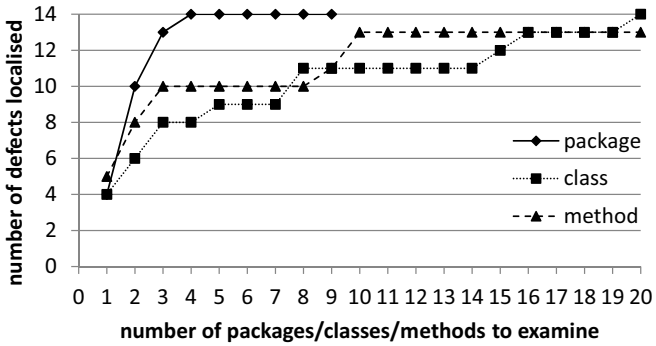


Figure 6.2: The numbers of defects localised when examining a certain number of packages/classes/methods.

examines a certain number of the top-ranked entities. For example, the third triangular point from the left means that 10 out of 14 defects are localised when examining up to three methods.

At the *package level*, the defective package is ranked at position one or two in 10 out of 14 cases, i.e., localisation is precise. The explanation for such good results at the coarsest level is the small number of nine packages in Rhino. At the *class level*, the results look a little worse at first sight. However, eight defects can be localised when examining three classes or less (out of 146). Only three defects are hard to localise, i.e., a developer has to inspect 15 or more classes. At the method level, 13 of the defects can be localised by examining 10 methods or less (out of 1,561), 10 of them with three methods or less. Only one defect, number 137181, cannot be localised at all. This defect does not affect the call-graph structure nor the call frequencies.

All in all, the call-graph representations at the different levels – as well as the localisation technique – localise most defects with a high precision. However, when using package-level call graphs to manually *zoom-in* into a package, packages ranked at position three or four might be misleading. This is not unexpected, as it is well known that many defects have effects only in their close neighbourhood [DZ07]. This might not affect a package-level call graph at all. The hierarchical approaches, in particular the merge-based ones, try to overcome this effect by investigating several packages systematically.

We use the results from this section to set the parameters k, l, m for the hierarchical approaches. The maximum localisation precision in Figure 6.2 is reached at four packages, 20 classes or 10 methods. When using these values as parameters, the hierarchical approaches do not miss any defects they could actually localise while avoiding to examine more source code than necessary.

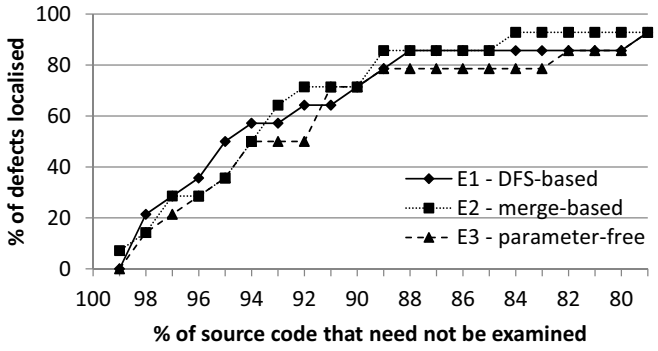


Figure 6.3: The percentage of defects localised when not examining a certain percentage of source code.

6.4.4 Experimental Results (Hierarchical)

We now present the results from three experiments with the different hierarchical approaches (see Section 6.3.2):

E1 DFS-based defect localisation
E2 Merge-based variant of DFS-based defect localisation
E3 Parameter-free variant of merge-based defect localisation

Table 6.3 contains the numerical results in two variants: the ranking positions at the method level and the corresponding percentage of source code. As before, Figure 6.3 is a graphical representation of this data. Similarly to related work (e.g., [JH05, LFY+06]), it represents the percentage of defects localised versus the percentage of source code that does not need to be examined.

In line with our hypothesis (see Section 6.3.2), the merge-based variant (E2) performs better than the pure DFS-based approach (E1) in all but four data points in Figure 6.3. The average values in Table 6.3 reflect this as well. With the merge-based variant (E2), one finds a defect by examining 6.1% of the source code on average. Not surprisingly, parameter-free defect localisation (E3) always performs worse than or equal to the parameterised variant (E2). However, it still allows a developer to find defects by inspecting 7.5% of the source code on average, without having to set any parameters.

Focusing on the best approach, the merge-based variant (E2), two defects are pinpointed directly, and six defects can be localised by investigating less than 10 methods. Only one defect cannot be localised at all (as before), and for only two defects 100 or more methods need to be inspected. All in all, we deem these results very helpful: On average, almost 94% of the source code can be excluded from manual debugging, and to find 86% of all defects, one can skip 89% of the code.

exp. \ defect	85880	114491	114493	137181	157509	159334	177314	179068	181654	181834	184107	185165	191668	194364	∅
E1	3	9	56	-	170	3	5	120	64	3	54	29	55	15	45.1
E2	52	12	3	-	46	1	1	68	9	3	100	154	8	25	37.1
E3	54	18	3	-	49	1	1	77	14	5	155	210	8	31	48.2
E1	1.2%	4.5%	10.3%	-	20.6%	2.6%	1.6%	9.8%	5.1%	4.4%	7.5%	3.1%	11.6%	1.5%	6.4%
E2	6.7%	2.6%	5.4%	-	10.3%	2.6%	1.3%	7.5%	0.3%	4.4%	10.4%	15.2%	5.9%	6.7%	6.1%
E3	8.2%	3.4%	5.4%	-	10.5%	2.6%	1.3%	8.1%	1.9%	4.5%	17.8%	20.1%	5.9%	8.3%	7.5%

Table 6.3: Hierarchical defect-localisation results. E1: DFS-based defect localisation; E2: merge-based variant thereof; E3: parameter-free defect localisation. Top: method-ranking position; bottom: LOC to examine.

6.5 Subsumption

In this chapter, we have brought forward call-graph-mining-based defect-localisation (see Chapter 5) to a hierarchical and scalable procedure. Our evaluation has shown that it is able to localise defects from the field in a relatively large software project, Mozilla Rhino. The result from our experiments is that the amount of source code a developer has to examine manually can be reduced to about 6% on average. This shows that our call-graph-based approach is able to detect real defects from the field. Furthermore, the results show that we are able to reduce the source code to be investigated significantly. However, 6% in Rhino still refer to ≈ 3,000 LOC. When applied in the field, we expect that the domain knowledge from a software developer can further reduce the amount of code to be investigated. For instance, a developer might be able to exclude certain packages from inspection as she or he knows that the code is not related to the kind of failure.

In Section 5.4.3, we have compared our basic approach using a small programme to both related approaches/concepts that rely on call-graph mining [CLZ+09, DFLS06, LYY+05] and well-known and proven approaches from the software-engineering community [AZGvG09, JHS02, LFY+06]. The experiments gave way to the conclusion that our approach performs well compared to the other approaches. It would certainly be interesting to compare the performance of our hierarchical approach from this chapter to alternative approaches, too. This could be done within a more comprehensive evaluation of defect-localisation techniques with software repositories from large projects (see Chapter 9). However, regarding the related work based on call-graph mining, such a comparison would not be possible due to scalability problems. This would at least not be possible as long as one does not extend these approaches with a hierarchical procedure similar to the one proposed in this chapter. Regarding the defect-localisation techniques from software engineering, a comparison would be difficult. This is as no complete implementations are available (see Section 5.4.3). Furthermore, at least for the SOBER method [LFY+06], it is unclear if it would scale for software projects of the size of Rhino. Predicate-based instrumentation is expensive in terms of runtime, and we are not aware of any evaluations of SOBER featuring programmes of this size.

As Rhino was released as a benchmark for defect-localisation tools within the iBUGS suite [DZ09], we expect that more and more evaluations in the future will be based on Rhino and can be compared to our evaluation. So far we are only aware of one study featuring the Rhino dataset: The approach based on graphical models from Dietz et al. [DDZS09] (see Section 3.1.2) has used the same benchmark, but in an earlier version. However, this approach is rather unknown compared to defect-localisation techniques such as Tarantula and SOBER. Furthermore, as mentioned in Section 3.1.2, the results can hardly be compared to ours: The evaluation by the authors covers only situations where one considers up to 1% of the source code in

order to find a defect. Besides that, the published results suggest that their approach might be better than our approach, in this particular situation.

In the following, we aim at improving the defect-localisation precision further. In Chapter 7, we develop a technique that is able to localise an additional class of defects, namely those that affect the dataflow of a programme. This also helps in improving the defect-localisation precision of defects that can already be localised.

7 Localisation of Dataflow-Affecting Bugs

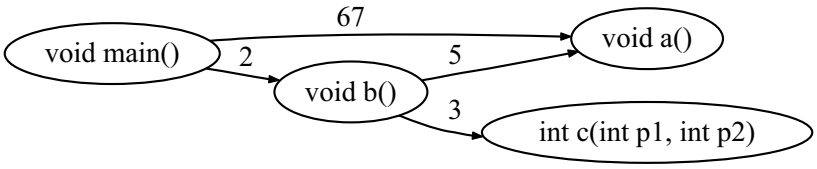
An important characteristic of the call-graph-based defect-localisation techniques discussed so far (both from the related work and introduced in this dissertation) is that they merely analyse the call-graph structure and the call frequencies. They can only localise defects which affect the call graph of a programme execution (simplified, the control flow). While this is an important class of defects, Cheng et al. [CLZ+09] point out that the current techniques are agnostic regarding defects that influence the dataflow. In this chapter, we present a technique to localise *dataflow-affecting bugs* by extending call graphs with information regarding the dataflow. For the graph representation and the localisation technique we build on concepts from the preceding chapters.

We first present an introductory overview in Section 7.1. Sections 7.2 and 7.3 then introduce dataflow-enabled call graphs (DEC graphs) and explain how we use them for defect localisation. Section 7.4 contains the experimental evaluation. Section 7.5 is a subsumption of this chapter.

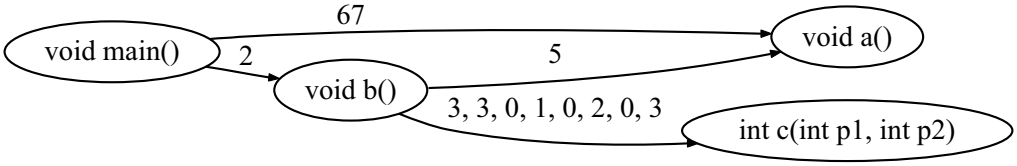
7.1 Overview

In this chapter, we present a call-graph-based technique which localises both *dataflow-affecting* and *call-graph-affecting bugs*. Dataflow-affecting bugs 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 infected value affects a control statement. Although this happens frequently, it might occur in methods which are actually defect-free, leading to erroneous localisations. In such cases, the incorporation of dataflow-related information into the call graphs and thus the analysis process can increase the localisation precision. In other cases, where defects affect the dataflow only, the incorporation of dataflow information is the sole possibility to capture such defects.

The specification of graphs that incorporate dataflow-related information 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 de-



(a) Call graph with call frequencies (not dataflow enabled).



(b) Dataflow-enabled call graph (DEC graph).

Figure 7.1: Example call graphs.

tail 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 chapter, 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 7.1(b) is a DEC graph corresponding to Figure 7.1(a). 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. 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 as before 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.

All in all, our technique for defect localisation that allows for the localisation of dataflow-affecting bugs 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.

A Defect-Localisation Approach for Dataflow-Affecting Bugs. We present a defect-localisation technique for DEC graphs. Similar to the previous chapters, it is an application of weighted graph mining, which ultimately identifies defective methods.

Results in Software Engineering. We demonstrate the appropriateness and precision of our DEC-graph-based approach for the localisation of defects. In a case study we evaluate the approach using defects introduced into the Weka machine-learning suite [HFH+09].

7.2 Dataflow-Enabled Call 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 7.3) 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 (Section 7.2.1). We then explain the dataflow abstractions and explain why they are useful for defect localisation (Section 7.2.2). Finally, we say how we obtain the graphs from programme traces and give a concrete example (Section 7.2.3).

7.2.1 Derivation of Programme Traces

As in the preceding chapters, we employ the aspect-oriented programming language AspectJ [KHH+01] to weave tracing functionality into Java programmes (see Section 4.4). For each method 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. [KKR10]. Certainly, these simplifications can be severe, but logging the full data would result in overly large amounts of data. Our evaluation (Section 7.4) primarily studies primitive data types. A systematic evaluation of

arrays, collections and strings as well as techniques for complex data types is beyond the scope of this dissertation, but is an interesting direction of future work.

Based on the experience from the previous chapters, we decide to make use of a *total-reduction* variant of call graphs. See Section 4.1.1 for details on the total-reduction scheme.

7.2.2 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.

Notation 7.1 (Edge-Weight Tuples)
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.

With the formal notation of edge-weight tuples (Notation 7.1), we are now able to introduce DEC graphs that are totally reduced graphs at the method level:

Notation 7.2 (Dataflow-Enabled Call Graphs (DEC Graphs))
In DEC graphs, every distinct method is represented by exactly one node. When one method has called another method at least once in an execution, a directed edge connects the corresponding nodes. These edges are annotated with numerical edge-weight tuples as introduced in Notation 7.1.

As we will see in the following, DEC graphs can only be derived for a number of executions, as meaningful discretisations need to be found that hold for all pro-

gramme executions considered. Figure 7.1(b) is an example DEC graph, we illustrate its construction in Example 7.1.

7.2.3 Construction of Dataflow-Enabled Call Graphs

We now explain how we derive the edge-weight tuples and construct *dataflow-enabled call graphs* (*DEC graphs*). The core task for the construction of DEC graphs is the discretisation of traced data values from a number of executions. The CAIM (class-attribute interdependence maximisation) algorithm [KC04] suits our requirements for intelligent discretisation: It (1) discretises single numerical attributes of a dataset, (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 Notation 7.1. Example 7.1 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 7.1: We consider the call of method c from method b in Figure 7.1(a) (execution 1 in Table 7.1) and three further programme executions (executions 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 7.1(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 7.1(b) for the exact intervals. The occurrences of elements of edge-weight tuples can then be counted easily – see Table 7.1(c), the discretised version of Table 7.1(a). The edge-weight tuple of $b \rightarrow c$ in execution 1 then is as displayed in Figure 7.1(b), 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 7.1: Example discretisation for the call of `int c(int p1, int p2)`.

7.3 Localising Dataflow-Affecting Bugs

We now explain how to derive defect localisations from DEC graphs. This is in principle the approach from Section 5.3.1, with adoptions for the dataflow-abstractions as introduced in Section 7.2. We first give an overview (Section 7.3.1), then we describe subgraph mining (Section 7.3.2) and the actual defect localisation (Section 7.3.3). Finally, we introduce three extensions to our approach (Sections 7.3.4, 7.3.5 and 7.3.6).

7.3.1 Overview

As in the earlier chapters, Algorithm 7.1 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.

7.3.2 Frequent Subgraph Mining

As shown in Line 6 in Algorithm 7.1 and as in the previous chapters, we use frequent subgraph mining to derive subgraphs which are frequent within the call graphs considered. We use these subgraphs as contexts for a more detailed analysis.

Algorithm 7.1 Procedure of defect localisation with DEC graphs.

Input: a set of programme traces $t \in T$
Output: a ranking based on each method's likelihood to be defective $P(m)$
1: $G = \varnothing$ // 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

Again, we rely on the ParSeMiS implementation [PWDW09] of CloseGraph [YH03] for frequent subgraph mining. For the minimum-support value, we use as in Section 6.3.1 $\min(|G_{\text{corr}}|, |G_{\text{fail}}|)/2$, where G_{corr} and G_{fail} are the sets of call graphs of correct and failing executions, respectively ($G = G_{\text{corr}} \cup G_{\text{fail}}$).

7.3.3 Entropy-Based Defect Localisation

Next, we calculate the likelihood that a method contains a defect (Line 7 in Algorithm 7.1). This is analogous to the previous chapters, with the exception that we now analyse the dataflow annotations, too. To this end, we assemble a feature table as follows:

Notation 7.3 (Feature tables for defect localisation with DEC graphs)
Our feature tables have the following structure: The rows stand for all programme executions, represented by their DEC graphs. For every edge in every frequent subgraph, there is one column for every edge-weight-tuple element, i.e., one column for the total call frequencies t and columns for all interval frequencies. These frequencies are normalised: They are divided by the corresponding t in order to obtain the ratio of calls falling into each interval. The table cells contain the call-frequency values and the normalised interval-frequency values. The very last column contains the $class \in \{correct, failing\}$. If a subgraph is not contained in a call graph, the corresponding cells now have value 0.

Example 7.2: Table 7.2 is an example table which assumes that two subgraphs were found in the previous graph mining step, sg_1 ($main \rightarrow b \rightarrow c$) and sg_2 ($main \rightarrow a$). The first column lists the call graphs $g \in G$. The second column corresponds to sg_1 and edge $main \rightarrow b$ 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

exec.	sg_1									sg_2		
	$main \to b$	$b \to c$								$main \to a$	$\cdots$	$class$
	t	t	$\frac{p_1^{i_1}}{t}$	$\frac{p_1^{i_2}}{t}$	$\frac{p_2^{i_1}}{t}$	$\frac{p_2^{i_2}}{t}$	$\frac{p_2^{i_3}}{t}$	$\frac{r^{i_1}}{t}$	$\frac{r^{i_2}}{t}$	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 7.2: Example feature table. g_1 refers to execution 1 from Example 7.1 (Figure 7.1(b)).

and return values. The very last column contains the class $correct$ or $failing$. g_n does not contain sg_2, and the corresponding cells have value 0.

After assembling the table, we employ the *information-gain-ratio* feature-selection algorithm ($GainRatio$, see Definition 2.7) in its Weka implementation [HFH+09] to calculate the discriminativeness of the columns and thus of the different edge-weight-tuple values. We have already successfully used the $GainRatio$ technique in Section 5.3.1. In comparison to $InfoGain$, $GainRatio$ reaches value 1 always when a column can perfectly tell classes apart. $InfoGain$ only reaches value 1 when in addition the class distribution is equal (see Section 2.3.2). This is an advantage of $GainRatio$ compared to $InfoGain$, as it makes it easier to interpret the value as a probability. In Section 7.4.3, we evaluate the usage of different feature-selection techniques.

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 7.3: The graphs g_1 and g_n in Table 7.2 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 $\frac{r^{i1}}{t}$ and $\frac{r^{i2}}{t}$ of $b \rightarrow c$ in sg_1. The $GainRatio$ measure will recognise fluctuating values in these columns, leading to a high ranking of method c.

The preceding example has illustrated 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 *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 7.2) – value 0 indicates subgraphs not supported by an execution. (2) Such defects are frequently caused by control statements (e.g., `if`, `for`) evaluating previously wrongly calculated values. Our analysis based on dataflow can detect such situations more directly.

7.3.4 Follow-Up-Infection Detection

Call graphs of failing executions frequently contain infection-like patterns which are caused by a preceding infection. As in Section 5.3.1, we employ a simple strategy to detect certain *follow-up infections* to enhance the method ranking. This strategy is an extension for Line 7 in Algorithm 7.1: We remove methods within the same subgraph belonging to a method call $m_2 \rightarrow m_3$ from the ranking when the following conditions hold: (1) $GainRatio(m_1 \rightarrow m_2) = GainRatio(m_2 \rightarrow m_3)$ (we consider the $GainRatio$ values from columns belonging to total call frequencies and parameters), and (2) $m_1 \rightarrow m_2 \rightarrow m_3$ 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. (Note that cycles can occur in totally reduced graphs but not in R_{subtree} graphs as used in Chapter 5.) However, as in Section 5.3.1, our detection is a heuristic, but it is helpful in practice (see Section 7.4).

7.3.5 Improvements for Structure-Affecting Bugs

The subgraphs mined in Line 6 in Algorithm 7.1 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:

$$P_{\text{corr}}(m) := \max_{m \in V(sg), sg \in SG} (supp(sg, G_{\text{corr}}))$$
$$P_{\text{fail}}(m) := \max_{m \in V(sg), sg \in SG} (supp(sg, G_{\text{fail}}))$$

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

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

Preliminary experiments have revealed that this kind of structural scoring leads to better results with the totally reduced graphs used in this chapter than the structural scoring function used in Section 5.3.2. To integrate P_{struct} into our $GainRatio$-based method ranking $P(m)$ (in Line 7 in Algorithm 7.1), we calculate the average:

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

7.3.6 Incorporation of Static Information

As in the previous chapter, static information can be used to improve the ranking accuracy. The starting point is the handling of methods with the same defect likelihood. As in related studies [JH05], we use the worst ranking position for all methods which have the same defect likelihood by default. As an extension, a second static ranking criterion helps distinguishing methods with the same defect likelihood: We sort such methods decreasingly by their size in normalised lines of code (LOC)[1]. Research has shown that the size in LOC frequently correlates with the defectiveness likelihood [NBZ06].

7.4 Experimental Evaluation

To investigate the defect-localisation capabilities of our approach, we use the Weka machine-learning suite [HFH+09], 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:

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.

Experiments E4–6 essentially are a comparison to the technique presented in Section 5.3 using $R^{\text{w}}_{\text{total}}$ graphs. We use the same localisation technique as with the DEC graphs for a fair comparison.

[1] In this dissertation, we use "method lines of code", the sum of non-blank and non-comment LOC inside method bodies, as derived with the Metrics eclipse plugin [Sau05].

We now describe the experimental setting in detail (Section 7.4.1) before we present the experimental results (Section 7.4.2). We further present some supplementary experiments (Section 7.4.3).

7.4.1 Experimental Setting

Weka is a data-intensive open-source application with a total of 19,938 methods and 255k lines of code (LOC). We now use Weka as our programme under test, as it heavily deals with data passed between methods, which is not the case in the previous evaluations in this dissertation. As we have done in Section 5.4.1, we introduce five different kinds of defects. They are of the same types as the defects in related evaluations, e.g., the *Siemens programmes* [HFGO94], which are often used to evaluate defect-localisation techniques for C programmes (see Section 3.1.2).

The defect types introduced to Weka are typical programming mistakes, are non-crashing, occasional and dataflow-affecting and/or call-graph-affecting. In total, we evaluate ten separate defects (defect 1–10) as well as six combinations of two of these defects (defects 11–16). These combinations mimic typical situations where a programme contains more than one defect.

We have introduced all defects in `weka.classifiers.trees.DecisionStump`. This class is the implementation of a decision-tree algorithm which comprises 18 methods or 471 LOC. 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. This is the reason why we can can analyse this rather large project without any hierarchical procedure (see Chapter 6).

We execute each defective version of Weka with 90 sets of sampled data from the UCI machine-learning repository [FA10] 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.

7.4.2 Experimental Results

We present the results – the ranking position which pinpoints the actual defect – of the six experiments for all sixteen defects in Table 7.3. This position quantifies the number of methods a software developer has to review in order to find the defect. 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 [JH05]. (We look at the results featuring the incorporation of static information as described in Section 7.3.6 at the end of this experimental evaluation section.)

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 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.

Regarding the Weka versions with two defects (defects 11–16), defect localisation always works better on average than for versions with only one defect (E1–10). 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.

exp. \ 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 7.3: Defect-localisation results. (E2/3/5/6) incl. follow-up, (E3/6) incl. structural ranking.

Improved Experimental Results using Static Analysis

When we apply the secondary static ranking criterion (see Section 7.3.6) 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.

7.4.3 Supplementary Experiments

We now present supplementary experiments that are not intended to demonstrate the usefulness of DEC graphs, but evaluate selected aspects from the defect-localisation technique. Concretely, we evaluate the feature-selection technique employed, and we evaluate one aspect of the feature tables. This aspect concerns the question whether *null values* or *value* 0 in the feature tables leads to better defect-localisation results.

In the preceding chapters, we have used the *information-gain technique* for feature selection ($InfoGain$), while we have used a related technique in this chapter, *information-gain ratio* ($GainRatio$). In Section 5.3, we have already described that both techniques lead to very similar results. However, gain ratio has the nice property that it reaches value 1 always when a column discriminates perfectly. Information gain in turn additionally requires a balanced class distribution to reach value 1 (see Section 2.3.2). This property from gain ratio might make it easier for software developers to interpret the resulting values as a probability to contain a defect.

In the feature tables used for defect localisation, it happens that certain columns can not be filled with values when a certain call graph (a row in the table) does not embed a certain subgraph (corresponding to columns). In these situations we have used a zero (‘0’) in this chapter and in Chapter 5, while we have used a null value (‘-’) in Chapter 6. Both alternatives are reasonable, as one can argue that a null value refers to not existing embeddings, and as one can likewise argue that a zero stands for zero method calls. We now evaluate these two alternatives.

In our supplementary experiments, we focus on defects 1–10 from the previous evaluation in this chapter. We do so, as defects 11–14 are combinations thereof, and as we want to study the pure results from defect localisation in the standard case with one defect. Table 7.4 contains the results from the supplementary experiments (the first line is taken from Table 7.3).

exp. \ defect	1	2	3	4	5	6	7	8	9	10	∅
(E1) with $GainRatio$ (as before)	3	3	1	3	2	2	12	3	1	1	3.1
(E1) with $InfoGain$	4	1	2	4	2	3	12	1	1	1	3.1
(E1) with $GainRatio$ & *null values*	4	4	5	9	2	7	8	6	1	1	4.7

Table 7.4: Supplementary experimental results.

Regarding $GainRatio$ and $InfoGain$, the results deviate a little between the individual defects. On average, the defect-localisation precision of both alternatives is equal. This is in line with our results in Section 5.3. Therefore, we consider both alternatives to be equally suited for defect localisation. However, the $GainRatio$ results might be a little more intuitive as they are always in the same interval (between 0 and 1).

Regarding the influence of null values instead of zeros in the feature tables, the picture is different. Despite of defect 7, where the null values lead to better defect localisations, the variant presented in this chapter (the first line in Table 7.4, referring to the usage of zeros in the feature tables) performs equal or better. This is clearly indicated by the increased average values for the null-value variant. The advantage of zeros can be explained by the fact that tuples containing null values are ignored when $InfoGain$ or $GainRatio$ is calculated. Therefore, more information that is potentially important for defect localisation is considered when using zeros.

7.5 Subsumption

The defect-localisation techniques investigated in the preceding chapters of this dissertation are agnostic regarding the dataflow. This is, they are not able to localise defects that affect the dataflow only, and they have difficulties localising defects that affect primarily the dataflow. In this chapter, we have extended our call-graph representations (see Chapter 4) with abstractions referring to the dataflow, resulting in dataflow-enabled call graphs (DEC graphs). Further, we have adopted our defect-localisation scheme (see Chapter 5) to deal with DEC graphs. With these extensions and adoptions we are able to localise a broader range of defects. DEC graphs can also be used within a hierarchical defect-localisation scheme as introduced in Chapter 6 without any special challenges.

Besides well defect-localisation results achieved with DEC graphs, there are a number of possible improvements for the technique:

- As mentioned before in Section 7.2.1, we have primarily studied dataflows through primitive data types. This is partly caused by the absence of respective defective programmes featuring other situations. However, a systematic evaluation of dataflows related to arrays, collections and strings would sub-

stantiate the results from this chapter. Furthermore, as mentioned, we currently do not deal with complex data types. However, one can define heuristics to incorporate such dataflows. Then, complex data types can be handled with our technique in the same way as we handle other data types.

- Besides the question which data types to investigate, not all kinds of dataflows are directly related to method calls. For instance, a dataflow can also be realised by interchanging data through global variables. Currently, our approach does not cover such situations. However, they might be integrated into our approach as follows: Static code analysis could help to identify relevant variables that are read within a method. They can then be treated like additional method-call parameters.

- Another starting point for further investigations is the evaluation of different discretisation algorithms. As described in Section 7.2.3, we have decided to employ the CAIM algorithm [KC04], as it suits our requirements and has outperformed a number of alternative algorithms in the evaluations by the authors. Furthermore, we have achieved well results with this kind of discretisation in our evaluation (see Section 7.4). However, other supervised discretisation techniques (i.e., discretisation of numerical data with respect to a class) have been described in the literature [CWC95, DKS95, FI93, Ker92, LS97, WC87, Wu96] and are in principle suited for our approach, too. Although we do not expect significant improvements in result accuracy, these alternatives could be evaluated. Besides the discretisation algorithms mentioned, decision-tree-induction algorithms [BK98, Qui93] with different parametrisations could be used for this task as well. When applied to one attribute only, they partition the value-range into intervals containing homogeneous values referring to the same class with an increased likelihood.

8 Constraint-Based Mining of Weighted Graphs

In the previous chapters, we have focused on software-defect localisation with call graphs. Concretely, we have discussed various data representations (call graphs) and data-mining techniques for their analysis. For the latter, we have so far followed a post-processing approach for mining weighted call graphs: We have analysed the weights in an analysis step that follows subgraph mining. We now investigate an integrated approach for weighted subgraph mining that brings together subgraph mining and the analysis of edge weights. We do so by proposing a constraint-based approach and by investigating its difficulties. We show that this approach can generally be used for various applications, including our software-defect-localisation setting.

In this chapter, we first present an introductory overview in Section 8.1. In Section 8.2, we introduce weight-based constraints, and in Section 8.3 we explain their integration into mining algorithms. Section 8.4 describes application settings. Section 8.5 contains the evaluation. Section 8.6 is a subsumption of this chapter.

8.1 Overview

Two general approaches for subgraph mining with weighted graphs are *preprocessing* and *postprocessing*. These strategies refer to the analysis of the weights: Are they analysed *before* of *after* the mining of the graph structure? However, both of these variants have issues: As discussed in Section 3.2.1, discretising numerical values during preprocessing might lose important information. Postprocessing (as investigated in Chapters 5–7) in turn is not always efficient: The mining algorithm first ignores the weights and might generate a huge number of subgraphs. The second step however discards most of them. Cheaper ways to perform frequent subgraph mining with weights are *approximate graph mining* (see Section 3.2.2) and *constraint-based mining* (see Section 3.2.3). In this chapter, we investigate *approximate frequent subgraph mining with weight-based constraints*. This is, we analyse the weights *during* the mining of the graph structure. Such a constraint-based approach is promising, since various higher-level analysis tasks imply meaningful weight-based constraints. In a classification scenario, to give an example, a natural constraint would demand weights in the subgraph patterns with a high discriminativeness. While constraints lead to smaller result sets, we hypothesise that those application-specific constraints

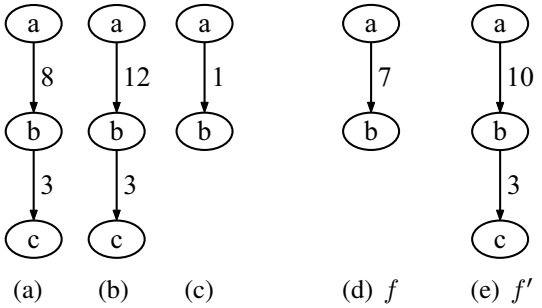


Figure 8.1: Example graphs.

do not lower the result quality of the higher-level problem. The same principle applies to our software-defect-localisation scenario. However, not every constraint is good for pruning in a straightforward way. Literature has introduced *anti-monotone constraints* (see Section 2.3.3 and Section 3.2). When using them for pruning, the algorithm still finds all patterns. However, most weight-based constraints are not anti-monotone, for the following reason: Graph topology and weights are independent of each other, at least in theory. Example 8.1 illustrates that weight-based properties of graphs may behave unpredictably when the support changes. Thus, pruning a pattern at a certain point bears the risk of missing elements of the result.

Example 8.1: Think of an upper-bound constraint defined as a numerical threshold t_{u} on the average weight of a certain edge $a \to b$ in all supporting graphs: $avg(a \to b) \leq t_{\mathrm{u}}$. This would prevent mining from expanding a pattern f where $avg(a \to b) > t_{\mathrm{u}}$. Now consider the graph database D consisting of (a)–(c) as well as pattern f in Figure 8.1. f is annotated with the average weight of the edges in D. If we now extend f by one edge, resulting in pattern f', the average weight increases from 7 to 10. Graph (c) causes this effect. It does not support f', and its weight value is below average.

Despite this adverse characteristic, we study frequent subgraph mining with non-anti-monotone weight-based constraints in this chapter. The rationale is that certain characteristics of real-world graphs give way to the expectation that results are good. Namely, there frequently is a correlation between the graph topology and the weights in real-world weighted graphs.

Example 8.2: Consider a road-map graph where every edge is attributed with the maximum speed allowed. Large cities, having a high node degree (a topological property), tend to have more highway connections (high edge-weight values) than smaller towns. This is a positive correlation.

In software engineering, a similar observation holds: Think of a node in a weighted call graph representing a small method consisting of a loop. This method tends to

invoke a few different methods only (low degree), but with high frequency (high weights). This is a negative correlation.

McGlohon et al. [MAF08] have studied a number of weighted graphs from different domains such as citation networks, social networks and computer-network-traffic networks. They have observed similar correlations as in Example 8.2. Concretely, they have formulated the so-called *weight power law (WPL)* and the *snapshot power law (SPL)*. The *WPL* links the total weight of a graph to the number of edges and to the number of nodes in the graph, each following a power law with exponents that are specific for a graph dataset. Even more interestingly, similar to our road-map example, the *SPL* describes a proportional relationship between the weights of outgoing edges to the out-degree of a certain node (and accordingly for incoming edges). This is again a power-law relationship with exponents that are specific for the graph dataset. However, all these observations in real-world graphs are in contrast to the property sketched before: In theory, weights might be independent from the graph structure. Therefore, although there is strong evidence that certain relationships between weights and graph topology exist, such relationships cannot be guaranteed for arbitrary graph datasets.

Motivated by the examples given in Example 8.2 and the observations from McGlohon et al. [MAF08] referring to real-world graphs, we propose the following approach for weighted subgraph mining:

Approach 8.1 (Approximate weight-constraint-based frequent subgraph mining)
Given a database of weighted graphs*, find subgraphs satisfying a minimum frequency constraint and user-defined constraints referring to weights.*

Note that the subgraphs returned are unweighted – weights are considered only in the constraints. In this chapter, we compose a constraint-based mining technique by integrating constraints referring to weights (that are not anti-monotone) into frequent-subgraph-mining algorithms. This leads to approximate results. We then investigate the following problem:

Problem 8.1
What is the completeness and the usefulness of results obtained from approximate weight-constraint-based frequent subgraph mining?

In concrete terms, we study the degree of *completeness* of mining results compared to non-constrained results. To assess the *usefulness* of an approximate result, we consider the result quality of higher-level analysis tasks, based on approximate graph-mining results as input.

To deal with this problem, this chapter features the following points:

Weight-Constraint-Based Subgraph Mining. We say how to extend standard pattern-growth algorithms for frequent subgraph mining with pruning based on

weight-based constraints. We do so for gSpan [YH02] and CloseGraph [YH03] (see Section 2.3.3).

Application to Real-World Problems. Besides our defect-localisation application, we describe further data-analysis problems that build on weighted graphs. We say how to employ weight-constraint-based subgraph mining to solve these problems.

Evaluation. We report on the outcomes of an evaluation featuring different domains and analysis settings. This includes our software-defect-localisation scenario as well as data and analysis problems from logistics. A fundamental result is that the correlation of weights with the graph structure indeed exists, and we can exploit it in real-world analysis problems.

8.2 Weight-Based Constraints

In this section, we define the weight-based constraints we investigate in this chapter. We do not deal with anti-monotone constraints, since we are interested in investigating approximate mining results from non-anti-monotone constraints. However, the techniques would work with anti-monotone constraints as well.

Definition 8.1 (Weight-based measures)
A weight-based measure *is a function $E(p) \rightarrow \mathbb{R}$ which assigns every edge of a graph pattern p a numerical value. The function takes the weights of the corresponding edges in all embeddings of p in all graphs in a graph database D into account.*

Depending on the actual problem, one can assign some numerical or categorical value such as a class label to each graph. In our software-defect-localisation scenario, these labels stand for correct and failing executions. Measures like $InfoGain$ and $PMCC$ make use of such values, in addition to the weights. – If labels are not unique, subgraphs can be embedded at several positions within a graph. We consider every single embedding of a subgraph to calculate a measure for an edge.

Definition 8.2 (Weight-based constraints)
A lower bound predicate c_l *for a pattern p is a predicate with the following structure:*

$$c_l(p) := (\exists\, e_1 \in E(p) : measure(e_1) > t_l) \vee (|p| < size_{min})$$

An upper bound predicate c_u *in turn is as follows:*

$$c_u(p) := (\nexists\, e_2 \in E(p) : measure(e_2) > t_u) \vee (|p| < size_{min})$$

A weight-based constraint*, applied to a pattern p, is a set containing c_l, c_u, or both, connected conjunctively.*

The lower- and upper-bound predicates let the user specify a minimum and maximum interestingness based on the *measure* chosen. We comment on the two predicates as well as on parameter $size_{\min}$ in Section 8.3. Note that Definition 8.2 requires to consider *all* edges of a pattern p. This is necessary, as illustrated in Example 8.1. The value of the measure of *any* edge of p can change when the set of graphs supporting p changes.

Weight-Based Measures

Any function on a set of numbers can be used as a measure. We have chosen to evaluate three measures with a high relevance in real data-analysis problems: $InfoGain$, $PMCC$ and $variance$. None of these measures is anti-monotone. Two of them, $InfoGain$ and $PMCC$, require the existence of a class associated with each graph. Such classes are available, e.g., in any graph-classification task, and the goal of the mining process is to derive subgraph patterns for a good discrimination between the classes. $variance$ does not depend on any class. It is useful in explorative mining scenarios where one is interested in subgraphs with varying weights.

Example 8.3: If one wants to search for patterns p with a certain minimum $variance$ of weights, one would specify the measure '$variance$', the threshold value t_1 and set $size_{\min}$ to 0. The constraint then is '$\exists\, e : variance(e) > t_1$'. This could be useful when analysing logistics data, where one wants to find subgraphs with unbalanced load or highly varying transportation times.

Although we have dealt with some of the measures in earlier chapters, we give a short summary of the three measures chosen in the following. Besides these measures, many further measures from statistics and data analysis can be used similarly to build weight-based constraints. This includes, say, different attribute-selection measures known from decision-tree induction [BK98].

Information Gain. The $InfoGain$ (see Definition 2.7) is a measure in the interval $[0, 1]$ and quantifies the ability of an attribute A to discriminate between classes in a dataset (without a restriction to binary classes). In the context of weighted graphs, A refers to the weights of a certain edge of a subgraph pattern in all embeddings in all graphs in the graph database D.

Pearson's Product-Moment Correlation Coefficient ($PMCC$). The *correlation coefficient* is widely used to quantify the strength of the linear dependence between two variables (see, e.g., [WF05]). In our graph-mining context, these two variables are the weight of a certain edge in a subgraph pattern in all embeddings in graphs in D and their binary classes. For our purposes, positive and negative corre-

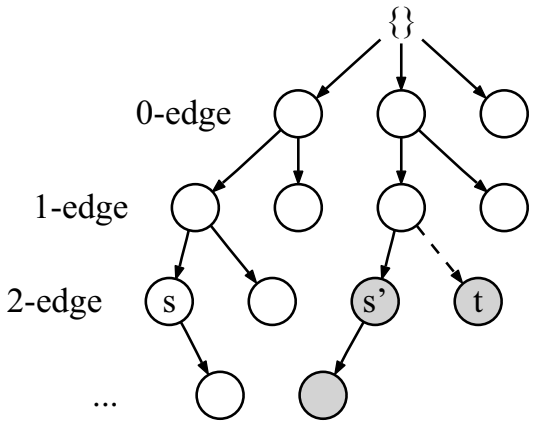


Figure 8.2: A pattern-growth search space with conventional isomorphism-based pruning (s') and constraint-based pruning (t, new in this dissertation).

lation have the same importance, and we use the absolute value. Then $PMCC$ is in the interval $[0, 1]$ as well.

Variance. The $variance$ quantifies the variation of the values of a random variable Y. It is in the interval $[0, \infty)$. In our scenarios, Y is the set of weights of a certain edge in all subgraph patterns in all embeddings in D.

8.3 Weight-Based Mining

We now describe how to integrate *weight-based constraints* into *pattern-growth-based frequent subgraph mining*. We first focus on vanilla pattern-growth algorithms before turning to closed mining. The basic idea is to use weight-based constraints – even if they are not anti-monotone – to prune the search space.

Example 8.4: Figure 8.2 illustrates pattern-growth mining with and without weight-based constraints. Without such constraints, s' and its successors are pruned, as s' is isomorphic to s. With weight-based constraints, the search is additionally pruned at pattern t. The dashed edge extends its parent, and t including the new edge violates a weight-based constraint. Note that it is not necessarily the newly added edge itself which violates the constraint, but any edge in t.

In concrete terms, we treat the lower and upper-bound predicates c_l and c_u (as defined in Definition 8.2) in weight-constraint-based mining as follows:

Approach 8.2
When a pattern p does not satisfy c_l or c_u, the search is pruned. If it is c_u that is not satisfied, p is added to the mining result, otherwise not.

Upper Bounds. The rationale behind an *upper bound* is to speed up mining by pruning the search when a sufficiently interesting edge weight is found. Therefore, we use it to prune the search, but save the current pattern. For example, if the user wants to use the graph patterns mined for classification, a pattern with one edge with a very discriminative weight will be fair enough. Clearly, larger graphs can still be more discriminative. Setting the threshold therefore involves a trade-off between efficient pruning and finding relevant graphs. Section 8.5.3 will show that small changes in the upper bound do not change the results significantly. It is therefore sufficient to rely on few different threshold values to obtain satisfactory results.

Lower Bounds. With a *lower bound*, the user specifies a minimal interestingness. This bound stops mining when the value specified is not reached. The rationale is that one does not expect to find any patterns which are more interesting. However, this might miss patterns. The parameter $size_{\text{min}}$ (see Definition 8.2) controls this effect.

Pattern-Growth Algorithms

Algorithm 8.1 describes the integration into pattern-growth-based frequent-subgraph-mining algorithms (see Section 2.3.3). The algorithm works recursively, and the steps in the algorithm are executed for every node in Figure 8.2. Lines 1–2, 9–13 and 20 are the generic steps in pattern-growth-based graph mining [YH06]. They perform the isomorphism test (Lines 1–2), add patterns to the result set (Line 9) and extend the current pattern (Line 11), leading to a set of frequent patterns P. The algorithm then processes them recursively (Lines 12–13) and stops depth-first search when P is empty (Line 20).

Lines 4–7 and 15–17 are new in our extension. Instead of directly adding the current pattern p into the result set F, the algorithm first checks the $size_{\text{min}}$ parameter (Line 4). Only if the minimum size is reached, it calculates the weight-based measures (Line 5). Line 7 checks the constraints (if c_l or c_u is not set, the thresholds are 0 or ∞, respectively; see Definition 8.2). If they are not violated, or the minimum size is not reached, the algorithm saves the pattern to the result set (Line 9) and continues as in generic pattern growth (Lines 12–13). Otherwise, the algorithm prunes the search, i.e., it does not continue the search in that branch. Note that this step is critical, as it determines both the speedup and the result quality. As mentioned before, we always save the last pattern before we prune due to upper bounds (Lines 16–17). This leads to result sets which are larger than those from standard graph mining when the constraints are applied in a postprocessing step.

One can realise constraints on more than one measure in the same way, by evaluating several constraints instead of one, at the same step of the algorithm. As mentioned before, mining with weight-based constraints produces a result set with unweighted subgraph patterns. In case one needs weighted subgraphs in the result set, arbitrary

Algorithm 8.1 *pattern-growth*$(p, D, supp_{\text{min}}, t_{\text{l}}, t_{\text{u}}, size_{\text{min}}, F)$

Input: current pattern p, database D, $supp_{\text{min}}$, parameters $measure$, t_{l}, t_{u} and $size_{\text{min}}$
Output: result set F

1: **if** $p \in F$ **then**
2: **return**
3: **end if**
4: **if** $|p| \geq size_{\text{min}}$ **then**
5: calculate weight-based measures for all edges
6: **end if**
7: **if** $(\exists\, e_1 : measure(e_1) > t_{\text{l}} \;\wedge\; \nexists\, e_2 : measure(e_2) > t_{\text{u}}) \vee (|p| < size_{\text{min}})$ **then**
8: **if** ($algorithm \neq$ *CloseGraph* $\vee$ p is closed) **then**
9: $F = F \cup \{p\}$
10: **end if**
11: $P =$ *extend-by-one-edge*$(p, D, supp_{\text{min}})$
12: **for all** $p' \in P$ **do**
13: *pattern-growth*$(p', D, supp_{\text{min}}, t_{\text{l}}, t_{\text{u}}, size_{\text{min}}, F)$
14: **end for**
15: **else**
16: **if** $\exists\, e : measure(e) > t_{\text{u}}$ **then**
17: $F = F \cup \{p\}$
18: **end if**
19: **end if**
20: **return**

functions, e.g., the average, can be used to derive weights from the supporting graphs in the graph database.

Closed Mining

Closed mining returns closed graph patterns only (see Section 2.3.3). When dealing with weight-based constraints, we deviate from this characteristic. We favour graphs which are interesting (according to the measures) over graphs which are closed. This is because the weight-based constraints might stop mining when 'interesting enough' patterns are found. Extending the CloseGraph [YH03] algorithm is slightly more complicated than pattern growth as described before. CloseGraph performs further tests in order to check for closedness (Line 8 in Algorithm 8.1). In our extension, these tests are done after weight-based pruning. Therefore, when the search is pruned due to a constraint, it might happen that the algorithm misses a larger closed pattern. In this case it adds patterns to the result set which are not closed.

Implementation

The extensions we describe here are compatible with any pattern-growth graph miner. We for our part use the ParSeMiS graph-mining suite [PWDW09] with its gSpan [YH02] and CloseGraph [YH03] implementations (see Section 2.3.3).

8.4 Weighted Graph Mining Applied

We now say how to exploit the information contained in the weights of graphs in different application scenarios building on weight-constraint-based frequent subgraph mining. Concretely, we first review our software-defect-localisation scenario from Chapter 5 (Section 8.4.1). Then we introduce weighted graph classification (Section 8.4.2) and exploitative graph mining (Section 8.4.3).

8.4.1 Software-Defect Localisation

In order to localise defects with our weight-constraint-based frequent-subgraph-mining technique, we alter the defect-localisation approach from Section 5.3.1 as follows: Instead of employing two separate analysis steps for frequent subgraph mining and weight analysis (Lines 6 and 7 in Algorithm 5.1, respectively), we perform a single weight-constraint-based subgraph-mining step. Our implementation calculates the values of the employed $measure$ for all edges in all frequent subgraphs by default, and we interpret these values as defectiveness likelihoods. We now use the $InfoGain$ measure instead of *gain ratio*, as $InfoGain$ leads to results of the same quality (see Section 7.4.3) and can be calculated more efficiently (see Definition 2.7).

As in Section 5.3.1, we then use the maximum value from all outgoing edges in all subgraphs in the result set as the weight-based likelihood of a method m:

$$P_{\text{w}}(m) := \max(measure(\{(m,x)|(m,x) \in \mathcal{E} \wedge x \in \mathcal{V}\}))$$

where $\mathcal{V}$ and $\mathcal{E}$ are the unions of the vertex and edge sets of all subgraph patterns in the result set, and $measure$ applied to a set calculates the $measure$ of every element separately.

Similar to our combined approach in Section 5.3.2, we look at the subgraph structures as well. The result sets mined with weight-based constraints let us define another likelihood based on support. They contain a higher number of interesting graphs with interesting edges (according to the measure chosen) than a result set from vanilla graph mining. Therefore, it seems promising not only to give a high likelihood to edges with interesting weights. We additionally consider nodes (methods) occurring frequently in the graph patterns in the result set. We calculate this structural likelihood similar to a support in the result set F:

$$P_{\text{s}}(m) := \frac{|\{f|f \in F \wedge m \in f\}|}{|F|}$$

The next step is to combine the two likelihoods. We do this by averaging the normalised values:

$$P^{\text{constr}}_{\text{comb}}(m) := \frac{P_{\text{w}}(m)}{2 \max\limits_{n \in V(sg), sg \subseteq g \in D}(P_{\text{w}}(n))} + \frac{P_{\text{s}}(m)}{2 \max\limits_{n \in V(sg), sg \subseteq g \in D}(P_{\text{s}}(n))}$$

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

For the evaluation of this technique, one can use the measures we have used in the previous chapters. In particular, a suitable evaluation measure is the amount of methods or source code a developer has to investigate when the debugging process is guided by the ranking obtained with $P^{\text{constr}}_{\text{comb}}$.

8.4.2 Weighted-Graph Classification

Subgraph patterns from weighted graphs cannot directly be used for classification. With unweighted graphs, it is common to use binary feature vectors, indicating which subgraph is included in a graph [CYH10a]. Every such vector corresponds to a graph in the graph database. In the following, we explain how we assemble feature vectors including weights to use them for classification. We use one feature in the vector for every edge in every frequent subgraph mined. These features are numerical and stand for the corresponding weight in the original graph. If a graph does not contain a certain subgraph, the corresponding features are null values.

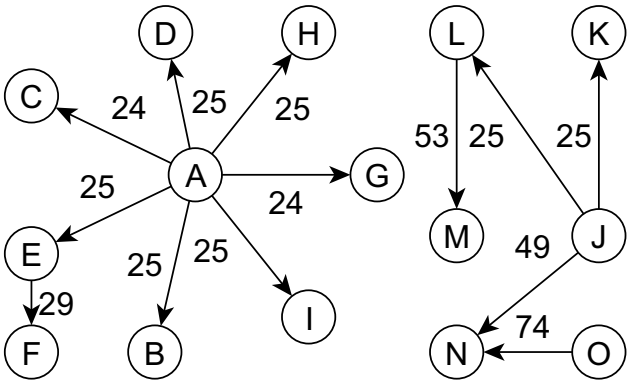


Figure 8.3: Two typical fragments from a small unconnected graph in the logistics dataset.

Example 8.5: We construct a feature vector for the graph in Figure 8.3. Imagine that there are two frequent subgraphs, $A \rightarrow E \rightarrow F$ and $L \rightarrow M$. The vector consists of the values of the edges $A \rightarrow E$, $E \rightarrow F$ and $L \rightarrow M$: $(25, 29, 53)$.

In cases where labels in the subgraph patterns are not unique, the position of an edge in a subgraph describes a certain edge. In case of multiple embeddings of a pattern, we use aggregates of the weights from all embeddings. This encoding allows to analyse every edge weight in the context of every subgraph.

Finally, any classifier featuring numerical attributes and null values can work with the vectors to learn a model or to make predictions. Arbitrary evaluation measures for classification can quantify the predictive quality of the weighted-graph-classification problem. We for our part use the established measures $accuracy$ and AUC (area under the ROC *curve*; see, e.g., [WF05]).

8.4.3 Explorative Mining

Besides automated analysis steps following graph mining, another important application is explorative mining. Here, the results are interpreted directly by humans. One is interested in deriving useful information from a dataset. In our weight-constraint-based scenario, such information is represented as subgraphs with certain edge-weight properties in line with the constraints. For instance, the logistics dataset is well suited for explorative mining. As motivated in Example 8.3, one might be interested in subgraphs featuring edges with high or low variance.

Evaluation in this context is difficult, as it is supposed to provide information for humans. Therefore, it is hard to define a universal measure. In this study, we focus on basic properties of the dataset mined, in particular the size of the subgraphs. This size can be seen as a measure of expressiveness, as larger subgraphs tend to be more significant.

8.5 Experimental Evaluation

We now investigate the characteristics of pruning with several non-anti-monotone constraints, given the real-world analysis problems described before. We do so by comparing different application-specific quality criteria with the speedup in runtime as well as by assessing the completeness of approximate result sets. While other solutions to the real-world problems (without weighted graph mining) might be conceivable as well, studying them to a larger extend is not the concern of this dissertation. (In Section 5.4.3, we have compared call-graph-based software-defect localisation to alternative proposals from the literature.) We first describe the datasets in Section 8.5.1. We then present the experimental settings in Section 8.5.2 and the results in Section 8.5.3.

8.5.1 Datasets

Software-Defect Localisation

We investigate the dataset we have already used in Chapter 5 (see Section 5.4.1), which consists of classified *weighted call graphs*. It consists of 14 defective versions of a Java programme. Every version was executed exactly 100 times with different input data, resulting in roughly the same number of graphs representing *correct* and *failing* executions. The graphs are quite homogeneous; the following numbers describe one of the 14 datasets. The mean number of nodes is 19.6 (standard deviation σ = 1.9), the mean number of edges is 23.8 (σ = 4.6), but the edge weights are quite diverse with a mean value of 227.6 (σ = 434.5).

Logistics

This dataset is the one from [JVB+05]. It is origin-destination data from a logistics company, attributed with different information. The graphs are as follows: Transports fall into two classes with full truckload (*TL*) and less than truckload (*LTL*). The transports from the two classes form two sets of graphs, which we label accordingly. We further arrange transports (edges) with a similar weight of the load in one graph. Next, as the spatial coordinates in the dataset are fine grained, we combine locations close to each other to a single node, e.g., locations from the same town. We use the time needed to get from origin to destination as edge weight. The duration is a crucial parameter in transportation logistics, and there is no obvious connection to the class label. The dataset describes a weighted-graph-classification problem, i.e., predict if a graph contains fully or partly-loaded transports.

Finally, the dataset consists of 51 graphs. The two class labels are evenly distributed, the mean number of nodes is 234.3 (σ = 517.1), and the mean number of edges is 616.1 (σ = 2,418.6). As indicated by the high standard deviations, this is

a very diverse dataset, containing some very large graphs. The large graphs are not problematic for mining algorithms in this case, as most graphs are unconnected, and the fragments are quite small. Besides heterogeneous structural properties, the edge weights with a mean value of 73.2 (σ = 50.9) are quite close to each other. Figure 8.3 is a part of one of the logistics graphs.

8.5.2 Experimental Settings

In our experiments we compare a regular CloseGraph implementation to ours with weight-based constraints. We evaluate the quality of the results with scenario-specific evaluation measures (see Section 8.4) along with the runtime. We use a single core of an AMD Opteron 2218 with 2.6 GHz and 8 GB RAM for all experiments. We mine with a $supp_{\text{min}}$ of 3 in all experiments with the defect-localisation dataset and with a $supp_{\text{min}}$ of 8 in all experiments with the logistics data. We set the $size_{\text{min}}$ to 0 in all experiments, as we are interested in the pure results with the different lower and upper bounds.

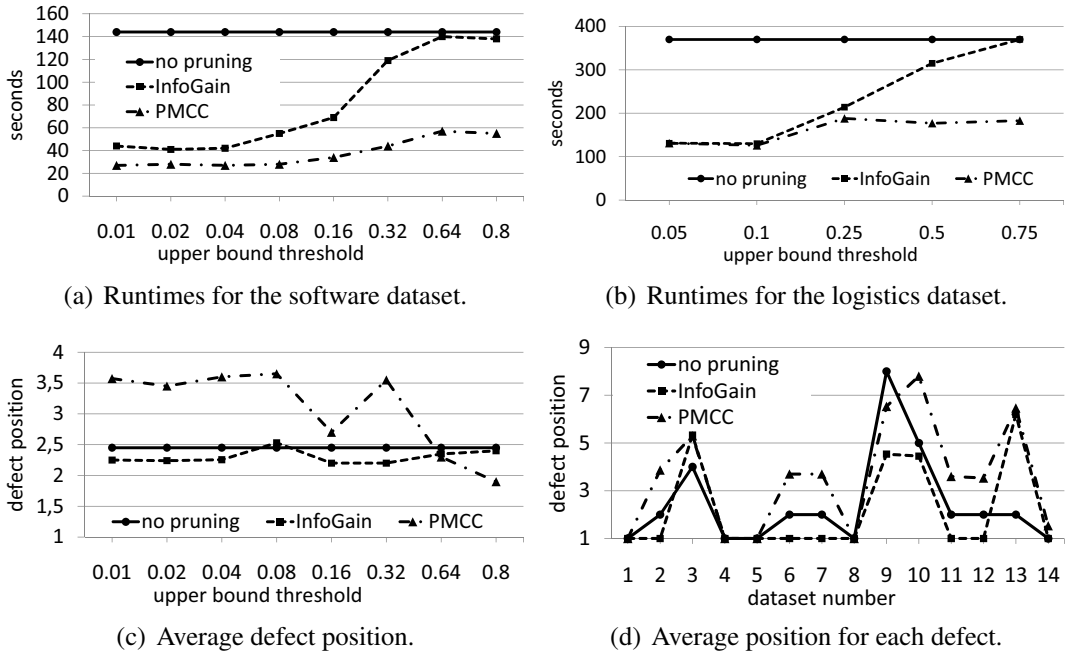
Software-Defect Localisation

In this scenario, we compare our results based on edge-weight-based pruning with a vanilla graph-mining technique. To be fair, we repeat the experiments from Chapter 5 with slight revisions[1] and the same $supp_{\text{min}}$ (3). We use upper-bound constraints on the two class-aware measures.

Weighted-Graph Classification

For classification experiments, we use both datasets. In the software-defect-localisation dataset, we predict the class labels *correct* or *failing*, in the logistics dataset the truck-load labels *TL* and *LTL* (see Section 8.5.1). We mine the graph databases with different upper-bound-constraint thresholds on the two class-aware measures and assemble feature vectors, as described in Section 8.4. We then use them along with the corresponding class labels in a 10-fold-cross-validation setting with standard algorithms. In concrete terms, we use the Weka implementation [HFH+09] of the C4.5 decision-tree classifier [Qui93] and the LIBSVM support-vector machine [CL01] with standard parameters. For scalability reasons, we employ a standard *chi-squared feature-selection* implementation [HFH+09] for dimensionality reduction before applying LIBSVM.

[1] In Chapter 5, a zero in the feature vectors indicates that a certain call does not occur. We now use null values, as this allows for a fair comparison to the new approach.



(a) Runtimes for the software dataset.

(b) Runtimes for the logistics dataset.

(c) Average defect position.

(d) Average position for each defect.

Figure 8.4: Experimental results.

Explorative Mining

For explorative-mining experiments, we investigate different lower-bound-constraint thresholds on $variance$ in the logistics dataset. We compare their quality and runtime with mining runs without constraints.

8.5.3 Experimental Results

Software-Defect Localisation

Figure 8.4(a) displays the runtimes of $InfoGain$ and $PMCC$ with different upper-bound thresholds on all 14 versions of the dataset. The $InfoGain$ constraint is always faster than the execution time without pruning, irrespective of the threshold. For low threshold values (0.01 to 0.04), $InfoGain$ reaches speedups of around 3.5. $PMCC$ in turn always performs better than $InfoGain$ and reaches speedups of up to 5.2. This is natural, as the calculations to be done during mining in order to derive the measures are more complicated for $InfoGain$ (involving logarithms) than for $PMCC$. For high thresholds (0.32 to 0.8) on both measures, the runtime increases significantly. This is caused by less pruning with such thresholds.

Figure 8.4(c) contains the results in defect localisation without pruning and with $InfoGain$ and $PMCC$ pruning with various upper bounds. The figure shows the average position of the defect in the returned ranking of suspicious methods, averaged

for all 14 versions. The $InfoGain$ almost always performs a little bit (a fifth ranking position for the two lowest thresholds) better than the baseline ('no pruning'). As the baseline approach uses $InfoGain$ as well, we explain this effect by the improved structural likelihood computation (P_s, see Section 8.4), which takes advantage of the edge-weight-based pruning. The $PMCC$ curve is worse in most situations. This is as expected, as we know that entropy-based measures perform well in defect localisation (see Section 5.3.1). Figure 8.4(d) contains the defect-localisation results for the 14 different versions. We use the average of the three executions with the best runtime (thresholds 0.01 to 0.04). The figure reveals that the precision of localisations varies for the different defects, and the curve representing the $InfoGain$ pruning is best in all but two cases. Concerning the threshold values, observe that small changes always lead to very small changes in the resulting defect-localisation precision, with mild effects on runtime.

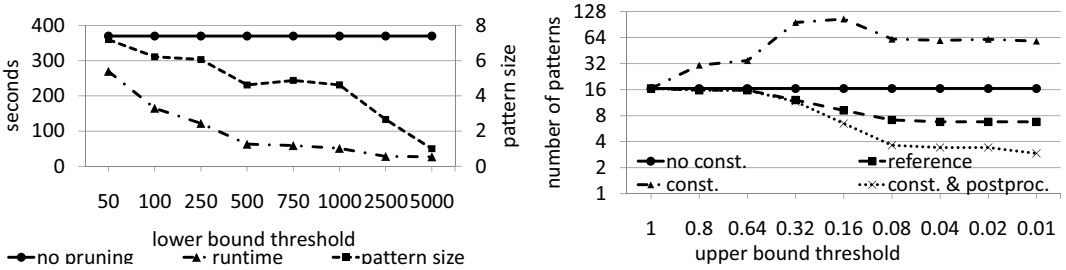
Next to the defect-localisation results, the performance of classifiers learned with the software dataset is very high. The values with $InfoGain$-pruning only vary slightly for the different thresholds on both classifiers, the SVM ($accuracy$: 0.982–0.986; AUC: 0.972–0.979) and the decision tree (*accuracy*: 0.989–0.994; AUC: 0.989–0.994). Although the variance is very low, higher thresholds yield slightly higher values in most cases. This is as expected, as less pruning leads to larger graphs, encapsulating potentially more information. With $PMCC$, the values are very close to those before, and one can make the same observation.

Logistics

Figure 8.4(b) shows the runtimes of both measures with different upper-bound thresholds. With an upper bound of up to 0.10 on $InfoGain$ or $PMCC$, our extension runs about 2.9 times faster than the reference without pruning. For larger upper bounds on $PMCC$, graph mining with our extension still needs only half of the runtime. $InfoGain$ becomes less efficient for larger values, and for a high threshold of 0.75 it needs the same time as the algorithm without edge-weight-based pruning. As before, $PMCC$ performs better than $InfoGain$.

In the experiments, the performance of classifiers does not depend on the upper bound, independently of the threshold. We evaluated the same values as in Figure 8.4(b). For the $InfoGain$ measure, $accuracy$ and AUC of the SVM are 0.902 and 0.898, and they are a little lower with the decision tree: 0.863 and 0.840. For $PMCC$, the results are the same for most upper bounds. Only for the bounds 0.50 and 0.75, where less pruning takes place and more subgraphs are generated, the results are slightly better (decision tree only). Next to classification performance, the runtimes change only slightly when the threshold values change.

These results demonstrate that the edge weights in this dataset are well suited for classification. Further, the degree of edge-weight-based pruning did not influence the results significantly. Therefore, $InfoGain$ and $PMCC$ obviously are appropriate



(a) Runtimes & quality, logistics dataset. (b) Comparison of approximate result sets.

Figure 8.5: Experimental results.

measures. With low upper-bound values on both measures, the runtime can be improved by a factor of about 2.9, while the classifiers have almost the same quality. On the other side, these results also show that the graph structure of this particular dataset is less important to solve the classification problem than the edge weights.

Besides the performance of classification, we also evaluate the $variance$ measure in an explorative mining setting on the logistics dataset. Figure 8.5(a) shows the runtimes with several lower bounds along with the corresponding averaged subgraph-pattern sizes (in edges) in the result set. At the lowest threshold (50), the runtime already decreases to 73% of the runtime without pruning. At the highest value (5,000), the runtime decreases to 7% only, which is a speedup of 13. At the same time, the average subgraph size decreases from 7 to 1. Therefore, values between 250 and 1,000 might be good choices for this dataset (depending on the user requirements), as the runtime is 3 to 7 times faster, while the average subgraph size decreases moderately from 7.4 to 6.1 and 4.6.

Completeness of Approximate Result Sets

We now investigate the completeness of our result sets and look at the defect-localisation experiments with $InfoGain$-constraints another time. Figure 8.5(b) refers to these experiments with the approximate constraint-based CloseGraph algorithm, but displays the sizes of result sets (averaged for all 14 versions). We compare these results with a non-approximate reference, obtained from a non-constrained execution, where we remove all subgraph patterns violating an upper bound afterwards. Our constraint-based mining algorithms save all patterns violating upper bounds before pruning the search (see Section 8.3). For comparison, we apply the same postprocessing as with the reference and present two variants of constraint-based mining: The pure variant (‘const.’) and the postprocessed one (‘const. & postproc.’). Comparing the two postprocessed curves, for thresholds of 0.64 and larger, constraint-based result sets have the same size as the reference and are smaller for thresholds of 0.32 and lower. Preliminary experiments with different $supp_{\min}$ values have revealed

that the difference between the curves decreases ($supp_{\text{min}}$ of around 20 instead of 3) or vanishes ($supp_{\text{min}}$ of 70). The pure result sets (those we used in the experiments before), are always larger than closed mining, even if no constraints are applied. To conclude, our approximate result sets contain less than half of the patterns as the non-approximate reference, for small $supp_{\text{min}}$ and upper bound values. However, the pure result sets obtained from constraint-based mining in a shorter runtime (see Figure 8.4(a)) contain many more interesting subgraph patterns (see curve 'const.'), which is beneficial for the applications.

8.6 Subsumption

In this chapter, we have dealt with mining of weighted graphs. We have integrated non-anti-monotone constraints based on weights into pattern-growth frequent-subgraph-mining algorithms. This has led to improved runtime and approximate results. The goal of our study was to investigate the quality of these results. Besides an assessment of result completeness, we have evaluated its usefulness, i.e., the result quality of higher-level real-world analysis problems based on this data. The evaluation shows that a correlation of weights with the graph structure exists and can be exploited by means of faster analyses. Frequent subgraph mining with weight-based constraints has proven to be useful – at least for the problems investigated.

Besides the hierarchical approach presented in Chapter 6, weight-constraint-based approximate mining is another contribution to scalable software-defect localisation. It allows to perform faster analyses than with our approach presented in Chapter 5. Alternatively, constraint-based approximate mining allows for analyses of larger software projects. At the same time, the results in defect localisation even are a little more precise.

The constraint-based approximate-mining approach presented in this chapter can be employed in a hierarchical-mining scenario (see Chapter 6) and with dataflow-enabled call graphs (DEC graphs, see Chapter 7) without any special challenges. Both proposals (hierarchical mining and DEC graphs) feature graphs with edges annotated with tuples of weights, and our approach can deal with several constraints at the same time. Therefore, constraints can be defined on all tuple elements. However, when measures need to be calculated for an increased number of weights, this will increase runtime. Depending on the number of tuple elements and the nature of the dataset investigated, the post-processing approaches used in Chapters 6 and 7 might be faster than the constraint-based variant. However, there are also possibilities to increase the efficiency of the implementation presented in this chapter. For instance, incremental techniques could be used for the calculation of the measures during mining.

9 Conclusions and Future Research Directions

Defect localisation is an important problem in software engineering. In this dissertation, we have investigated call-graph-mining-based software defect localisation, a relatively recent direction of research in defect localisation. Respective approaches aim at supporting software developers by providing hints where defects might be located, in order to reduce the amount of code a developer has to inspect manually. They rely on the analysis of dynamic call graphs, which are representations from correct and failing programme executions. In this dissertation, we have investigated call-graph-mining-based techniques to draw conclusions on their suitability to derive useful defect localisations. To this end, we have extended the state-of-the-art in call-graph-based defect localisation in various ways. This leads to a broader range of detectable defects, to an increased localisation precision and finally to the conclusion that dynamic call graphs are indeed a suitable data representation for defect localisation.

From a data-mining point of view, mining dynamic call graphs bears a number of challenges. Most importantly, graphs need to be represented adequately, techniques for mining weighted graphs and to derive defect localisations need to be developed, and respective techniques should scale for the analysis of large software projects. In this dissertation, we have dealt with all these challenges, resulting in several call-graph representations, various techniques for defect localisation with weighted call graphs and a technique for graph mining with weight-based constraints. The weight-constraint-based technique in particular is not only limited to the software-engineering application domain, but is a general approach for constraint-based mining of weighted graphs.

In the following, we review the different contributions of this dissertation in more detail (Section 9.1), discuss the lessons learned (Section 9.2) and present some interesting opportunities of future work (Section 9.3).

9.1 Summary of this Dissertation

At the beginning of this dissertation, we have observed that related call-graph-based defect-localisation techniques localise structure-affecting bugs well, but have difficulties in localising frequency-affecting bugs (Chapter 5). This is as the graphs analysed

do not encode the information needed to derive good defect localisations. Therefore, we have proposed graph representations that include such information: call frequencies annotated as numerical edge weights (Chapter 4). In order to use the respective graphs for defect localisation, we have developed a technique that analyses both the graph structure (topology) and the numerical edge weights (Chapter 5). To this end we have developed a combined approach that consists of frequent subgraph mining and feature selection. Besides this, we have identified that the relatively severe total-reduction techniques for call graphs used in the related work lead to a loss of structural information. We have therefore also defined call-graph representations that are a little larger than totally reduced graphs and encode more structural information. In order to be able to localise both kinds of defects, structure-affecting bugs and frequency-affecting bugs, we have proposed combined approaches for defect localisation.

In a first evaluation (Chapter 5) with defects we have artificially seeded into a small programme, we have shown that our call-graph representations and analysis techniques are indeed useful for the localisation of defects. Concretely, we have achieved defect-localisations with a doubled precision compared to related call-graph-based techniques. We have also shown that our approach can detect defects that other approaches cannot detect in principle. Further, we have demonstrated that the numerical information kept with our call-graph representations is important for good results. Besides the comparison to closely related techniques, we have also compared our technique to established techniques from software engineering. The result has been, based on our admittedly relatively small test suite, that our technique performs better than these approaches in most cases.

The next step in this dissertation has dealt with scalability and with a broader evaluation with a real software project and defects from the field (Chapter 6). Based on the observation that our approach proposed so far has difficulties scaling to larger software projects (the approaches from the related work are faced with the same problem), we have proposed a hierarchical procedure: Starting with novel call-graph representations at coarse levels of abstraction, our approach identifies suspicious regions in the call graphs and then zooms-in into these regions. There, it applies the same technique to graphs of a more fine-grained abstraction etc. In an evaluation with real defects from a relatively large open-source project, we have shown that our new call-graph abstractions, as well as our hierarchical mining procedure, are well suited to localise defects and scale for larger software projects. In particular, in our experiments, the source code a software developer has to investigate could be limited to 6% of the whole software project on average. To our knowledge, this is the first study applying call-graph mining to a software project of this size.

A principle problem of all call-graph-based approaches for defect localisation – including our techniques described so far – is that they analyse the graph structure, but are agnostic regarding the dataflow. Therefore, they are unable to detect defects that influence the dataflow only. In order to be able to capture those defects as well,

we have proposed dataflow-enabled call graphs that include abstractions referring to the dataflow (Chapter 7). As well, we have adopted our mining technique and have evaluated our procedure with various defects. The result is that considering the dataflow information allows us to localise defects that cannot be localised otherwise with techniques relying on call graphs. Furthermore, with these enhancements, we are able to increase the defect-localisation precision of a number of further defects.

In most parts of this dissertation, we have relied on combined approaches for mining weighted graphs and ultimately for the localisation of defects. This is, caused by the absence of suitable techniques for mining weighted graphs directly, we have employed frequent subgraph mining in a first analysis step and feature selection in a subsequent postprocessing step. At the end of this dissertation (Chapter 8), we have proposed a unified approach for mining weighted graphs in a single analysis step. Concretely, we have pushed the postprocessing step into the mining algorithm by formulating and processing weight-based constraints. These constraints consider the weights of the graph and allow for pruning the internal search space of frequent subgraph mining. This leads to speed-ups of the mining algorithm (e.g., 3.5 times for the defect-localisation dataset), while obtaining results of a comparable quality. For defect localisation, we have even obtained mining results that are a little more precise. Besides the application to defect localisation, mining with weight-based constraints is a universal approach, and we have as well successfully evaluated it with data from a completely different domain, transportation logistics.

In this dissertation, we have focused on the localisation of defects that occur in single-threaded sequential programmes. However, there are certain classes of defects related to the parallel execution of several threads within the same programme. In order to show that call-graph-based techniques are in principle as well suited to localise certain defects referring to parallel executions, we have performed a first study with a call-graph-representation for multithreaded programmes and an adopted localisation technique (Appendix A). The result is that certain defects can be localised well, but that further investigations will probably lead to more sophisticated call-graph representations that allow for the localisation of a broader range of defects related to parallel executions.

9.2 Lessons Learned

Throughout the research conducted for this dissertation, we have experienced and learned many things. In the following, we highlight the most important lessons we have learned.

Data representations are key for good results. Although it seems to be quite obvious, a data-mining technique can only find patterns, predict behavious or

localise defects if there is respective evidence in the data. In the context of this dissertation, we have investigated several call-graph variations as data representations. Our experience is that particularly weighted call graphs are key to successfully localise a broad range of defects (see Chapters 5 and 7). Weighted call graphs are annotated with numerical information such as call frequencies and dataflow abstractions. Further, finding suitable graph topologies is key for both well results (see Chapter 5) and scalable defect localisation. While frequent subgraph mining does not scale for method-level call graphs from large software projects, it can be used for graphs at coarser levels of granularity or for cut-outs of call graphs (see Chapter 6). To sum up, finding the right data representation and acquiring the data needed to solve the analysis problem is essential – maybe even more important than the actual analysis technique. These observations confirm the more general literature on the data-mining process and on applied data mining [CCK+00, FPSS96, HG08].

Dynamic call graphs are a suitable abstraction for defect localisation. In this dissertation, we have investigated the suitability of dynamic call graphs for defect localisation in software. As shown by the evaluations in the different chapters, call-graph mining does lead to defect localisations that are useful. This is, the amount of code that needs to be investigated manually can be reduced significantly. Furthermore, the comparative evaluation in Chapter 5 has shown that our technique can compete with state-of-the-art approaches that do not rely on call graphs, at least using the test suite considered. More concretely, our technique has outperformed the other approaches in 12 out of 14 cases in our test suite, and we have shown that there are types of defects that can be localised with our technique, but not with the other techniques considered.

Our conclusion that dynamic call graphs are a suitable abstraction for defect localisation holds for the call-graph representations considered in this dissertation. This is, in particular graphs that are annotated with numerical information referring to call frequencies or dataflow values are well suited – along with an analysis technique that makes use of both structural and numerical evidence encoded in the graphs.

Even if graph mining is expensive, it leads to good results. For the analysis of call graphs, we have successfully followed approaches that employ frequent-subgraph-mining techniques. This allows for a very detailed analysis of numerical weights in the context of the different subgraphs and for the derivation of structural scoring measures. However, graph mining is computationally expensive, and it is the bottleneck of our proposed analysis techniques. Even if instrumenting source code leads to moderate runtime overheads, and running feature-selection algorithms needs some time as well, our experience is that graph mining is the most expensive step. However, with our hierarchical procedures (see Chapter 6), the runtime of graph-mining algorithms is in the range of a few minutes. We consider such runtimes to

be acceptable for defect localisation. However, there might be other approaches that analyse the call-graph representations proposed and lead to good localisation results, too. Investigating all such possible approaches was not the aim of this dissertation.

Pruning with non-anti-monotone constraints is useful for applications. Most weight-based constraints are not anti-monotone. Thus, using them for pruning does not guarantee the completeness of the mining results and leads to approximate results. This is as there are no guaranteed laws that relate the graph structure (topology) to the weights attached to the graphs. As topology and weights are nevertheless correlated in many real-world graphs [MAF08], we have proposed weight-based constraints and have integrated them into frequent-subgraph-mining algorithms. Then, we have investigated the effect from approximate (incomplete) result sets on real-world analysis problems. The result is that approximate results have very mild effects on the final result quality, while achieving well speed-ups in runtime.

9.3 Future Research Directions

There are a number of problems with respect to call-graph-mining-based defect localisation that we did not address in order to focus the scope of this dissertation. Some possible extensions of the proposed techniques have already been discussed in the subsumption sections of the individual chapters. We now highlight some more general directions of possible future research. They build on – or arise from – the techniques, results and lessons learned in this dissertation.

Clustering Call Graphs. Caused by the class hierarchy of a grown software project, class and package sizes frequently are very imbalanced. This can lead to a limited applicability of the hierarchical mining approach proposed in Chapter 6 and thus to scalability issues. Furthermore, the manual assignment of software entities to larger units, as typically done by the software developer (e.g., of a class to a package), is often arbitrary. To overcome such problems, it would be helpful to have natural and balanced hierarchies. Such hierarchies could be obtained by means of (weighted) graph clustering [AW10a] on call graphs. More concretely, clustering algorithms could be applied to (sets of) large call graphs. The clusters identified would then be the first hierarchy level, and the same technique could be applied within the individual clusters in order to find more fine-grained clusters. Alternatively, hierarchical clustering methods could be employed. Such an approach has recently been proposed in the context of mining for community structures in (social) networks [HSH+10]. However, it is unclear if clustering techniques can be identified that would result in the desired balanced call-graph hierarchies.

From a general data-mining perspective, clustering call graphs would be interesting as well, because our setting would provide an objective evaluation framework. In

this context, 'objective' means that cluster-analysis results of different quality are expected to yield results with different defect-localisation precision as well. This is in contrast to numerous evaluations where domain experts have decided how good the various clustering results are.

Weighted Subgraph Mining. In this dissertation, we have proposed two principal approaches for mining weighted call graphs:

1. Postprocessing, i.e., we analyse weighted call graphs by means of a two-step approach, consisting of frequent subgraph mining and feature selection (Chapters 5–7).

2. Constraint-based mining, i.e., we let the user specify constraints based on weights and integrate the two steps from the approach mentioned before into a single analysis step (Chapter 8).

Besides the two approaches mentioned, the discretisation-based approaches presented in Section 3.2.1 analyse weights in a preprocessing step. As a drawback of such approaches, we have identified a loss of information, which would possibly lead to worse defect-localisation results. However, it would be interesting to develop such a preprocessing-based approach that relies on discretisation and is tailored for localising defects with weighted call graphs. Even if discretisation leads to a loss of information, this effect could be minimised by employing supervised discretisation techniques (see Section 7.5). Further, such an approach might lead to other positive properties that compensate for this effect, such as decreased runtime.

Besides the techniques based on preprocessing, postprocessing and weight-based constraints discussed so far – and a small number of further studies presented in Section 3.2.1 – weighted subgraph mining has not drawn a lot of attention. In particular, it has never been studied systematically, and most available approaches deal with very specific analysis problems for dedicated applications. As respective algorithms could be used in many domains where weighted graphs are present, and as they promise to achieve good results, it would be rewarding to systematically investigate weighted subgraph mining. It would in particular be desirable to propose techniques that complement the ones proposed in this dissertation – specifically weight-based constraints – and can be applied to a broad field of applications.

Evaluations with Software Repositories from Large Projects. This dissertation contains a number of sections that evaluate the defect-localisation techniques proposed. Some of these evaluations build on relatively small programmes and on defects that have been seeded artificially into them. However, as it is desirable that localisation techniques scale for large programmes, evaluations with larger

software projects substantiate the results of an evaluation. Further, when an evaluation features defects that actually occurred in a real software project, the evaluation is much more credible. In Chapter 6, we have presented an evaluation that features both a real and relatively large project and defects from the field. In order to draw more substantial conclusions about the effectiveness of defect-localisation techniques – not only the ones proposed in this dissertation – it would be desirable if future evaluations would feature even larger software projects, more defects, more kinds of defects and a broader comparison of different defect-localisation techniques. This would require to assemble more test suites that fulfil (parts of) the aspects mentioned before and include enough test cases that lead to both correct and failing executions. Similarly to the iBUGS repository [DZ07, DZ09] we have used in Chapter 6, such test suites could be derived from the test cases and repositories of real (open-source) software projects. Such repositories, in particular bug-tracking systems and revision-control systems, are used in most large software projects and contain lots of interesting data to derive test suites for defect-localisation tools.

Controlled User Experiments with Software Developers. All defect-localisation techniques that have been discussed or have been newly proposed in this dissertation have been evaluated with quantitative evaluation measures. These measures refer directly or indirectly to the amount of code a software developer still has to investigate to find the defect when the respective localisation technique was employed. Thus, they rely on the assumption that all kinds of defects can be identified with the same effort when the same hints are given by a defect-localisation technique. However, this assumption might not hold in reality. This is as developers might have background knowledge that can hardly be assessed, and different defect-localisation results that refer to the same amount of source code to be investigated might be more or less helpful for the developer. Therefore, it would be an interesting experiment to let software developers having the same level of experience find (and fix) defects with the aid of different defect-localisation techniques.

Localising Defects in Multithreaded Programmes. This dissertation has focused on the localisation of defects in single-threaded programmes. However, multithreaded programmes are a challenging field for defect localisation, as respective defects are notoriously hard to localise. In Section 4.3, we have already presented some call-graph representations for multithreaded programmes, and we have conducted a first study on call-graph-based defect localisation in Appendix A. Due to a number of issues related to multithreaded executions, we have employed a relatively simple call-graph representation in this study. However, we believe that alternatives with more sophisticated graph representations that overcome the problems discussed in Section 4.3 are worth being investigated and might substantiate the encouraging results. This is motivated by the experiments with single-threaded programmes in

Chapter 5, where graphs more sophisticated than the total reduction have localised defects more precisely. Graph representations for multithreaded programmes can, for instance, include additional information on thread IDs, as well as information about synchronisation constructs used. Section 4.3 contains some more concrete ideas for possible call-graph extensions. Further, the study in Appendix A does not exploit dataflow-related information. A dataflow extension for call graphs from multithreaded programmes, similar to the one in Chapter 7 for the singe-threaded case, is likely to make race detection more accurate. This is because unsynchronised threads incorrectly alter data and affect the values in the dataflow in typical race situations. All these ideas – as well as further proposals for call-graph representations – are worth being investigated along with respective defect-localisation techniques to a larger extend.

To conclude this dissertation, we have developed different techniques for call-graph-mining-based defect localisation, and we have shown that they are useful. With the mentioned directions for future work in mind, we feel that more software projects and data-mining problems will benefit from this dissertation, and that data-mining-based defect localisation remains an exciting field of research.

Appendix

A Multithreading Defect Localisation

This dissertation focuses on call-graph-based defect localisation in sequential programmes. Apart from that, debugging multithreaded programmes is an important and challenging field of research of its own (see Section 3.1.3). We have introduced call-graph representations for multithreaded programmes in Section 4.3 and present a first study on localising defects with such graphs in this appendix. It is a variation from our approach in Chapter 5. The result is that call-graph-based defect localisation can be used to localise typical defects in multithreaded programmes. However, there are open questions remaining, and we describe some ideas how to extend call-graph-based defect localisation to adequately deal with defects in multithreaded programmes (see Section 4.3 and Chapter 9).

In this appendix, we first present an introductory overview in Section A.1. Section A.2 introduces a simple approach for defect localisation. Section A.3 evaluates the approach. Section A.4 shows a detailed example. Section A.5 compares our technique with other approaches. Section A.6 is a subsumption of this appendix.

A.1 Overview

Debugging multithreaded programmes is an important and challenging problem. Debugging aids for multithreaded programmes that are available today focus on identifying atomicity violations, race conditions or deadlocks (see Section 3.1.3). These tools are specialised on a particular class of parallel programming errors that are due to wrong usage of synchronisation constructs. In this appendix, we investigate further causes, i.e., anomalies in the execution that might produce wrong *parallel* programme behaviour. Let us consider another example besides the one presented in Example 3.1:

Example A.1: Think of a programmer who incorrectly uses a sequential memory allocator in a multithreaded context in a language without automatic garbage collection. In rare cases, different threads could allocate overlapping parts of the memory and perform concurrent accesses, which leads to races. Even though race detectors would be able to intervene and show a report when a race occurs on a particular memory location, many tools offer little insight on the real cause of the problem.

The examples illustrate that there is a need for more general defect-localisation techniques to deal with such situations. This appendix addresses this problem area and investigates the usage of call graphs for defect localisation in multithreaded shared-memory programmes. The approach presented aims to detect a wider range of defects that affect parallel execution rather than just race conditions. The controlled experiments with typical applications presented in this appendix show that mining of call graphs works and that it finds defects in multithreaded programmes.

A.2 Multithreading Defect Localisation

As in the other parts of this dissertation, the overall aim of the defect-localisation procedure presented here is to derive a ranking of potentially defective methods. We present an overview of the defect-localisation procedure in Section A.2.1 and then more details on the localisation technique in Section A.2.2.

A.2.1 Overview

Algorithm A.1 works with a set T of traces obtained from programme executions. Using a test oracle, the algorithm assigns a class ($correct$ or $failing$) to every trace $t \in T$. Then the algorithm reduces every t to obtain a new call graph (using the $R^{\text{mult}}_{\text{total}}$ call-graph reduction, see Section 4.3), which is assigned to a class of either correct or failing executions. Based on these $R^{\text{mult}}_{\text{total}}$ graphs, the last step calculates for every method m its likelihood of being defective. The likelihood is used to rank the order of potentially defective methods.

Algorithm A.1 Overview of call-graph-based defect localisation.

Input: a set of programme traces $t \in T$
Output: a ranking based on each method's likelihood to be defective $P(m)$

1: $G = \varnothing$ // initialise a set of reduced graphs
2: **for all** traces $t \in T$ **do**
3: check if t refers to a correct execution,
and assign a $class \in \{correct, failing\}$ to t
4: $G = G \cup \{reduce(t)\}$
5: **end for**
6: calculate $P(m)$ for all methods m in G

We employ a *test oracle* to decide whether a programme execution is correct or not (Line 3 in Algorithm A.1). Such oracles are specific for the examined programme, and their purpose is to decide if a certain execution yields any observable problems (i.e., a *failure*). An observable problem can be a wrong output or other erroneous behaviour such as a deadlock.

	$a \to b$	$b \to c$	$a \to d$	$\cdots$	Class
g_1	445	445	7	$\cdots$	*failing*
g_2	128	256	0	$\cdots$	*correct*
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$

Table A.1: Example of a feature table.

A.2.2 Calculating Defectiveness Likelihoods

We now describe how to calculate the defect likelihood of a method (Line 6 in Algorithm A.1). In contrast to the methods presented in the earlier parts of this dissertation, we now follow a relatively simple approach: We analyse the edge weights of the $R^{\text{mult}}_{\text{total}}$ call graphs (see Section 4.3) without employing any graph-mining technique. We do so as the programmes investigated in this appendix are rather small and the resulting call graphs do not deviate much between the different executions. Concretely, we create a feature table as follows:

Notation A.1 (Feature tables for defect localisation in multithreaded programmes)
The feature tables have the following structure: The rows stand for all programme executions, represented by their reduced call graphs. For every edge, there is one column. The table cells contain the edge weights, except for the very last column, which contains the $class \in \{correct, failing\}$. *If an edge is not contained in a call graph, the corresponding cells have value 0.*

Example A.2: Table A.1 serves as an example. The first column in Table A.1 corresponds to the edge from method a to method b, the second column to the edge from b to c, and the third column represents an edge from a to d. The last column contains the class $correct$ or $failing$. Graph g_2 does not possess edge $a \to d$; therefore, the respective cell has value 0.

We analyse the edge weights in tables as introduced in Notation A.1. Concretely, we employ the *information-gain-ratio* measure ($GainRatio$, see Definition 2.7) in its Weka implementation [HFH+09] to calculate the strength of discrimination of columns. We then use these values as defect likelihoods for every column in the table, i.e., for method calls. However, we are interested in likelihoods for methods m. As a method can call several other methods, we assign every column to the calling method. We then calculate the method likelihood $P(m)$ as the maximum of the $GainRatio$ values of the columns assigned to method m. We use the maximum because it refers to the most suspicious invocation of a method. Other invocations are less important, as they might not be related to a defect. However, the information which specific invocation within method m is most suspicious (the column with the highest likelihood) can be important for a software developer to find and fix the defect. We therefore report this additional information to the user.

Programme	#M	LOC	#T	Source	Description
AllocationVector (*Test*)	6	133	2	[EU04]	Allocation of memory
GarageManager	30	475	4	[EU04]	Simulation of a garage
Liveness (*BugGen*)	8	120	100	[EU04]	Client-server simulation
MergeSort	11	201	4	[EU04]	Recursive sorting implementation
ThreadTest	12	101	50	[EU04]	CPU benchmark (random divisions)
Tornado	122	632	100	[C+09]	HTTP Server
Weblech	88	802	10	[PH+02]	Website download/ mirror tool

Table A.2: Programmes considered (#M/#T is the number of methods/threads).

A.3 Experimental Evaluation

We now present the experimental results to validate our approach. This section describes the benchmark programmes and their defects (Section A.3.1), the experimental setting (Section A.3.2), the metrics used to interpret the results (Section A.3.3) and the actual results (Section A.3.4). Section A.5 presents comparisons to related techniques.

A.3.1 Benchmark Programmes and Defects

Our benchmark contains a range of different multithreaded programmes. The benchmark covers a broad range of tasks, from basic sorting algorithms and various client-server settings to memory allocators, which are fundamental constructs in many programmes [BMBW00]. As our prototype is implemented in AspectJ, all benchmark programmes are in Java. Most of these programmes have been used in previous studies and were developed in student assignments [EU04]. We slightly modified some of the programmes; for example, in the *GarageManager* application, we replaced different `println()` statements with methods containing code simulating the assignment of work to different tasks. Furthermore, we included two typical client-server applications from the open-source community in our benchmark. These programmes are larger and represent an important class of real applications. Table A.2 lists all programmes along with their size in terms of methods and normalised lines of code (LOC)[1].

The authors of the benchmark programmes have seeded known defects into the programmes. In the two open-source programmes, we manually inserted typical synchronisation defects. All defects are representative for common multithreaded

[1] In this appendix, we use the sum of non-blank and non-comment LOC inside method bodies.

programming errors, e.g., forgotten synchronisation for some variable, and are occasional. The defects cover a broad range of error patterns, such as atomicity violations/race conditions, on one or several correlated variables, deadlocks, but also other kinds of programming errors, e.g., originating from non-parallel constructs, that can influence parallel programme behaviour.

We categorise the defect patterns in the programmes of our evaluation as follows, according to the classification by Farchi et al. [FNU03]:

1. ***AllocationVector***, defect pattern: *"two-stage access"*.
   Two steps of finding and allocating blocks for memory access are not executed atomically, even though the individual steps are synchronised. Thus, two threads might allocate the same memory and cause incorrect interference.

2. ***GarageManager***, defect pattern: *"blocking critical section"*.
   The defect itself is a combination of an incorrectly calculated value due to a forgotten switch case. When this situation occurs, no task is assigned to a particular thread, while a global variable is treated as if work had been assigned. Thus, fewer than the number of threads recorded as active are active. This makes the programme deadlock. We illustrate the *GarageManager* programme in more detail in Section A.4.

3. ***Liveness***, defect pattern: similar to the *"orphaned thread"* pattern.
   When the maximum number of clients is reached, the next requesting client is added to a stack. Although this data structure and a global counter are synchronised, it can happen that the server becomes available while the client is added to the stack. In this case, the client will never resume and will not finish its task.

4. ***MergeSort***, defect pattern: *"two-stage access"*.
   Although methods working on global thread counters are synchronised, the variables themselves are not, which might lead to atomicity violations. In particular, threads ask how many subthreads they are allowed to generate. When two threads apply at the same time, more threads than allowed are generated. This can lead to situations in which parts of the data are not sorted.

5. ***ThreadTest***, defect pattern: *"blocking critical section"*.
   The generation of new threads and checking a global variable for the maximum number of threads currently available is not done correctly in case of exceptions, which occur randomly in *ThreadTest*, due to divisions by zero. This leads to a deadlock when all threads encounter this situation. We classify an execution as failing when at least one thread encounters this problem, due to reduced performance.

6. ***Tornado***, defect pattern: *"no lock"*.
   Synchronisation statements are removed in one method. This leads to a race condition and ultimately, in the context of *Tornado*, to unanswered HTTP requests.

7. ***Weblech***, defect pattern: *"no lock"*.
   Removed synchronisation statements as in *Tornado*, resulting in Web pages that are not downloaded.

For the *Weblech* programme, we have two versions: *Weblech.orig* and *Weblech.inj*. In *Weblech.inj*, we introduced a defect in method `run()` by removing all `synchronized` statements (Listing A.1 shows an excerpt of this method with one such statement), aiming to simulate a typical programming error. During our experiments, we realised that the original non-injected version (*Weblech.orig*) led to failures in very rare cases as well. The failure occurred in only 5 out of 5,000 executions; we used a sample of the correct executions in the experiments. Thus, *Weblech.inj* contains the original defect besides the injected defects. With our tool, we were able to localise the real defect by investigating two methods only. The result is that two global unsynchronised variables (`downloadsInProgress` and `running`) are modified in `run()`, occasionally causing race conditions. To fix the defect in order to produce a defect-free reference, we added the `volatile` keyword to the variable declaration in the class header.

```
1 while ((queueSize() > 0 || downloadsInProgress > 0)
2   && quit == false) {
3   // ...
4   synchronized (queue) {
5     nextURL = queue.getNextInQueue();
6     downloadsInProgress++;
7   }
8   // ...
9 }
10 running--;
```

Listing A.1: Method `void weblech.spider.run()` (excerpt).

A.3.2 Experimental Setting

Number of Executions. Our defect-localisation technique requires that we execute every programme several times and that we ensure that there are sufficiently many examples for correct and failing executions. This is necessary since we focus on occasional bugs (see Chapter 2), i.e., failures whose occurrence depends on input

data, random components or non-deterministic thread interleavings. Furthermore, we tried to achieve stable results, i.e., analysing more executions would not lead to significant changes. We used this criterion to determine the number of executions required, in addition to obtaining enough correct and failing cases. Table A.3 lists the number of correct and failing executions for each benchmark programme.

Varying Execution Traces. In order to obtain different execution traces from the same programme, we rely on the original test cases that are provided in the benchmark suite. *MergeSort*, for instance, comes with a generator creating random arrays as input data. Some programmes have an internal random component as part of the programme logic, i.e., they automatically lead to varying executions. *GarageManager*, for instance, simulates varying processes in a garage. Other programmes produce different executions due to different thread interleavings that can lead to observable failures occasionally. For the two open-source programmes, we constructed typical test cases ourselves; for the *Tornado* Web server, we start a number of scripts simultaneously downloading files from the server. For *Weblech*, we download a number of files from a (defect-free) Web server.

Test Oracles. We use individual test oracles that come with every benchmark programme. For the two open-source programmes, we compose test oracles that automatically compare the actual output of a programme to the expected one. For example, we compare the files downloaded with *Weblech* to the original ones.

Testing Environment. We run all experiments on a standard HP workstation with an AMD Athlon 64 X2 dual-core processor 4800+. We employed a standard Sun Java 6 virtual machine on Microsoft Windows XP.

A.3.3 Accuracy Measures for Defect-Localisation Results

As in the earlier parts of this dissertation, the locations of the actual defects are known, so the report of a method containing a defect can be directly compared to the known location. If there is more than one location which can be altered to fix a defect, we refer to the position of the first of such methods in the ranking. For cases as in *Weblech.orig* where the defect can be fixed outside a method body (e.g., in the class header), one can still identify methods that can be altered to fix the erroneous behaviour.

In order to evaluate the accuracy of the results, we report the position of the defective method in an ordered result list, as before. Similar to the approach investigated in Chapters 6 and 7, we now always use a second static ranking criterion: We sort the methods with the same likelihood decreasingly by their size in LOC. We provide the percentage of LOC to review additionally to the ranking position. This is calculated

Program	Executions		Defect Localisation	
	#correct	#failing	Ranking Pos.	%LOC to Review
AllocationVector	383	117	1	17.3%
GarageManager	74	26	1	14.2%
Liveness	149	53	1	44.2%
MergeSort	668	332	1	25.9%
ThreadTest	207	193	1	18.8%
Tornado	362	8	14	23.3%
Weblech.orig	494	5	2	23.3%
Weblech.inj	985	15	5	21.8%

Table A.3: Defect-localisation results.

as the ratio of methods that has to be considered in the programme, i.e., the sum of LOC of all methods having a ranking position smaller than or equal to the position reported in the table, divided by the total LOC (see Table A.2).

A.3.4 Results

We present our results in Table A.3. The numbers are encouraging: In all five benchmark programmes, the defective method is ranked first. The ranking position is lower only in the two large programmes. However, taking the size of these programmes into account, the quality of defect localisation is within the same range (see column "LOC to Review").

Overall, the average ranking position for methods containing the defects is 3.3. Nevertheless, as Table A.2 shows, a developer only has to review just 7.1% of all methods to find the defects or 23.6% of the normalised source code, which is low. In other words, a developer has to consider less than a quarter of the source code of our programmes in order to find a defect in the worst case. This reduces the percentage of methods (code) to review by a factor of seven (code: more than by half) when compared to an average expected amount of 50% of methods (code) to review. Note that these all values are obtained without any possible prior knowledge of the developer, which might further narrow down the code to be inspected. Furthermore, they are maximum values, for two reasons: (1) Usually not all lines of a method need to be inspected, in particular due to information reported additionally which call within a method is most suspicious. (2) The methods ranked highest frequently are good hints for the defect, even if the defective method itself is ranked lower. This is as we know from our experience that non-defective methods that are ranked high often are in the vicinity of the defective method, e.g., they might be invoked from the defective method.

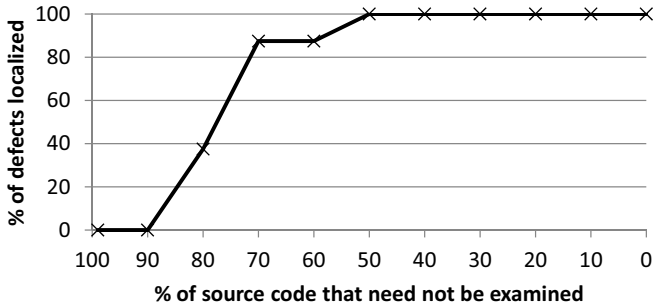


Figure A.1: The percentage of defects localised when not examining a certain percentage of source code.

Figure A.1 provides an illustration of the percentage of localised defects versus the percentage of source code that does not need to be examined. In our case, it shows that we can skip the inspection of 50% of the code and still find 100% of the defects. If we skip inspecting 70%, we would still find more than 80% of the defects. This is a significant gain in programmer productivity.

A.4 A Detailed Example

We now illustrate a typical defect and the process of its localisation with our approach using excerpts form the *GarageManager* programme [EU04]:

The Defect. In our example, the calculation of the `taskNumber` variable can produce a negative value, which is read in method `GoToWork()` (see Listing A.2) to calculate its modulo-8 value, which is then fed into a `switch-case` block. This block, however, expects values between 0 and 7. Negative values can result when Java calculates the modulo operation on a negative number. There are two alternative positions where a developer can modify the code to fix the bug: (1) The `switch-case` block, by adding negative cases or a default case; (2) The parts of the source code where `taskNumber` is calculated (method `SetTaskToWorker()`).

From the Defect to an Infection. We now look at the call graph from a failing execution in more detail, shown in Figure A.2. The call of `run()` generates five threads: Four "worker" threads calling methods `WaitForManager()`, `GoToWork()` and `PrintCard()` and one "manager" thread calling the remaining methods. In `WorkingOn()` (a defective method), the programme state becomes infected: Three threads evaluate their switch statement to 0, 1 and 7, but the fourth thread has a negative value, thus causing the thread not to call any further methods.

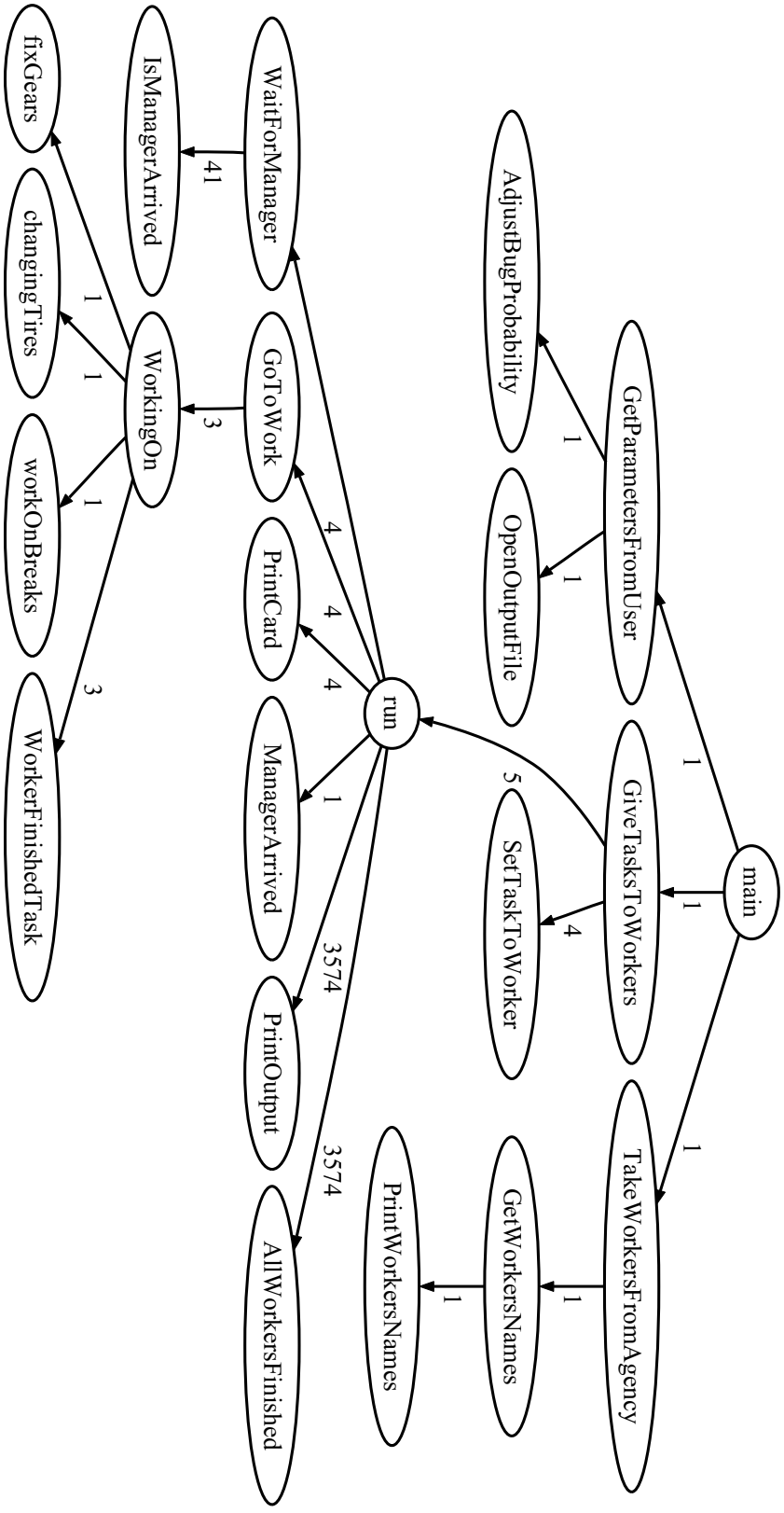


Figure A.2: Call graph from a failing *GarageManager* execution.

```
switch (taskNumber % 8) {
  case 0:
    WorkingOn("Cleaning", 1000);
    break;
  // similar for cases 1 to 5...
  case 6:
    WorkingOn("Working␣on␣breaks", 2200);
    break;
  case 7:
    WorkingOn("Fixing␣engines", 2400);
    break;
}
```

Listing A.2: Method `void GoToWork()` (excerpt).

From an Infection to a Failure. The aforementioned infection causes the fourth thread not to call `WorkerFinishedTask()`. This method decreases a variable of the global `status` object. This object is queried by `AllWorkersFinished()` in method `run()` (see Listing A.3). `AllWorkersFinished()` will never be `true`, as `status` will always indicate that only three out of four "worker" threads have finished their tasks. This causes an infinite loop in `run()`. We manually stopped the loop after 3,574 iterations. In other words, the infection has caused a deadlock, an observable programme behaviour, which we consider a failure.

```
synchronized (status) {
  System.out.println("Manager␣arrived␣!");
  status.ManagerArrived();
}
boolean tasksNotFinished = true, printedOutput = false;
while (tasksNotFinished) {
  printedOutput = PrintOutput(printedOutput);
  synchronized (status) {
    if (status.AllWorkersFinished())
      tasksNotFinished = false;
    else
      yield();
  }
}
```

Listing A.3: Method `void run()` (excerpt).

Localising the Defect. In our experiments, our approach has found the three methods `GoToWork()`, `WorkingOn()` and `run()` (ordered by increasing ranking position) to be most likely defective. Thus, the defect was pinpointed directly. The high likelihood for `WorkingOn()` is due to a follow-up infection, as it is always called from `GoToWork()`. The `run()` method has a high likelihood as well, caused by the huge number of method calls in the infinite loop, compared to correct executions. Both methods are inherently connected to the defect.

A.5 Result Comparisons with Related Work

We now compare our approach with two applicable techniques from the related work.

Our experiments with the IBM MulticoreSDK [QDLT09] applied to all programme versions from our evaluation (see Section A.3) reveal that it is not able to find any of the defects. From the eight versions, the MulticoreSDK incorrectly classified seven versions as defect-free, while producing a false-positive warning for the eighth version.

We applied FindBugs [AHM+08] to all programmes in our benchmark. We observed that FindBugs did not directly report any of the defects. At the same time, FindBugs produces false-positive warnings: On average, there are 5.8 warnings per programme that on average affect 4.5 different methods. The warnings refer to the correct method names in just four out of eight programmes. Further, the warnings are not prioritised, so a developer would have to inspect the entire code of all methods with warnings. In each of the four programmes, inspection amounts to 47.5%, 36.8%, 29.2% and 29.2% of the source code, respectively. If FindBugs was improved by a method ranking technique, such as inspecting larger methods first (as in this appendix), then developers could save time finding the respective defects and reduce the amount of reviewed code to 14.2%, 25.9%, 25.4% and 25.4%, respectively. In contrast, inspecting up to 25.9% of the source code with our technique finds seven out of the eight defects (see the last column in Table A.3). These results are better than FindBugs. Compared to our approach, FindBugs does not offer the developer any hint on finding the remaining four defects, as they are not reported at all.

A.6 Subsumption

In this appendix, we have presented and evaluated a variation from our frequency-based defect-localisation approach (see Section 5.3.1) for multithreaded programmes. Although the call-graph variations investigated (see Section 4.3) are rather simple, we were able to achieve well results in localising defects in multithreaded programmes. The evaluation shows that mining call graphs is an effective approach to detect a wide range of errors that affect parallel programme behaviour. These errors include race

conditions, deadlocks and errors originating from the wrong usage of non-parallel language constructs. This is in contrast to existing multithreading debugging aids that concentrate on detecting specific situations such as race conditions. Notably, the approach presented was able to localise a previously unknown (and undocumented) defect in an open-source tool. However, certain defects in a multithreaded environment might not be captured by the approach presented in this appendix. Extensions to the call-graph representations and to the mining technique might help to broaden the range of detectable defects (see Section 4.3 and Chapter 9).

Bibliography

[AAK+02] Tatsuya Asai, Kenji Abe, Shinji Kawasoe, Hiroki Arimura, Hiroshi Sakamoto and Setsuo Arikawa. Efficient Substructure Discovery from Large Semi-structured Data. *Proceedings of the 2nd SIAM International Conference on Data Mining (SDM)*. 2002.

[AAUN03] Tatsuya Asai, Hiroki Arimura, Takeaki Uno and Shin-Ichi Nakano. Discovering Frequent Substructures in Large Unordered Trees. *Proceedings of the 6th International Conference on Discovery Science (DS)*. 2003.

[AHM+08] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix and William Pugh. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5):22–29, 2008.

[All70] Frances E. Allen. Control Flow Analysis. *ACM SIGPLAN Notices*, 5(7):1–19, 1970.

[All74] Frances E. Allen. Interprocedural Data Flow Analysis. *Proceedings of the IFIP Congress*. 1974.

[AMS+96] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen and A. Inkeri Verkamo. Fast Discovery of Association Rules. In Fayyad et al. [FPSSU96], chap. 12, pp. 307–328.

[AP10] Nathaniel Ayewah and William Pugh. The Google FindBugs Fixit. *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA)*. 2010.

[AS95] Rakesh Agrawal and Ramakrishnan Srikant. Mining Sequential Patterns. *Proceedings of the 11th International Conference on Data Engineering (ICDE)*. 1995.

[AW10a] Charu C. Aggarwal and Haixun Wang. A Survey of Clustering Algorithms for Graph Data. In *Managing and Mining Graph Data* [AW10c], chap. 9, pp. 275–301.

[AW10b] Charu C. Aggarwal and Haixun Wang. Graph Data Management and Mining: A Survey of Algorithms and Applications. In *Managing and Mining Graph Data* [AW10c], chap. 2, pp. 13–68.

[AW10c] Charu C. Aggarwal and Haixun Wang, eds. *Managing and Mining Graph Data*, vol. 40 of *Advances in Database Systems*. Springer, 2010.

[AZGvG09] Rui Abreu, Peter Zoeteweij, Rob Golsteijn and Arjan J.C. van Gemund. A Practical Evaluation of Spectrum-Based Fault Localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[BB02] Christian Borgelt and Michael R. Berthold. Mining Molecular Fragments: Finding Relevant Substructures of Molecules. *Proceedings of the 2nd IEEE International Conference on Data Mining (ICDM)*. 2002.

[BBHK10] Michael R. Berthold, Christian Borgelt, Frank Höppner and Frank Klawonn. *Guide to Intelligent Data Analysis: How to Intelligently Make Sense of Real Data*, vol. 42 of *Texts in Computer Science*. Springer, 2010.

[Bei90] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold Co., 2nd edn., 1990.

[BGRS99] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan and Uri Shaft. When Is "Nearest Neighbor" Meaningful? *proceedings of the 7th International Conference on Database Theory (ICDT)*. 1999.

[Bin07] David Binkley. Source Code Analysis: A Road Map. *Proceedings of the 29th International Conference on Software Engineering (ICSE)*. 2007.

[BK98] Christian Borgelt and Rudolf Kruse. Attributauswahlmaße für die Induktion von Entscheidungsbäumen: Ein Überblick. In Gholamreza Nakhaeizadeh, ed., *Data Mining: Theoretische Aspekte und Anwendungen*, Beiträge zur Wirtschaftsinformatik, pp. 77–98. Physica, 1998.

[BMB05] Christian Borgelt, Thorsten Meinl and Michael R. Berthold. MoSS: A Program for Molecular Substructure Mining. *Proceedings of the Workshop on Open Source Data Mining Software (OSDM)*. 2005.

[BMBW00] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. *SIGPLAN Notices*, 35(11):117–128, 2000.

[BP98] Sergey Brin and Lawrence Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[C+09] Neil Conway et al. Tornado HTTP Server, 2009. Software available at `http://tornado.sourceforge.net/`.

[CCK+00] Pete Chapman, Julian Clinton, Randy Kerber, Thomas Khabaza, Thomas Reinartz, Colin Shearer and Rüdiger Wirth. *CRISP-DM 1.0 – Step-by-Step Data Mining Guide*. The CRISP-DM Consortium, 2000.

[CH06] Diane J. Cook and Lawrence B. Holder, eds. *Mining Graph Data*. John Wiley & Sons, 2006.

[CHS+08] Vineet Chaoji, Mohammad Al Hasan, Saeed Salem, Jeremy Besson and Mohammed J. Zaki. ORIGAMI: A Novel and Effective Approach for Mining Representative Orthogonal Graph Patterns. *Statistical Analysis and Data Mining*, 1(2):67–84, 2008.

[CL01] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A Library for Support Vector Machines. Department of Computer Science and and Information Engineering, National Taiwan University, Taipei, Taiwan, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`.

[CLZ+09] Hong Cheng, David Lo, Yang Zhou, Xiaoyin Wang and Xifeng Yan. Identifying Bug Signatures Using Discriminative Graph Mining. *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)*. 2009.

[CMNK05] Yun Chi, Richard R. Muntz, Siegfried Nijssen and Joost N. Kok. Frequent Subtree Mining – An Overview. *Fundamenta Informaticae*, 66(1–2):161–198, 2005.

[CPY08] Ray-Yaung Chang, Andy Podgurski and Jiong Yang. Discovering Neglected Conditions in Software by Mining Dependence Graphs. *IEEE Transactions on Software Engeneering*, 34(5):579–596, 2008.

[CS98] Jong-Deok Choi and Harini Srinivasan. Deterministic Replay of Java Multithreaded Applications. *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*. 1998.

[CWC95] John Y. Ching, Andrew K. C. Wong and Keith C. C. Chan. Class-Dependent Discretization for Inductive Learning from Continuous and Mixed-Mode Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(7):641–651, 1995.

[CYH10a] Hong Cheng, Xifeng Yan and Jiawei Han. Discriminative Frequent Pattern-Based Graph Classification. In Yu et al. [YHF10], chap. 9, pp. 237–262.

[CYH10b] Hong Cheng, Xifeng Yan and Jiawei Han. Mining Graph Patterns. In Aggarwal and Wang [AW10c], chap. 9, pp. 365–392.

[CYM03] Yun Chi, Yirong Yang and Richard R. Muntz. Indexing and Mining Free Trees. *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM)*. 2003.

[CYM04] Yun Chi, Yirong Yang and Richard R. Muntz. HybridTreeMiner: An Efficient Algorithm for Mining Frequent Rooted Trees and Free Trees Using Canonical Forms. *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM)*. 2004.

[CYZ+09] Chen Chen, Xifeng Yan, Feida Zhu, Jiawei Han and Philip S. Yu. Graph OLAP: A Multi-Dimensional Framework for Graph Data Analysis. *Knowledge and Information Systems*, 21(1):41–63, 2009.

[CYZZ10] Longbing Cao, Philip S. Yu, Chengqi Zhang and Yanchang Zhao. *Domain Driven Data Mining*. Springer, 2010.

[CZ02] Jong-Deok Choi and Andreas Zeller. Isolating Failure Inducing Thread Schedules. *Proceedings of the 11th International Symposium on Software Testing and Analysis (ISSTA)*. 2002.

[CZ05] Holger Cleve and Andreas Zeller. Locating Causes of Program Failures. *Proceedings of the 27th International Conference on Software Engineering (ICSE)*. 2005.

[Dar04] Ian F. Darwin. *Java Cookbook*. O'Reilly & Associates, 2004.

[DDG+08] Thomas G. Dietterich, Pedro Domingos, Lise Getoor, Stephen Muggleton and Prasad Tadepalli. Structured Machine Learning: The Next Ten Years. *Machine Learning*, 73(1):3–23, 2008.

[DDZS09] Laura Dietz, Valentin Dallmeier, Andreas Zeller and Tobias Scheffer. Localizing Bugs in Program Executions with Graphical Models. *Proceedings of the 23rd Conference on Neural Information Processing Systems (NIPS)*. 2009.

[DFLS06] Giuseppe Di Fatta, Stefan Leue and Evghenia Stegantova. Discriminative Pattern Mining in Software Fault Detection. *Proceedings of the*

3rd International Workshop on Software Quality Assurance (SOQUA). 2006.

[Die06] Reinhard Diestel. *Graph Theory*. Springer, 3rd edn., 2006.

[DKS95] James Dougherty, Ron Kohavi and Mehran Sahami. Supervised and Unsupervised Discretization of Continuous Features. *Proceedings of the 12th International Conference on Machine Learning (ICML)*. 1995.

[DLZ05] Valentin Dallmeier, Christian Lindig and Andreas Zeller. Lightweight Defect Localization for Java. *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP)*. 2005.

[DP07] Guozhu Dong and Jian Pei. *Sequence Data Mining*, vol. 33 of *Advances in Database Systems*. Springer, 2007.

[DZ07] Valentin Dallmeier and Thomas Zimmermann. Extraction of Bug Localization Benchmarks from History. *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2007.

[DZ09] Valentin Dallmeier and Thomas Zimmermann. iBUGS – Bug Repositories Extracted from Project History. Department of Computer Science, Saarland University, Saarbrücken, Germany, 2009. Repository available at `http://www.st.cs.uni-saarland.de/ibugs/`.

[EA03] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*. 2003.

[EB09] Frank Eichinger and Klemens Böhm. Towards Scalability of Graph-Mining Based Bug Localisation. *Proceedings of the 7th International Workshop on Mining and Learning with Graphs (MLG)*. 2009.

[EB10] Frank Eichinger and Klemens Böhm. Software-Bug Localization with Graph Mining. In Aggarwal and Wang [AW10c], chap. 17, pp. 515–546. © Springer Science+Business Media, LLC 2010. The original publication is available at `http://www.springerlink.com/`.

[EBH08a] Frank Eichinger, Klemens Böhm and Matthias Huber. Improved Software Fault Detection with Graph Mining. *Proceedings of the 6th International Workshop on Mining and Learning with Graphs (MLG)*. 2008.

[EBH08b] Frank Eichinger, Klemens Böhm and Matthias Huber. Mining Edge-Weighted Call Graphs to Localise Software Bugs. *Proceedings of the*

8th European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD). 2008. © Springer-Verlag Berlin Heidelberg 2008. The original publication is available at `http://www.springerlink.com/`.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[ECJ+10] Ashraf Elsayed, Frans Coenen, Chuntao Jiang, Marta García-Fiñana and Vanessa Sluming. Corpus Callosum MR Image Classification. *Knowledge-Based Systems*, 23(4):330–336, 2010.

[EHB10a] Frank Eichinger, Matthias Huber and Klemens Böhm. On the Usefulness of Weight-Based Constraints in Frequent Subgraph Mining. *Proceedings of the 30th BCS SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence (AI)*. 2010.

[EHB10b] Frank Eichinger, Matthias Huber and Klemens Böhm. On the Usefulness of Weight-Based Constraints in Frequent Subgraph Mining. Karlsruhe Reports in Informatics 2010,10, Department of Informatics, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, 2010.

[EKKB10] Frank Eichinger, Klaus Krogmann, Roland Klug and Klemens Böhm. Software-Defect Localisation by Mining Dataflow-Enabled Call Graphs. *Proceedings of the 10th European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*. 2010. © Springer-Verlag Berlin Heidelberg 2010. The original publication is available at `http://www.springerlink.com/`.

[EOB11] Frank Eichinger, Christopher Oßner and Klemens Böhm. Scalable Software-Defect Localisation by Hierarchical Mining of Dynamic Call Graphs. *Proceedings of the 11th SIAM International Conference on Data Mining (SDM)*. 2011.

[EPGB10] Frank Eichinger, Victor Pankratius, Philipp W. L. Große and Klemens Böhm. Localizing Defects in Multithreaded Programs by Mining Dynamic Call Graphs. *Proceedings of the 5th Testing: Academic and Industrial Conference – Practice and Research Techniques (TAIC PART)*. 2010. © Springer-Verlag Berlin Heidelberg 2010. The original publication is available at `http://www.springerlink.com/`.

[ER97] Tapio Elomaa and Juho Rousu. Efficient Multisplitting on Numerical Data. *Proceedings of the 1st European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD)*. 1997.

[EU04] Yaniv Eytani and Shmuel Ur. Compiling a Benchmark of Documented Multi-Threaded Bugs. *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*. 2004.

[FA10] Andrew Frank and Arthur Asuncion. UCI Machine Learning Repository. School of Information and Computer Sciences, University of California, Irvine, USA, 2010. Repository available at `http://archive.ics.uci.edu/ml/`.

[FI93] Usama M. Fayyad and Keki B. Irani. Multi-Interval Discretization of Continuousvalued Attributes for Classification Learning. *Proceedings of the 13th International Joint Conference on Articial Intelligence*. 1993.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe and Raymie Stata. Extended Static Checking for Java. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2002.

[FNU03] Eitan Farchi, Yarden Nir and Shmuel Ur. Concurrent Bug Patterns and How to Test Them. *Proceedings of the 1st Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD)*. 2003.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Language Systems*, 9(3):319–349, 1987.

[FPSS96] Usama M. Fayyad, Gregory Piatetsky-Shapiro and Padhraic Smyth. From Data Mining to Knowledge Discovery: An Overview. In Fayyad et al. [FPSSU96], chap. 1, pp. 1–34.

[FPSSU96] Usama M. Fayyad, G. Gregory Piatetsky-Shapiro, Padhraic Smyth and Ramasmy Uthurusamy, eds. *Advances in Knowledge Discovery and Data Mining*. AAAI Press/MIT Press, 1996.

[Gai86] Jason Gait. A Probe Effect in Concurrent Programs. *Software: Practice and Experience*, 16(3):225–233, 1986.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[GKM82] Susan L. Graham, Peter B. Kessler and Marshall K. Mckusick. gprof: A Call Graph Execution Profiler. *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*. 1982.

[GRS99] Minos N. Garofalakis, Rajeev Rastogi and Kyuseok Shim. SPIRIT: Sequential Pattern Mining with Regular Expression Constraints. *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*. 1999.

[GWBV02] Isabelle Guyon, Jason Weston, Stephen Barnhill and Vladimir Vapnik. Gene Selection for Cancer Classification using Support Vector Machines. *Machine Learning*, 46(1–3):389–422, 2002.

[HCXY07] Jiawei Han, Hong Cheng, Dong Xin and Xifeng Yan. Frequent Pattern Mining: Current Status and Future Directions. *Data Mining and Knowledge Discovery*, 15(1):55–86, 2007.

[HFGO94] Monica Hutchins, Herb Foster, Tarak Goradia and Thomas Ostrand. Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria. *Proceedings of the 16th International Conference on Software Engineering (ICSE)*. 1994.

[HFH+09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann and Ian H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[HG08] Jiawei Han and Jing Gao. Research Challenges for Data Mining in Science and Engineering. In Hillol Kargupta, Jiawei Han, Philip S. Yu, Rajeev Motwani and Vipin Kumar, eds., *Next Generation of Data Mining*, Data Mining and Knowledge Discovery, chap. 1, pp. 3–27. Chapman & Hall/CRC, 2008.

[HJO08] Hwa-You Hsu, James A. Jones and Alessandro Orso. RAPID: Identifying Bug Signatures to Support Debugging Activities. *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2008.

[HK00] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 2nd edn., 2000.

[HMS01] David Hand, Heikki Mannila and Padhraic Smyth. *Principles of Data Mining*. Adaptive Computation and Machine Learning. MIT Press, 2001.

[How78] William E. Howden. A Survey of Dynamic Analysis Methods. In Edward Miller and William E. Howden, eds., *Software Testing and Validation Techniques*, pp. 184–206. IEEE Computer Society Press, 1978.

[HPY00] Jiawei Han, Jian Pei and Yiwen Yin. Mining Frequent Patterns without Candidate Generation. *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2000.

[HS95] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1995.

[HSH+10] Jianbin Huang, Heli Sun, Jiawei Han, Hongbo Deng, Yizhou Sun and Yaguang Liu. SHRINK: A Structural Clustering Algorithm for Detecting Hierarchical Communities in Networks. *Proceedings of the 19th International Conference on Information and Knowledge Management (CIKM)*. 2010.

[HWP03] Jun Huan, Wei Wang and Jan Prins. Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism. *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM)*. 2003.

[IWM00] Akihiro Inokuchi, Takashi Washio and Hiroshi Motoda. An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data. *Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery (PKDD)*. 2000.

[JCSZ08] Chuntao Jiang, Frans Coenen, Robert Sanderson and Michele Zito. Graph-Based Image Classification by Weighting Scheme. *Proceedings of the 28th BCS SGAI International Conference on Artificial Intelligence (AI)*. 2008.

[JCSZ10] Chuntao Jiang, Frans Coenen, Robert Sanderson and Michele Zito. Text Classification using Graph Mining-Based Feature Extraction. *Knowledge-Based Systems*, 23(4):302–308, 2010.

[JCZ10] Chuntao Jiang, Frans Coenen and Michele Zito. Frequent Sub-graph Mining on Edge Weighted Graphs. *Proceedings of the 12th International Conference on Data Warehousing and Knowledge Discovery (DAWAK)*. 2010.

[Jen09] Finn V. Jensen. Bayesian Networks. *Interdisciplinary Reviews: Computational Statistics*, 1(3):307–315, 2009.

[JH05] James A. Jones and Mary Jean Harrold. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. *Proceedings of the*

20th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2005.

[JHS02] James A. Jones, Mary Jean Harrold and John Stasko. Visualization of Test Information to Assist Fault Localization. *Proceedings of the 24th International Conference on Software Engineering (ICSE)*. 2002.

[Joh00] Philip M. Johnson. A Comparative Review of LOCC and CodeCount. Tech. Rep. CSDL-00-10, Department of Information and Computer Sciences, University of Hawaii, Honolulu, USA, 2000.

[Jon08] Capers Jones. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill, 2008.

[Jor99] Michael I. Jordan, ed. *Learning in Graphical Models*. MIT Press, 1999.

[JVB+05] Wei Jiang, Jaideep Vaidya, Zahir Balaporia, Chris Clifton and Brett Banich. Knowledge Discovery from Transportation Network Data. *Proceedings of the 21st International Conference on Data Engineering (ICDE)*. 2005.

[KC04] Lukasz A. Kurgan and Krzysztof J. Cios. CAIM Discretization Algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 16(2):145–153, 2004.

[Ker92] Randy Kerber. ChiMerge: Discretization of Numeric Attributes. *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI)*. 1992.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold. An Overview of AspectJ. *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*. 2001.

[KK01] Michihiro Kuramochi and George Karypis. Frequent Subgraph Discovery. *Proceedings of the 1st IEEE International Conference on Data Mining (ICDM)*. 2001.

[KKR10] Klaus Krogmann, Michael Kuperberg and Ralf Reussner. Using Genetic Search for Reverse Engineering of Parametric Behaviour Models for Performance Prediction. *IEEE Transactions on Software Engineering*, 36(6):865–877, 2010.

[KL88] Bogdan Korel and Janusz Laski. Dynamic Program Slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin. Aspect-Oriented Programming. *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*. 1997.

[Kon94] Igor Kononenko. Estimating Attributes: Analysis and Extensions of RELIEF. *Proceedings of the 7th European Conference on Machine Learning (ECML)*. 1994.

[KPB06] Patrick Knab, Martin Pinzger and Abraham Bernstein. Predicting Defect Densities in Source Code Files with Decision Tree Learners. *Proceedings of the International Workshop on Mining Software Repositories (MSR) at ICSE*. 2006.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[LAZJ03] Ben Liblit, Alex Aiken, Alice X. Zheng and Michael I. Jordan. Bug Isolation via Remote Program Sampling. *ACM SIGPLAN Notices*, 38(5):141–154, 2003.

[LCH+09] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo and Chengnian Sun. Classification of Software Behaviors for Failure Detection: A Discriminative Pattern Mining Approach. *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 2009.

[LFY+06] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han and Samuel P. Midkiff. Statistical Debugging: A Hypothesis Testing-Based Approach. *IEEE Transactions on Software Engineering*, 32(10):831–848, 2006.

[LNZ+05] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken and Michael I. Jordan. Scalable Statistical Bug Isolation. *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2005.

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo and Yuanyuan Zhou. Learning from Mistakes – A Comprehensive Study on Real World Concurrency Bug Characteristics. *SIGARCH Computer Architecture News*, 36(1):329–339, 2008.

[LS97] Huan Liu and Rudy Setiono. Feature Selection via Discretization. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):642–645, 1997.

[LYY+05] Chao Liu, Xifeng Yan, Hwanjo Yu, Jiawei Han and Philip S. Yu. Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs. *Proceedings of the 5th SIAM International Conference on Data Mining (SDM)*. 2005.

[MAF08] Mary McGlohon, Leman Akoglu and Christos Faloutsos. Weighted Graphs and Disconnected Components: Patterns and a Generator. *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 2008.

[Mas09] Wes Masri. Fault Localization Based on Information Flow Coverage. *Software Testing, Verification and Reliability*, 20(2):121–147, 2009.

[Mat89] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. *Proceedings of the International Workshop on Parallel and Distributed Algorithms*. 1989.

[Mit97] Tom Mitchell. *Machine Learning*. McGraw Hill, 1997.

[MQB07] Madanlal Musuvathi, Shaz Qadeer and Thomas Ball. CHESS: A Systematic Testing Tool for Concurrent Software. Tech. Rep. MSR-TR-2007-149, Microsoft Research, 2007.

[MTV97] Heikki Mannila, Hannu Toivonen and A. Inkeri Verkamo. Discovery of Frequent Episodes in Event Sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.

[NBZ06] Nachiappan Nagappan, Thomas Ball and Andreas Zeller. Mining Metrics to Predict Component Failures. *Proceedings of the 28th International Conference on Software Engineering (ICSE)*. 2006.

[NK03] Siegfried Nijssen and Joost N. Kok. Efficient Discovery of Frequent Unordered Trees. *Proceedings of the first International Workshop on Mining Graphs, Trees and Sequences (MGTS) at ECML/PKDD*. 2003.

[NK04] Siegfried Nijssen and Joost N. Kok. A Quickstart in Frequent Structure Mining Can Make a Difference. *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 2004.

[NLHP98] Raymond T. Ng, Laks V. S. Lakshmanan, Jiawei Han and Alex Pang. Exploratory Mining and Pruning Optimizations of Constrained Associations Rules. *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1998.

[NTU+07] Sebastian Nowozin, Koji Tsuda, Takeaki Uno, Taku Kudo and Gökhan Bakir. Weighted Substructure Mining for Image Analysis. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*. 2007.

[OC03] Robert O'Callahan and Jong-Deok Choi. Hybrid Dynamic Data Race Detection. *SIGPLAN Notices*, 38(10):167–178, 2003.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The Program Dependence Graph in a Software Development Environment. *SIGSOFT Software Engineering Notes*, 9(3):177–184, 1984.

[Pan10] Victor Pankratius. Software Engineering in the Era of Parallelism. In Victor Pankratius and Samuel Kounev, eds., *Emerging Research Directions in Computer Science – Contributions from the Young Informatics Faculty in Karlsruhe*, pp. 45–52. KIT Scientific Publishing, 2010.

[PH+02] Brian Pitcher, Tom Hey et al. WebLech URL Spider, 2002. Software available at `http://weblech.sourceforge.net/`.

[PHL04] Jian Pei, Jiawei Han and Laks V. S. Lakshmanan. Pushing Convertible Constraints in Frequent Itemset Mining. *Data Mining and Knowledge Discovery*, 8(3):227–252, 2004.

[PHMA+04] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Jianyong Wang, Helen Pinto, Qiming Chen, Umeshwar Dayal and Mei-Chun Hsu. Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach. *IEEE Transactions on Knowledge and Data Engineering*, 16(10):1424–1440, 2004.

[PHW02] Jian Pei, Jiawei Han and Wei Wang. Mining Sequential Patterns with Constraints in Large Databases. *Proceedings of the 11th International Conference on Information and Knowledge Management (CIKM)*. 2002.

[PWDW09] Michael Philippsen, Marc Wörlein, Alexander Dreweke and Tobias Werth. ParSeMiS – The Parallel and Sequential Mining Suite. Department of Computer Science, School of Engineering, Friedrich-Alexander University of Erlangen-Nürnberg, Erlangen, Germany, 2009. Software available at `http://www2.informatik.uni-erlangen.de/EN/research/ParSeMiS/`.

[QDLT09] Yao Qi, Raja Das, Zhi Da Luo and Martin Trotter. MulticoreSDK: A Practical and Efficient Data Race Detector for Real-World Applications. *Proceedings of the 7th Workshop on Parallel and Distributed Systems (PADTAD)*. 2009.

[Qui93] John Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[RAF04] Nick Rutar, Christian B. Almazan and Jeffrey S. Foster. A Comparison of Bug Finding Tools for Java. *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE)*. 2004.

[RS09] Sayan Ranu and Ambuj K. Singh. GraphSig: A Scalable Approach to Mining Significant Subgraphs in Large Graph Databases. *Proceedings of the 25th International Conference on Data Engineering (ICDE)*. 2009.

[RTI02] Research Triangle Institute RTI. The Economic Impacts of Inadequate Infrastructure for Software Testing. Planning Report 02-3, National Institute of Standards and Technology (NIST), Gaithersburg, USA, 2002.

[SA96a] Ramakrishnan Srikant and Rakesh Agrawal. Mining Quantitative Association Rules in Large Relational Tables. *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1996.

[SA96b] Ramakrishnan Srikant and Rakesh Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. *Proceedings of the 5th International Conference on Extending Database Technology (EDBT)*. 1996.

[Sau05] Frank Sauer. Eclipse Metrics Plugin, 2005. Software available at `http://metrics.sourceforge.net/`.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[SJYH09] Raul Andres Santelices, James A. Jones, Yanbing Yu and Mary Jean Harrold. Lightweight Fault-Localization Using Multiple Coverage Types. *Proceedings of the 31st International Conference on Software Engineering (ICSE)*. 2009.

[SKT08] Hiroto Saigo, Nicole Krämer and Koji Tsuda. Partial Least Squares Regression for Graph Mining. *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 2008.

[SNK+09] Hiroto Saigo, Sebastian Nowozin, Tadashi Kadowaki, Taku Kudo and Koji Tsuda. gBoost: A Mathematical Programming Approach to Graph Classification and Regression. *Machine Learning*, 75:69–89, 2009.

[Som10] Ian Sommerville. *Software Engineering*. Pearson Education, 9th edn., 2010.

[SOO09] Masaki Shinoda, Tomonobu Ozaki and Takenao Ohkawa. Weighted Frequent Subgraph Mining in Weighted Graph Databases. *Proceedings of the 3rd International Workshop on Domain-Driven Data Mining (DDDM)*. 2009.

[SZZ06] Adrian Schröter, Thomas Zimmermann and Andreas Zeller. Predicting Component Failures at Design Time. *Proceedings of the 5th International Symposium on Empirical Software Engineering*. 2006.

[TCG+10] Marisa Thoma, Hong Cheng, Arthur Gretton, Jiawei Han, Hans-Peter Kriegel, Alex Smola, Le Song, Philip S. Yu, Xifeng Yan and Karsten M. Borgwardt. Discriminative Frequent Subgraph Mining with Optimality Guarantees. *Statistical Analysis and Data Mining*, 3(5):302–318, 2010.

[TUYT07] Rachel Tzoref, Shmuel Ur and Elad Yom-Tov. Instrumenting Where it Hurts – An Automatic Concurrent Debugging Technique. *Proceedings of the 16th International Symposium on Software Testing and Analysis (ISSTA)*. 2007.

[Vap95] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, 1995.

[WC87] Andrew K. C. Wong and David K. Y. Chiu. Synthesizing Statistical Knowledge from Incomplete Mixed-Mode Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 9(6):796–805, 1987.

[WF05] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Series in Data Management Systems. Morgan Kaufmann, 2nd edn., 2005.

[WH04] Jianyong Wang and Jiawei Han. BIDE: Efficient Mining of Frequent Closed Sequences. *Proceedings of the 20th International Conference on Data Engineering (ICDE)*. 2004.

[WMFP05] Marc Wörlein, Thorsten Meinl, Ingrid Fischer and Michael Philippsen. A Quantitative Comparison of the Subgraph Miners MoFa, gSpan, FFSM, and Gaston. *Proceedings of the 10th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*. 2005.

[Wu96] Xindong Wu. A Bayesian Discretizer for Real-Valued Attributes. *The Computer Journal*, 39(8):688–691, 1996.

[WZW+05] Chen Wang, Yongtai Zhu, Tianyi Wu, Wei Wang and Baile Shi. Constraint-Based Graph Mining in Large Database. *Proceedings of the 7th Asia-Pacific Web Conference (APWeb)*. 2005.

[XTLL09] Tao Xie, Suresh Thummalapenta, David Lo and Chao Liu. Data Mining for Software Engineering. *Computer*, 42(8):55–62, 2009.

[XY05] Yi Xia and Yirong Yang. Mining Closed and Maximal Frequent Subtrees from Databases of Labeled Rooted Trees. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 17(2):190–202, 2005.

[YCHY08] Xifeng Yan, Hong Cheng, Jiawei Han and Philip S. Yu. Mining Significant Graph Patterns by Leap Search. *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2008.

[YH02] Xifeng Yan and Jiawei Han. gSpan: Graph-Based Substructure Pattern Mining. *Proceedings of the 2nd IEEE International Conference on Data Mining (ICDM)*. 2002.

[YH03] Xifeng Yan and Jiawei Han. CloseGraph: Mining Closed Frequent Graph Patterns. *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 2003.

[YH06] Xifeng Yan and Jiawei Han. Discovery of Frequent Substructures. In Cook and Holder [CH06], chap. 5, pp. 99–115.

[YHA03] Xifeng Yan, Jiawei Han and Ramin Afshar. CloSpan: Mining Closed Sequential Patterns in Large Databases. *Proceedings of the Int. Conference SIAM Data Mining (SDM)*. 2003.

[YHF10] Philip S. Yu, Jiawei Han and Christos Faloutsos, eds. *Link Mining: Models, Algorithms, and Applications*. Springer, 2010.

[Zak00] Mohammed J. Zaki. Scalable Algorithms for Association Mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, 2000.

[Zak01] Mohammed J. Zaki. SPADE: An Efficient Algorithm for Mining Frequent Sequences. *Machine Learning*, 42(1–2):31–60, 2001.

[Zel99] Andreas Zeller. Yesterday, my Program Worked. Today, it Does Not. Why? *Proceedings of the 7th European Software Engineering Conference and the 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1999.

[Zel02] Andreas Zeller. Isolating Cause-Effect Chains from Computer Programs. *ACM SIGSOFT Software Engineering Notes*, 27(6):1–10, 2002.

[Zel09] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2nd edn., 2009.

[ZH02] Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28:183–200, 2002.

[ZNZ08] Thomas Zimmermann, Nachiappan Nagappan and Andreas Zeller. Predicting Bugs from History. In Tom Mens and Serge Demeyer, eds., *Software Evolution*, chap. 4, pp. 69–88. Springer, 2008.

[ZYHY07] Feida Zhu, Xifeng Yan, Jiawei Han and Philip S. Yu. gPrune: A Constraint Pushing Framework for Graph Pattern Mining. *Proceedings of the 11th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*. 2007.