Pontifícia Universidade Católica
do Rio de Janeiro

Marcelo Garnier Mota

EXPLORING STRUCTURED INFORMATION RETRIEVAL FOR BUG LOCALIZATION IN C# SOFTWARE PROJECTS

Dissertação de Mestrado

Dissertation presented to the Programa de Pós-Graduação em Informática, of the Departamento de Informática do Centro Técnico Científico da PUC-Rio, as partial fulfillment of the requirements for the degree of Mestre.

Advisor: Prof. Alessandro Fabrício Garcia

Rio de Janeiro
September 2016

Pontifícia Universidade Católica
do Rio de Janeiro

Marcelo Garnier Mota

Exploring structured information retrieval for bug localization in C# software projects

Dissertation presented to the Programa de Pós-Graduação em Informática, of the Departamento de Informática do Centro Técnico Científico da PUC-Rio, as partial fulfillment of the requirements for the degree of Mestre.

Prof. Alessandro Fabrício Garcia
Advisor
Departamento de Informática – PUC-Rio

Prof. Arndt von Staa
Departamento de Informática – PUC-Rio

Prof. Carlos José Pereira de Lucena
Departamento de Informática – PUC-Rio

Prof. Márcio da Silveira Carvalho
Coordinator of the Centro Técnico Científico – PUC-Rio

Rio de Janeiro, September 16th, 2016

Marcelo Garnier Mota

Marcelo Garnier received a Technologist degree in Informatics from the Federal Center of Technological Education (CEFET) in 2002 and a BSc degree in Civil Engineering from the State University of Northern Rio de Janeiro (UENF) in 2005. He currently works as a software architect for Petróleo Brasileiro S.A. (Petrobras), where he works since 2004. During this period, he occupied various positions in the software development field, such as quality assurance analyst, scrum master, and development team lead. His research interests include software maintainability, code quality, static analysis, and software metrics.

Bibliographic data

Mota, Marcelo Garnier

Exploring structured information retrieval for bug localization in C# software projects / Marcelo Garnier Mota; advisor: Alessandro Fabrício Garcia. – 2016.

91 f. : il. color. ; 30 cm

Dissertação (mestrado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2016.

Inclui bibliografia.

1. Informática – Teses. 2. Defeitos. 3. Localização de defeitos. 4. Recuperação de informação. I. Garcia, Alessandro Fabrício. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To Cibele, for her love and continued support.

Acknowledgments

I thank all the professors from the Informatics Department at PUC-Rio for their contribution to my education. I also thank Prof. Clevi Rapkiewicz, Prof. Marcelo Feres, and Prof. Rogério Atem, for their contributions during my graduation.

I thank Petróleo Brasileiro S.A. (Petrobras), for financially supporting my studies. I specially thank my teammates at Petrobras for assuming my tasks during my absences.

I warmly thank my colleagues at the Opus Research Group, for their valuable contributions during our meetings and rehearsals. Moreover, I thank for the fellowship. I will fondly remember the good times spent at the lab.

I am enormously grateful to my advisor, Prof. Alessandro Garcia. I thank, in first place, for the opportunity to join his research group, in spite of my schedule limitations. In addition, I thank for his enthusiasm, which helped me to keep motivated. I also thank him for his tireless pursuit for excellence, which raised the quality of my work to a higher level. Finally, I thank for his understanding and patience, particularly when I could not keep up with his breathtaking pace.

I thank my parents, José Carlos and Maria Elena, for supporting me and for having strived to provide me the best possible education since my childhood.

I thank my children, Lara, Guilherme, Lucas, and Felipe. They are sources of inspiration, motivation, and happiness. I thank Lara for her precious help, taking good care of Felipe while I was busy. Moreover, I immensely thank Guilherme and Lucas for understanding when their dad could not play with them.

Finally, I thank my wife, Cibele, for… everything. It is hard to find the right words to express how much I owe to her. Without her support, this work would not be possible. Thank you for finding strength to endure these troubled times, and for always being there for me.

Abstract

Mota, Marcelo Garnier; Garcia, Alessandro Fabrício (Advisor). **Exploring structured information retrieval for bug localization in C# software projects**. Rio de Janeiro, 2016. 91p. MSc. Dissertation – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Software projects can grow very rapidly, reaching hundreds or thousands of files in a relatively short time span. Therefore, manually finding the source code parts that should be changed in order to fix a bug is a difficult task. Static bug localization techniques provide effective means of finding files related to a bug. Recently, structured information retrieval has been used to improve the effectiveness of static bug localization, being successfully applied by techniques such as BLUiR, BLUiR+, and AmaLgam. However, there are significant shortcomings on how these techniques were evaluated. BLUiR, BLUiR+, and AmaLgam were tested only with four projects, all of them structured with the same language, namely, Java. Moreover, the evaluations of these techniques (i) did not consider appropriate program versions, (ii) included bug reports that already mentioned the bug location, and (iii) did not exclude spurious files, such as test files. These shortcomings suggest the actual effectiveness of these techniques may be lower than reported in recent studies. Furthermore, there is limited knowledge on whether and how the effectiveness of these state-of-the-art techniques can be improved. In this dissertation, we evaluate the three aforementioned techniques on 20 open-source C# software projects, providing a rigorous assessment of the effectiveness of these techniques on a previously untested object-oriented language. Moreover, we address the simplistic assumptions commonly present in bug localization studies, thereby providing evidence on how their findings may be biased. Finally, we study the contribution of different program construct types to bug localization. This is a key aspect of how structured information retrieval is applied in bug localization. Therefore, understanding how each construct type

influences bug localization may lead to effectiveness improvements in projects structured with a specific programming language, such as C#.

Keywords

Bugs; bug localization; information retrieval.

Resumo

Mota, Marcelo Garnier; Garcia, Alessandro Fabrício. **Explorando recuperação de informação estruturada para localização de defeitos em projetos de software C#**. Rio de Janeiro, 2016. 91p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Projetos de software podem crescer rapidamente, alcançando centenas ou milhares de arquivos num período relativamente curto. Portanto, torna-se difícil a tarefa de encontrar partes do código-fonte que devem ser modificadas para consertar um defeito. Técnicas de análise estática para localização de defeitos fornecem um meio eficaz de encontrar arquivos relacionados a um defeito. Recentemente, recuperação de informação estruturada vem sendo usada para aumentar a eficácia de localização estática de defeitos, sendo aplicada com sucesso por técnicas como BLUiR, BLUiR+ e AmaLgam. No entanto, existem limitações significativas na maneira como essas técnicas foram avaliadas. BLUiR, BLUiR+ e AmaLgam foram testadas em apenas quatro projetos, todos eles estruturados com a mesma linguagem, Java. Adicionalmente, as avaliações dessas técnicas (i) não consideraram versões apropriadas dos programas, (ii) incluíram relatórios de falhas que já mencionavam a localização do defeito, e (iii) não excluíram arquivos espúrios, como arquivos de teste. Essas limitações sugerem que a eficácia real dessas técnicas seja menor do que o informado em estudos recentes. Além do mais, há limitações no conhecimento sobre se e como a eficácia dessas técnicas do estado-da-arte pode ser aumentada. Nesta dissertação, nós avaliamos as três técnicas supracitadas em 20 projetos C# de código aberto, fornecendo uma avaliação rigorosa da eficácia dessas técnicas numa linguagem orientada a objetos não testada anteriormente. Além disso, nós endereçamos os pressupostos simplistas comumente presentes em estudos de localização de defeitos, fornecendo assim evidências sobre como seus achados podem ser enviesados. Finalmente, nós estudamos a contribuição de diferentes tipos de construtos de programa para a

localização de defeitos. Este é um aspecto-chave na forma como recuperação de informação estruturada é aplicada em localização de defeitos. Portanto, entender como cada tipo de construto influencia a localização de defeitos pode levar a melhorias na eficácia em projetos estruturados com linguagens de programação específicas, como C#.

Palavras-chave

Defeitos; localização de defeitos; recuperação de informação.

Summary

List of figures

List of tables

But I still haven't found
What I'm looking for…

U2, *I Still Haven't Found What I'm Looking For*

1
Introduction

Software *failures* represent a serious concern for developers and maintainers. A *failure* occurs when a system does not perform a required function according to its specification [1]. It is widely known that the later a failure occurs, the higher the cost to fix it. Software failures are caused by *defects* in the source code. A *defect* is a problem in the source code that, if not corrected, could cause an application to either fail or produce incorrect results [1]. Therefore, effectively identifying and removing defects associated with failures is a routine activity to software maintainers.

Despite their different meanings, both defects and failures are popularly referred to as *bugs* [2] [3] [4] [5]. When bugs occur in a software system, they are usually reported to a developer or development team. Each report, colloquially called a *bug report*, contains information about the circumstances in which the bug occurred (Section 2.1.1). This information is used by developers to investigate and fix the bug. In order to fix a bug, one must first know where in the source code it is located. The activity of locating the portion of the source code that must be modified to fix a bug using information from a bug report is called *bug localization* [2].

Manual bug localization is a painstaking activity [3]. Therefore, effective methods for automatically locating bugs from bug reports are highly desirable [4], as they would reduce software maintenance effort [3]. Automated techniques for static bug localization have been a popular research topic [3] [4] [5] [6] [7] and attracted the attention of many well-known software companies, such as Google [8] and Microsoft [9]. Static bug localization techniques have the benefit of being applicable at any stage of the software development process [2]. Differently from dynamic techniques, they do not require large test suites, which are often not available [10] [11]. Thus, static bug localization techniques are more flexible, as they can be applied to a wider range of scenarios, such as legacy software systems where automated test suites were not originally implemented.

To foster the process of effectively identifying source code that is relevant to a particular bug report, a number of techniques have been developed using *information retrieval* models, such as Latent Dirichlet Allocation (LDA) [2], Vector Space Model (VSM) [3], Latent Semantic Analysis (LSA) [12], Clustering [12], and various combinations. *Information retrieval* (IR) consists of finding documents within a collection that match a search query [13]. The IR approach to bug localization generally consists of treating source files as documents and bug reports as queries. Source files that share more terms with the bug report are ranked as having a higher probability of containing the bug. The effectiveness of bug localization techniques is commonly measured by their ability to rank potentially buggy files at the first positions of a list (Section 2.2.3).

Recently, *structured* information retrieval has also been used for bug localization [4] [5]. Traditional information retrieval handles documents as a "bag of words" [13], meaning that every term in the document is indistinctly computed, regardless of its position, order, or function in the document. Conversely, structured IR splits a document according to relevant fields or zones [13]. Applying this principle to the bug localization domain, structured IR-based techniques map a set of program *constructs*, such as class and variable names [4] [5], to document fields. This principle allows the techniques to place different emphasis on different program constructs in order to calculate textual similarity.

Bug localization techniques based on structured information retrieval, such as BLUiR [4] and AmaLgam [5], have shown improvements over other traditional IR approaches (Section 2.3). BLUiR [4] and AmaLgam [5] are currently the best-performing IR-based bug localization techniques available (Section 2.3). In spite of the promising results, these studies contain significant shortcomings. BLUiR and AmaLgam have been evaluated in only four projects, implemented in a single programming language (namely, Java), and under non-realistic experimental setups (Section 3.4). For instance, these studies [4] [5] include in their effectiveness evaluation bug reports that already mention the location of the bug in their description. Developers are unlikely to benefit from the assistance of automated techniques to localize this kind of bug. Moreover, such bug reports have been shown to influence bug localization results [14]. Therefore, they should be removed from effectiveness evaluations of bug localization techniques (Section 3.4.2).

Additionally, these techniques need to be evaluated on a higher number of projects, structured in languages other than Java.

1.1 Problem statement

Although IR-based bug localization has been an active research topic in recent years (e.g., [2] [3] [4] [5] [6] [7] [10] [12] [14] [15]), techniques based on this approach are still not widely used in practice [10]. A significant part of the effort spent to contribute to the state-of-the-art in bug localization is concentrated on the development of new, more effective techniques (e.g., [2] [3] [4] [5] [6] [7] [12] [15]), although only a few of the mentioned studies are dedicated to structured IR [4] [5] [15]. Another thread discusses practical aspects of incorporating such techniques into developers' workflows [10] [16]. In this dissertation, we focus on the first thread, as there are significant shortcomings on how some of these techniques [4] [5] were evaluated.

First, the experimental setups of these studies [4] [5] adopt non-realistic assumptions, which add bias to the evaluation of the techniques (Section 1.2.1). Thus, the techniques may be less effective than reported (Section 1.2.2). Furthermore, there is still limited understanding on how structured information retrieval can be used to further increase the techniques' effectiveness (Section 1.2.3). Particularly, we explore the influence of program constructs – a key aspect of structured information retrieval applied to bug localization – on bug localization effectiveness.

Next, we summarize the problem addressed in this dissertation in a single statement, followed by further discussion in Section 1.2.

Bug localization techniques based on structured information retrieval have been evaluated under non-realistic experimental setups, which suggests they might be less effective than reported, and there is limited knowledge on how to use structured information retrieval to increase their effectiveness.

1.2 Limitations of related work

This section discusses shortcomings of recent studies on IR-based bug localization. First, we highlight important issues in the experimental setup of recent evaluations that may be skewing reported results. Next, we discuss to which extent current effectiveness of IR-based bug localization techniques allows them to be used in practice. Finally, we discuss how specific characteristics of structured IR could be applied to bug localization in order to improve bug localization effectiveness.

1.2.1 Biased evaluations

The evaluation of a bug localization technique should aim to reproduce, to the maximum possible extent, the actual scenario where the techniques might be used in practice. This scenario involves the process followed by a development team to fix a reported bug. The team would retrieve the source code from the version of the program where the problem occurred and look for the bug, based on the information provided by the bug report. If this information includes program elements, such as a class or a method name, developers would probably start their investigation by opening the file containing the mentioned element. However, if such information is not present, developers might consider using an automated bug localization technique to generate a list of files potentially related to the bug. We will see, however, that current bug localization techniques fail to include these premises in their experimental setup, thus biasing their evaluations.

Previous studies on IR-based bug localization provide little or no information about the version of the project that is used as input to the bug localization technique. In many studies, as Rao and Kak [17] point out, researchers have merely chosen a single version of the project and run the localization algorithm for all available bug reports on that same version. However, a rigorous evaluation of bug localization effectiveness should consider the appropriate project version, i.e., the version where the bug actually occurred, for every bug report under analysis [17]. Thus, the version where each bug occurred should be determined and corresponding source code obtained before performing bug localization. Without this step, there

is a high chance that the bug is not even present on the code being analyzed. Any result obtained in these cases would be random and, thus, useless.

Another shortcoming of previous studies involves what Kochhar et al. [14] call *localized bug reports*, i.e., bug reports that already contain references to the files containing the bug on its own description. These bug reports should *not* be considered in the evaluation of bug localization techniques for the following reasons. First, they artificially increase the effectiveness of the techniques [14]. Localized bug reports accounted for 54% of the bug reports studied in [14], and 49% in our own study (Section 3.4.2). Therefore, the incidence of localized bug reports cannot be overlooked, as it significantly influences bug localization results [14]. Second, it is unlikely that developers would even need assistance from an automated technique to localize such bugs. As Wang et al. point out, when a bug report mentions a program entity, developers use its name as a keyword for searching source files [10]. This provides developers with a good starting point for investigating the failure. Therefore, in order to generate relevant contributions, bug localization techniques should focus on bug reports with no identifiable information [10], i.e., *non-localized* bug reports [14].

Finally, test files should not be included in the bug localization scope, as they might inappropriately influence bug localization results. This happens because the oracle in bug localization studies [3] [4] [5] corresponds to the files modified to fix the bug. These files are obtained from the source control, by determining which commit was related to the resolution of the bug (Section 3.2). Such commits often include test files, which might have been modified as part of the bug resolution. However, test files rarely contain the code that *triggered* the bug (Section 3.4.3). Therefore, test files constitute false positives in the context of bug localization results. Thus, they should be removed from the search scope of bug localization techniques.

In addition to all the shortcomings described above, there is the fact that the techniques [3] [4] [5] were evaluated only on four Java projects. To the best of our knowledge, structured IR has never been studied on software projects written on any other object-oriented (OO) language. In fact, the only other language we could find being used in similar studies was C [15], a procedural language. The lack of studies applying bug localization to different OO languages threatens the generalizability of the results, as the projects selected in [3] [4] [5] might be

particularly suitable for these bug localization techniques, whereas other projects might not. Thus, it is important to assess the effectiveness of bug localization techniques on a higher number of projects, structured in different programming languages and encompassing different domains (Section 3.3). Providing evidence of effectiveness in different scenarios will increase confidence in the reported results.

1.2.2 Limited effectiveness

Bug localization techniques return their results as a list of files ranked by the probability of relationship with the bug (Section 2.2.1). The lists returned by bug localization techniques are commonly evaluated up to the tenth position [3] [4] [5]. This convention acts as an implicit effectiveness threshold. In other words, bug localization techniques should be able to return at least one buggy file among the first ten positions, as it would not be reasonable to expect the developer to examine more than ten files to localize a bug.

However, in spite of continuous improvement fostered by recent studies (i.e., [2] [3] [4] [5] [6] [7] [12]), IR-based techniques are still not effective enough to be widely used in practice. Consider, for example, AmaLgam [5], one of the best performing bug localization techniques based on IR. According to [5], AmaLgam was able to return a buggy file at the top of the list for up to 62% of the bugs. Considering the 10 first positions of the list, AmaLgam was able to return a buggy file in the top-10 positions up to 90% of the time. This means that, 10% of the time, developers would have to inspect more than 10 files to find a buggy file. These are the best results from an evaluation performed on only four projects [5]. However, actual effectiveness may be even lower, due to experimental shortcomings presented in the previous subsection (and further discussed in Section 3.4).

In order to increase effectiveness of IR-based bug localization, researchers have been incorporating additional sources of information to bug localization techniques, e.g., version history [5] [7], bug report history [3] [5], and source file structure [4] [5] [15]. Results from BLUiR [4] and AmaLgam [5] suggest that structured information retrieval applied to source file structure was the prominent factor for the effectiveness increase reported by these studies. Unfortunately, structured IR has not been explored enough within the bug localization domain. In

particular, it is unknown how specific program constructs contribute to structured IR-based techniques. As the possibility of handling program constructs differently is the key advantage provided by structured IR, knowledge about the influence of different program constructs could be used to increase bug localization effectiveness, as we will see in the next subsection.

1.2.3 Unknown contribution of constructs

Structured information retrieval has emerged as a prominent factor in increasing bug localization effectiveness. The difference lies on how documents are handled. Traditional IR-based techniques deal with all the terms contained in documents without distinction. Conversely, bug localization techniques based on structured IR break documents up according to their underlying structure. Source files, for example, are split according to types of program constructs, e.g., classes and methods (Section 2.3.2). Thus, a single source file would be handled as several distinct documents, each containing exclusively terms corresponding to the respective construct type. Consequently, the importance of terms in a document is modeled not only by the number of occurrences, but also by the number of construct types in which they appear. Therefore, constructs are central features to the functioning of bug localization based on structured information retrieval.

It has been preliminarily observed that considering source file structure improves bug localization effectiveness [4] [5]. However, few studies have investigated structured IR so far [4] [5] [15], leaving a number of questions regarding usage of program constructs remain unanswered. For instance, should every available construct type be considered? Constructs not explored in existing techniques may also be able to improve effectiveness. Is there any construct type whose contribution is negative? If so, effectiveness could be improved even further by ignoring these specific constructs. Are the contributions from every type of construct equivalent? If they are not, the relevance of most contributing ones could be highlighted by assigning higher weights to these constructs. Pondering the effectiveness improvement brought by the usage of structured IR, we consider that investigating such questions in detail have the potential to contribute to the state-of-the-art in structured IR-based bug localization.

1.3 Goals and research questions

The shortcomings regarding the evaluation of structured IR-based bug localization techniques [4] [5] have been discussed in Section 1.2.1. Addressing these shortcomings is also an opportunity to evaluate them on different scenarios. An initial step in that direction is to understand the behavior of bug localization techniques applied to an object-oriented programming language that is slightly different from Java. Java has been the focus of previous studies of structured IR-based techniques for bug localization (Section 2.3). Nonetheless, software engineers remain unaware to what extent they can rely on these techniques to perform bug localization activities in projects structured with other programming languages.

Thus, as an alternative to Java, we have selected projects written in C# (pronounced as *see sharp*). C# is a general-purpose, object-oriented language that shares many traits with the Java language, but also have some distinct programming features. For example, some widely used C# constructs, like properties and structures ("structs"), are inexistent in Java. Moreover, C# is a popular language [18] that figures within the top 10 languages in number of GitHub repositories [19]. To the best of our knowledge, neither of these techniques have been previously evaluated on other object-oriented programming languages, such as C#.

Considering the problem stated in the Section 1.1 and further discussed in Section 1.2, the goal of this dissertation can be stated as follows.

Perform a realistic, in-depth effectiveness evaluation
of state-of-the-art bug localization techniques
on a previously untested programming language.

The limitations discussed in Section 1.2.1 undermine the results obtained by state-of-the-art bug localization techniques based on structured information retrieval [4] [5]. Consequently, there is a need to address those shortcomings and perform a realistic evaluation of those techniques. Moreover, in spite of the potential brought by structured information retrieval, the limited effectiveness of current techniques (Section 1.2.2) encourages an in-depth evaluation of which structured IR aspects can be explored to increase bug localization effectiveness

further. These aspects are directly related with how bug localization uses programming language constructs (Section 1.2.3).

The proposed goal unfolds in the following research questions:

RQ1: Are BLUiR, BLUiR+, and AmaLgam effective to locate bugs in C# projects?

This research question aims to provide a first evaluation of state-of-the-art bug localization techniques on C# software projects (Section 3.6.1). Results for this research question cannot be compared with results from Java studies due to the distinct nature and quantity of selected projects (Section 3.3). Nevertheless, this evaluation will include preparation steps (Section 3.4) needed to mitigate biases that are commonly found in bug localization evaluations (Section 1.2.1), thus providing evidence on how these steps cannot be omitted from the experimental setup of bug localization studies.

RQ2: Does the addition of more program constructs increase the effectiveness of bug localization on C# projects?

To answer this research question, a first attempt of exploring structured IR to increase bug localization effectiveness is made. Initially, we adapt a bug localization technique to explicitly consider all C# constructs available (Section 3.5). Then, we run the adaptations on the selected projects, and compare the results with those obtained in the first research question (Section 3.6.2).

RQ3: Which program constructs contribute more to the effectiveness of bug localization on C# projects?

This question aims to quantify the contribution of different program constructs to bug localization effectiveness. A statistical procedure called *principal component analysis* (PCA) is used for this purpose (Section 4.3). This procedure extracts components from the data, sorted by relevance, and identifies the correlation of each construct to the most relevant components. The answer to this RQ will enable further experimentation with program constructs, formalized in the next two questions.

RQ4: Does the effectiveness of bug localization increase with the suppression of constructs with the lowest contributions?

Data from RQ3 may reveal that some construct correlate negatively to the more relevant components, suggesting they may be contributing negatively to bug localization effectiveness. In this case, the technique will be applied again with these constructs suppressed, in order to verify if the suppression of these constructs causes the effectiveness of the technique to increase.

RQ5: Does the effectiveness of bug localization increase with the emphasis on constructs with the highest contributions?

RQ3 will also reveal which construct correlate positively to the more relevant components, i.e., which ones contribute more to bug localization effectiveness. Knowing which constructs fall in this category will allow us to modify the structural similarity scores (Section 2.3.2) by assigning higher weights to the similarity associated with these constructs. The goal of RQ5 is to verify if this modification is able to increase the effectiveness of bug localization.

1.4 Dissertation outline

This dissertation is organized as follows. Chapter 2 contains background information, providing an overview about bug localization and information retrieval. Chapter 3 adapts the selected bug localization technique and compares the adaptation against the original techniques in terms of effectiveness. This chapter also discusses commonly neglected issues in the experimental setup of previous bug localization studies, detailing how we solved them in our experiments. Chapter 4 builds on the findings from previous chapter to measure the contribution of different program constructs in order to improve bug localization effectiveness further. Chapter 5 concludes.

2
Background

Bug localization is a complex process, which can be carried out through distinct approaches. Each approach to bug localization encompasses its own set of concepts, borrowed from different areas of knowledge. For instance, information retrieval is an area closely related to static bug localization, as information retrieval is a key technology applied by techniques that follow a static analysis approach [3] [4] [5] [15]. Thus, it becomes necessary to provide some background information for a complete understanding of the process and the technologies employed by bug localization techniques.

In this chapter, we describe the main concepts involved in bug localization. We summarize the main characteristics of dynamic and static approaches to bug localization, and we discuss the motivation for focusing on static analysis techniques. Afterwards, we provide an overview of information retrieval. We describe the underlying information retrieval model used by most static bug localization techniques, the vector space model (VSM). Then, we demonstrate how VSM works, providing an example using a term weighting scheme based on term frequency statistics, popularly known as TF-IDF (term frequency – inverse document frequency). Finally, we present the bug localization techniques that will be used throughout this dissertation. These techniques represent the state-of-the-art on bug localization based on structured information retrieval.

2.1
Bug localization

2.1.1
Overview

In spite of all the effort invested by developers and testers to produce bug-free software, in practice every program contains bugs. Maintainers must be warned about the occurrence of bugs so they can fix them. Communication between project members is usually supported by *issue tracking systems*. Issue trackers may be

integrated in code hosting services, such as GitHub[1] and BitBucket[2], or stand-alone applications, such as Jira[3]. Issue trackers allow the creation of *issues*, which can represent distinct concepts from software development lifecycle, such as tasks, features, and *bugs*. The concept represented by an issue (e.g., task, feature, bug) can be identified by *labels* (or *tags*). Labels may also be used to classify issues according to other criteria, such as module or severity, for example. Some systems, such as Bugzilla[4], are *bug trackers*. They are similar to issue trackers, however focusing only on tracking bugs.

When a team uses an issue tracker, the occurrence of a bug is commonly reported by creating an issue and labeling it as "bug". We will refer to issues that report bugs as *bug reports*, in order to differentiate these issues from those related to other concepts or events.

Although the contents of bug reports may vary, depending on the tracking system being used, most of them contain a few common attributes [20] [21]: an *identification number* (*id*); a short *summary* (or *title*); a full *description*; a creation *date*; a *status*; a *reporter*, the user who created the report; and an *assignee*, the person currently working on the bug.

Bug reports can be created by different stakeholders, such as developers, testers, end users, or help desk operators. Typically, the person who creates a bug report is not responsible for fixing it. The report serves to communicate information about the bug to maintainers, who rely on this report to perform the necessary activities for identifying and removing the cause of the bug. Before actually modifying source code in order to fix the bug, developers need to find the defective source code based on the information provided in the bug report. This activity is called *bug localization*.

Perhaps one of the most common forms of bug localization is debugging with the aid of an integrated development environment (IDE). Developers may load the source code and run an application statement by statement until they reach the part of the code that triggers the bug. This approach is costly, especially for large projects, because the number of files and statements that need to be inspected until the bug is reproduced can be very high. Moreover, there might be no information

[1] https://github.com/
[2] https://bitbucket.org/
[3] https://www.atlassian.com/software/jira
[4] https://www.bugzilla.org/

available about the parts of a program that would possibly trigger the bug. Thus, debugging would still benefit from the use of complementary techniques that could narrow the search space where the developer has to find the bug.

Automated bug localization techniques fulfill this role, by helping developers to locate defective source code. These techniques take as input information about a subject software system and produce as output a list of files potentially related to each bug [22]. Automatic bug localization techniques are mainly divided into *dynamic* and *static* techniques [4]. The next subsections briefly discuss the main characteristics of each type of technique.

2.1.2 Dynamic techniques

Dynamic bug localization refers to techniques that rely on program execution to localize bugs. A common approach uses program spectra, i.e., collections of data that provide a specific view on the dynamic behavior of software [23]. Spectrum-based techniques (e.g., [24] [25]) use program execution information to track program behavior [26]. When an execution fails, this information can be used to identify suspicious code that is responsible for the failure. By identifying parts of the program covered during an execution, it is possible to identify the components involved in a failure. Spectrum-based techniques rely on the contrast between the passing and failing executions to localize bugs effectively [26]. For this reason, numerous test cases must be available [27].

Other approaches to dynamic bug localization include model-based [28], program state-based [29] [30], and mutation-based [31] techniques. Note this list is by no means complete. Regardless of the approach, dynamic techniques share a set of common advantages and drawbacks. The main advantage is precision: dynamic techniques are often capable of locating bugs at statement level [4]. However, as aforementioned, they usually require large test suites. In fact, as several passing and failing test cases need to be provided, dynamic techniques are only viable in projects where a comprehensive test suite is previously available. Nonetheless, this is not the case in most software projects [10] [11]. Thus, the applicability of dynamic techniques is severely reduced.

2.1.3 Static techniques

Static bug localization techniques are largely based on information retrieval (e.g., [2] [3] [4] [5] [7] [12] [15]). Information retrieval-based techniques aim to locate a bug from its textual description [12]. Therefore, these techniques require only source code and bug reports in order to operate [4]. Nevertheless, static techniques are frequently combined with additional information in order to improve their effectiveness. Examples include change history [5] [7], bug report history [3] [5], source code structure [4] [5] [15], and file authorship information [32].

Static techniques usually do not reach the same effectiveness delivered by dynamic techniques. However, contrary to dynamic techniques, static bug localization techniques do not require program execution. Thus, they do not need a working subject system, which allows them to be applied at any stage of the software development process [2]. For the same reason, static techniques do not require test cases as well. Having less prerequisites grants flexibility to static techniques, which enables them to be applied on a wider range of scenarios, compared to their dynamic counterparts. This flexibility is particularly important in the case of legacy software systems, where automated tests might not have been originally implemented.

We discuss static, information retrieval-based bug localization techniques individually in Section 2.3. Before that, in the next section, we present a brief overview of information retrieval.

2.2 Information retrieval

2.2.1 Overview

Information retrieval (IR) consists of finding documents within a collection that match a search query [13]. When applying IR to bug localization, source code files become the collection of documents, and the bug report represents the query. Then, the task of finding buggy files is reduced to the IR problem of determining the relevance of a document to a query. Relevance is determined by preprocessing

the query and the set of documents and then calculating the textual similarity between each document and the query.

Preprocessing consists of three steps: text normalization, stop word removal and stemming. Text normalization extracts a list of terms that represents the documents and the query by removing punctuation marks and performing case folding. In the bug localization domain, normalization steps usually include identifier splitting. Many identifiers are made by concatenating words, either `separated_by_underscores` or using mixed case, such as `camelCase` or `PascalCase`. For example, an identifier named `MemoryMappedFile` would be split as “memory”, “mapped” and “file”. Therefore, splitting identifiers improves recall [4], by adding terms that represent conceptual information encoded in compound identifiers [33].

After normalization, common words, which usually add little value to a retrieval operation, are removed from the list of terms. These words (e.g., “the”, “to”, “get”) are called stop words [13]. The final preprocessing step is to reduce inflectional forms to a common base form in order to improve term matching by representing similar words with the same term. This can be accomplished via stemming or lemmatization. Stemming usually refers to a heuristic process that strips off derivational suffixes from words [13]. Lemmatization, on the other hand, uses a vocabulary and performs morphological analysis of words, aiming to return the base or dictionary form of a word (i.e., its lemma) [13]. Although lemmatization is more accurate than stemming, the latter is usually the preferred alternative for bug localization applications [2] [3] [4] [5] due to its simplicity. An ubiquitous stemming algorithm is the Porter Stemmer [34], used in [2] [3] [5] and available online [35].

After preprocessing, the similarity between documents and the query must be calculated. The most common approach is based on the vector space model (VSM) [3] [4] [5] [15]. Next subsection explains how documents and queries are represented in VSM, in order to allow similarity calculation.

2.2.2 Vector Space Model

In the vector space model, each document is expressed as a vector of term weights. These weights are typically the product of term frequency and inverse document frequency (TF-IDF) of each term. Therefore, if a collection of documents contains terms $(t_1, \dots, t_n)$, a document from this collection is represented as:

$$\vec{d} = \left(tf(t_1, d)idf(t_1), \ldots, tf(t_n, d)idf(t_n)\right)$$

Equation 1 – Vector representation of a document in VSM

$$idf(t) = \log \frac{N}{df_t}$$

Equation 2 – Inverse document frequency

In Equation 1, $tf(t_i)$ is the number of occurrences of t_i in d and $idf(t_i)$ is given by Equation 2, where N is the total number of documents in the collection and df_t is the document frequency, i.e., the number of documents that contain term t. Inverse document frequency serves to increase the weight of rare terms, i.e., terms that occur in few documents from the collection. If a term occurs in most of the documents, it has little discriminating power in determining relevance of a document [13]. Thus, terms that occur many times in a small number of documents are those that are assigned higher weights.

Given two documents, their similarity can be measured by computing the cosine similarity of their vector representations [13]:

$$sim(d_1, d_2) = \frac{\vec{V}(d_1) \cdot \vec{V}(d_2)}{\left|\vec{V}(d_1)\right|\left|\vec{V}(d_2)\right|}$$

Equation 3 – Cosine similarity

In Equation 3, the numerator represents the dot product (or inner product) of vectors $\vec{V}(d_1)$ and $\vec{V}(d_2)$, while the denominator is the product of their lengths [13]. To illustrate how cosine similarity works, we provide a simplified bug localization example. Consider a bug report that reads *"Error placing new customer order"*. Assume after applying all preprocessing steps (normalization, stop word removal, and stemming), the bug report content becomes *"place custom order"*. This will be the query used to localize similar source files, possibly related to this bug. Consider the subject system contains three files, which, after preprocessing, have the following content: *"customer"*, *"customer detail"*, and *"place order controller"*. Table 1 summarizes term and document frequencies for the terms in this example.

Table 1 – TF-IDF calculation

Term	Term frequency				Document frequency	Inverse document frequency
	Query	File 1	File 2	File 3		
place	1	0	0	1	2	0.301
customer	1	1	1	0	3	0.125
order	1	0	0	1	2	0.301
detail	0	0	1	0	1	0.602
controller	0	0	0	1	1	0.602

Table 1 displays term frequency statistics need to be computed in order to represent the bug report and source files from the example as vectors. The set containing every term from all the documents is called a *corpus*. The rows from Table 1 represent each term from the corpus. The columns under *Term frequency* count the number of occurrences of each term on each document (the query and the source files). The column *Document frequency* counts the number of documents where the term occurs. This value is applied to Equation 2 and displayed in the column *Inverse document frequency*. Then, applying Equation 1 to the bug report and the three files yields the vectors below:

$$\vec{b} = (0.301, 0.125, 0.301, 0.0, 0.0)$$

$$\overrightarrow{f_1} = (0.0, 0.125, 0.0, 0.0, 0.0)$$

$$\overrightarrow{f_2} = (0.0, 0.125, 0.0, 0.602, 0.0)$$

$$\overrightarrow{f_3} = (0.301, 0.0, 0.301, 0.0, 0.602)$$

Thus, from Equation 3, the similarities between the bug report and files 1 through 3 are, respectively, 0.282, 0.057, and 0.554. Therefore, file #3 would be the one more similar to the given bug report.

2.2.3 Effectiveness metrics

Information retrieval results are usually presented as a list of documents sorted by relevance (similarity) to the query. It is often enough to present a small set of documents that a user can browse to locate the needed information. For this reason, the *effectiveness* of IR models is commonly measured by the ability of the models to retrieve relevant documents at the first positions of a list. A common effectiveness metric is called *precision at k* [13], also referred to as *Top N* [3] [4] [15] or *Hit@N* [5]. Given a set of queries, the precision at *k* is the percentage of the

queries where the model was able to retrieve a relevant document ranked on the first *k* positions. Common practice in bug localization studies is to consider up to the 10 first positions of the resulting list [3] [4] [5] [15].

Another widely used metric for the effectiveness of IR models is the *mean average precision (MAP)*. MAP provides a single-figure measure of the quality of information retrieval when a query may have multiple relevant documents [13]. The MAP for a set of queries is the arithmetic mean of the *average precision (AP)* of individual queries [13]. Let $\{d_1, \dots, d_m\}$ be the set of the *m* documents that are relevant to a query, and R_k the set of ranked results from the top result until one reaches document d_k. Then, the average precision of a query *q* is given by:

$$AP(q) = \frac{1}{m}\sum_{k=1}^{m} Precision(R_k)$$

Equation 4 – Average precision for an information retrieval query

$$Precision(R) = \#(relevant\ items\ retrieved)/\#(retrieved\ items)$$

Equation 5 – Precision of a retrieval

For example, consider a query where the three relevant documents are ranked in 1st, 4th, and 10th positions. Then, precision for each document would be $1/1 = 1.0$, $2/4 = 0.5$, and $3/10 = 0.3$, yielding an average precision of 0.6. Note that a perfect average precision score (1.0) would only be obtained if the *m* relevant documents were ranked on the *m* first positions. If only the *k* first positions are being evaluated, it may be possible that a query does not retrieve a relevant document at all. For example, if we are interested only in the 10 first results, but the first relevant file is ranked in the 11th position, the precision of the query is considered zero [13].

2.3 Information retrieval-based bug localization techniques

Bug localization techniques built around information retrieval (IR) models have been available for a while. We present a brief summary of IR-based bug localization studies conducted in recent years.

In 2011, Rao and Kak [12] compared five information retrieval models for bug localization: vector space model (VSM), latent semantic analysis model (LSA),

unigram model (UM), latent Dirichlet allocation model (LDA) and cluster-based document model (CBDM). Their evaluation used iBUGS [36], a benchmarked dataset created to evaluate automated bug detection and localization tools [37], mainly composed of AspectJ bugs [36]. The authors show that IR-based bug localization techniques were at least as effective as other static and dynamic bug localization techniques developed until then. They also conclude that sophisticated models like LDA, LSA and CBDM do not outperform simpler models like Unigram or VSM for IR-based bug localization on large software systems.

Sisman and Kak [7] incorporated history information into retrieval models to improve bug localization accuracy. They proposed two base models to determine, from version history, the prior defect and modification probabilities associated with the files in a software project. These models were also extended to incorporate a time decay factor. This strategy reflects the expectation that if a file had been modified in the past but has not been modified recently the modification probability should decrease. Similarly, bug fixes that are more recent should have a stronger influence in estimation of prior defect probabilities. Each probability estimation model was incorporated into six baseline retrieval models [7]. All of the four probability models improved retrieval performance when compared with the baseline results of the retrieval models used in isolation. Evaluation showed that the defect history based model performed consistently better than the modification history based model. The inclusion of the time decay factor also improved results for both modification and defect history models. Sisman and Kak also compared their approach to other tools, namely Ample [37], FindBugs [38], and BugScout (Nguyen et al. [39]), obtaining improvements over all of them.

Two key findings from these studies can be highlighted, as they influenced upcoming work on IR-based bug localization. It was shown that (i) VSM provides a simple and effective basis for IR-based bug localization techniques [12] and (ii) incorporating additional information to bug localization techniques could improve their effectiveness [7]. Following these studies, a sequence of IR-based bug localization techniques contributed to the state-of-the-art by incorporating distinct information into their models. Next subsections present these techniques.

2.3.1 BugLocator

Zhou et al. proposed a bug localization method called BugLocator [3], which ranks files based on the textual similarity between a bug report and the source code using a revised Vector Space Model (rVSM). The authors modify classic VSM in order to improve the ranking of large documents. The rationale is that larger source files tend to have higher probability of containing a bug [3]. Therefore, these files should be ranked higher in the context of bug localization [3].

BugLocator also considers information about similar bugs that have been fixed before. It assumes that, in order to fix similar bugs, developers tend to modify the same files. The relevance of a file *f* to a bug *B*, based on the involvement of *f* in previous similar bugs, denoted by the authors as *SimiScore*, is given by the following equation:

$$SimiScore(f) = \sum_{s \in S_f} sim(B, s)/n(s)$$

Equation 6 – BugLocator's score due to similarity to previous bugs

In Equation 6, S_f is the set of previous bugs related to *f*, i.e., bugs where *f* was one of the files modified in order to fix them. The similarity between bug *B* and a bug *s* that is related to *f* is denoted as $sim(B, s)$, and is calculated using Equation 3. Finally, $n(s)$ is the number of files modified in order to fix *s*, determined using heuristics presented in [40]. The final score of a file is then combined with its own similarity to the bug being located (*rVSMScore*), as follows:

$$FinalScore(f) = (1 - \alpha) \times rVSMScore(f) + \alpha \times SimScore(f)$$

Equation 7 – BugLocator's final score

In Equation 7, α is a weighting factor, valued between 0 and 1. The best results reported by Zhou et al. were with α between 0.2 and 0.3. BugLocator was used to find more than 3,000 bugs in four open source projects: Eclipse, SWT (open source widget toolkit for Java), AspectJ, and ZXing (barcode image processing library for Android applications). BugLocator was compared with other bug localization techniques, namely VSM [12], LDA [22], LSI [41] [42] and SUM [12], and outperformed all of them. BugLocator was approximately 10% more effective than the second best performing technique, i.e., SUM [3].

BugLocator's importance to IR-based bug localization is paramount. It advanced the state-of-the-art substantially by outperforming various previous approaches [12] [22] [41] [42]. Thus, it served as a baseline for evaluation of future techniques, namely BLUiR [4] and AmaLgam [5], described in the next subsections.

2.3.2 BLUiR and BLUiR+

Saha et al. [4] developed BLUiR (Bug Localization Using information Retrieval), an automatic bug localization technique based on the concept of structured information retrieval. In structured IR, fields from a bug report and program constructs, such as class or method names, are separately modeled as distinct documents. Consequently, bug reports and source files are not counted as single documents, as they do in BugLocator. Instead, BLUiR breaks source files into four parts: class names, method names, variable names and comments. Bug reports are split in two parts: summary and description. BLUiR then calculates the similarity between each file part and bug part separately, summing the eight individual similarities in the end. This implicitly assigns greater weight to terms that appear in multiple parts. The formula below represents the core of the BLUiR approach.

$$sim(f, b) = \sum_{fp \in f} \sum_{bp \in b} sim(fp, bp)$$

Equation 8 – Structural similarity between a file and a bug report in BLUiR

In Equation 8, *f* and *b* are a source file and a bug report, and *fp* and *bp* are its respective parts. The similarity between a bug report *b* and a source file *f* is given by the sum of the similarities of their parts, calculated according to Equation 3.

Saha et al. performed their evaluation on the same projects used by BugLocator [3]. In addition to structured IR, Saha et al. investigated other variables in order to assemble their own information retrieval model. For example, they compared two different stemmers, Porter and Krovetz, without finding any significant difference in effectiveness. Another investigated variable regards identifier splitting. Recall from Section 2.2.1 that an identifier named `MemoryMappedFile` would be split as "memory", "mapped" and "file". BLUiR

modifies this step by also including the full identifier name ("memorymappedfile") to the list of terms, based on the observation that many bug reports mention code elements. This modification increased bug localization effectiveness by up to 30% in terms of MAP (0.20 to 0.26 for the Eclipse project [4]).

However, the key insight of BLUiR is, indeed, structured information retrieval. Comparison of BLUiR results with and without modeling source code structure revealed improvements for all evaluated projects when source code structure is considered. Hit@1, i.e., bugs where a related file was ranked in the first position, increased almost 46% (37 to 54) in the SWT project. As for MAP, it increased up to 23% in the Eclipse project (0.26 to 0.32). Comparison to the previous best performing technique, BugLocator, was also favorable. In the AspectJ project, BLUiR was able to increase MAP by 41% (0.17 to 0.24). Note this improvement refers to BugLocator with no similar bug information (i.e., $\alpha = 0$).

A variant of BLUiR, called BLUiR+ [4], also leverages information from previous similar bug reports, if available, similarly to BugLocator. The authors compared both variations, BLUiR and BLUiR+, to BugLocator with bug similarity data. The comparison between BLUiR+ and BugLocator with bug similarity data showed a performance improvement in terms of MAP of up to 28% (0.45 to 0.58 in project SWT). Another interesting result is that BLUiR achieved results similar or superior to those of BugLocator even when the latter did use bug similarity data and the former did not. This finding indicates that the structured IR approach used by BLUiR could compensate for the lack of previous bug report information.

2.3.3 AmaLgam

Another example of bug localization technique that combines structured IR with other sources of information is AmaLgam [5]. AmaLgam is a technique for locating buggy files that combines the analysis of: (i) version history, (ii) bug report similarity, and (iii) structure of documents, i.e., bug reports and source code files. AmaLgam has three components that produce suspiciousness scores for each source file. The suspiciousness score represents how likely a source file is of containing the searched bug. Each component uses a different source of information. Individual scores are then combined by a fourth component (composer) into a single score for each source file. AmaLgam components are described next.

Version history component. Change history has been previously used to predict which files are likely to contain defects in the future [6] [43]. Based on these studies, Wang and Lo included change history into AmaLgam's model through a version history component. This component consists of Google's adaptation [8] to the algorithm from Rahman et al. [6], described by the following equation.

$$scoreH(f,k,R) = \sum_{c \in R \wedge f \in c} \frac{1}{1 + e^{12(1-((k-t_c)/k))}}$$

Equation 9 – Version history component

In Equation 9, *R* refers to the set of relevant commits, i.e., commits associated to the resolution of a bug [5]. t_c is the number of days elapsed between a commit *c* and the creation of the bug report. Parameter *k* was included by Wang and Lo to restrict the version history period (in days) to be considered. It was observed by the authors that considering only more recent commits provided a good trade-off between precision and performance. The optimal value found by the authors was $k = 15$ [5].

Report similarity and Structure components. AmaLgam's report similarity component is based on the SimiScore formula (Equation 6) from BugLocator [3], also used by BLUiR+ [4]. The component considers the textual similarity between bug reports and the number of files modified to fix each bug report. The assumption is that, in order to fix similar bugs, developers tend to modify the same files. AmaLgam's structure component uses the same approach as BLUiR (Section 2.3.2, [4]). It breaks bug reports and source files into smaller documents, composed of summary and description (bug reports), and class names, method names, variable names and comments (source files).

Composer component. The composer component takes the scores produced by the three other components and combines them into a final suspiciousness score. It first combines the results from the report similarity ($scoreR$) and structure ($scoreS$) components. This result is then combined with the score from version history component ($scoreH$), according to the following equations:

$$scoreSR(f) = (1-a) \times scoreS(f) + a \times scoreR(f)$$

Equation 10 – Score combining structural and older bug report similarity scores

$$scoreSRH(f) = \begin{cases} (1-b) \times scoreSR(f) + b \times scoreH(f), & scoreSR > 0 \\ 0, & otherwise \end{cases}$$

Equation 11 – Final score attributed to a file by AmaLgam

Parameters *a* and *b* in the previous equations determine the weight of the contribution of each component to the final suspiciousness score. Based in their own experiments and in results from [3] and [4], the authors adopted the default values of 0.2 for parameter *a* and 0.3 for parameter *b* [5]. These parameter values are equivalent to attributing weights of 30% for version history, 14% for bug report similarity and 56% for structured IR from source files.

2.3.4 Discussion of the techniques

BugLocator [3], BLUiR [4], and AmaLgam [5] form a successful sequence of bug localization techniques where each work contributed with a new insight. BugLocator is based on textual similarity between an input bug report and (i) source files and (ii) older bug reports. BLUiR extends BugLocator by considering source file structure. AmaLgam aggregates version history, which has been previously used in isolation [7], but not combined with information retrieval techniques.

BLUiR results highlighted the impact that structured IR brought upon bug localization effectiveness. This is illustrated by the fact that even the less effective variation of BLUiR was still able to outperform the best configuration of BugLocator (Section 2.3.1). AmaLgam demonstrated that combining analyses of additional sources of information could improve bug localization effectiveness even further. However, even in that case, the optimal parameters found experimentally for AmaLgam pointed to a stronger contribution of structured IR.

All the mentioned techniques [3] [4] [5] were evaluated on the same set of four Java projects, namely, AspectJ, Eclipse, SWT, and ZXing. This strategy allowed the authors of the previous studies to draw a direct comparison of the techniques, highlighting the effectiveness improvement obtained with each technique. On the other hand, it is desirable to replicate or extend experiments using different datasets. Given the growing relevance of structured IR to bug localization techniques, it is of paramount importance to start testing this approach on a higher number of projects, encompassing different domains (Section 3.3).

In addition, a better understanding of structured IR applied to bug localization is needed. Object-oriented programming languages, such as C#, slightly differ from Java in terms of key programming constructs, such as properties. Moreover, BLUiR and AmaLgam did not consider certain programming constructs of Java, such as interfaces and packages, which are relevant to C# programs as well. It remains unaddressed what would be the contribution of such program constructs, ignored by such state-of-art models, on the localization of bugs in slightly different programming languages, such as C#. Consequently, software developers remain uninformed if the amount of constructs in a programming language influence the effectiveness of the technique. If so, we need to investigate whether structured IR-based techniques are suitable for more expressive languages (i.e., with more constructs available).

2.4 Conclusion

We can draw some conclusions from the aforementioned studies [3] [4] [5]. First, structured information retrieval, by leveraging the known structure of the documents involved in the retrieval process, has been more effective than traditional information retrieval. Second, hybrid approaches can improve the effectiveness of bug localization even further by combining different sources of information with different weights. However, even with hybrid approaches, the contribution of structured IR to the effectiveness of the approach is still prominent, as the comparison of BLUiR and AmaLgam to BugLocator demonstrates.

The potential of structured IR motivates us to investigate this approach to bug localization further, in the context of other programming languages, such as C#. Therefore, we selected some of the best performing techniques based on structured IR, namely, BLUiR, BLUiR+, and AmaLgam, for our study. Our goal is to verify the effectiveness of these techniques on another object-oriented programming language, in order to assess if the change of language itself could cause a significant impact on the techniques' effectiveness. We also plan to verify if the effectiveness of these techniques could be improved by using a different set of program constructs than that used in previous studies [4] [5]. Next chapter formalizes our research questions and describes the experiment conducted to answer them.

3
Evaluation of bug localization techniques

Bug fixing is a routine, however complex, activity. Determining which parts of the source code need to be modified to remove a bug can be a difficult task, as it requires the inspection of a large amount of files. Automated bug localization techniques aim to help developers in this task by providing a list of suspicious files potentially related to the bug, thus narrowing the search space where the developer must look for the bug.

The usage of source file structure has been the main responsible for increasing the effectiveness of state-of-the-art, information retrieval-based bug localization techniques. These techniques, namely, BLUiR [4], BLUiR+ [4], and AmaLgam [5], were evaluated in four Java projects. However, the evaluations performed on these techniques contain shortcomings that might have significantly biased the reported effectiveness (Section 1.2.1). These shortcomings suggest their effectiveness may be lower than reported (Section 1.2.2). In spite of these problems, results from BLUiR [4], BLUiR+ [4], and AmaLgam [5] mention improvements of up to 41%, 29%, and 32%, respectively (Sections 2.3.2 and 2.3.3). The evolution brought by these techniques prompts us to explore the potential of structured information retrieval.

In this chapter, we explore structured information retrieval aspects not investigated in previous studies. In particular, we investigate how the usage of different sets of program constructs influences the effectiveness of bug localization (Section 1.2.3). For such, we evaluate BLUiR [4] and AmaLgam [5] on 20 C# projects. C# is a popular language [18] [19], similar to Java, although with significant differences, especially regarding the available constructs. The similarity will allow us to draw a parallel with Java results. At the same time, the differences will allow us to explore constructs inexistent in Java, such as properties and structures.

We also discuss dataset preparation steps conducted to mitigate shortcomings from previous studies (Section 3.4). These preparation steps include selection of

appropriate project versions, removal of bug reports that could influence the evaluation, and removal of test files from the search scope. Results show that, with the appropriate data preparation steps, effectiveness of bug localization is at least 34% lower, compared to the effectiveness without the data preparation steps (Section 3.6.1).

After evaluating the techniques as they were conceived, we adapt them in order to assess their sensitivity to the consideration of more constructs (Section 3.5). We define three construct mapping modes, which represent different forms of splitting source files, namely, *Default*, *Complete*, and *Mixed* modes. The *Default* mode corresponds to the same mapping used by BLUiR [4], BLUiR+ [4], and AmaLgam [5], where source files are splitted into four documents, consisting of class names, method names, variable names, and comments. In the *Complete* mode, all the available constructs are considered separately and every source file is splitted in 12 parts, each one corresponding to each available C# construct. Finally, the *Mixed* mode maps all C# constructs into four groups, similarly to the original mapping used by BLUiR and AmaLgam. The *Mixed* and the *Complete* construct mapping modes were able to increase bug localization effectiveness by 8% and 18%, on average (Section 3.6.2).

3.1 Goal and research questions

Recall our stated goal from Section 1.3:

Perform a realistic, in-depth effectiveness evaluation
of state-of-the-art bug localization techniques
on a previously untested programming language.

In order to perform a realistic evaluation of bug localization techniques based on structured IR, we need to evaluate them on different scenarios. An initial step in that direction is to understand the behavior of prominent bug localization techniques applied to an object-oriented programming language that is slightly different from Java. Java has been the focus of previous studies of structured IR-based techniques for bug localization (Section 2.3). Nonetheless, software engineers remain unaware to what extent they can rely on these techniques to perform bug localization activities in projects structured with other programming languages. We have

selected C# as it is a general-purpose, object-oriented language that shares many traits with the Java language, but also have some distinct programming features. For example, some widely used C# constructs, like properties and structures (“structs”), are inexistent in Java. Moreover, C# is a popular language [18] that figures within the top 10 languages in number of GitHub repositories [19]. To the best of our knowledge, neither of these techniques have been previously evaluated on other object-oriented programming languages, such as C# (Section 1.2.1).

We unfold our general goal in the following research questions:

RQ1: Are BLUiR, BLUiR+, and AmaLgam effective to locate bugs in C# projects?

The effectiveness of current structured IR techniques, i.e., BLUiR, BLUiR+, and AmaLgam, have been assessed and confirmed only for Java projects. However, developers using many other languages could also benefit from such techniques. In order to address this gap, we ran the selected techniques in their best performing configurations (Section 2.3) on a set of C# projects. The results enabled us to address RQ1 by assessing the effectiveness of these techniques on a previously untested programming language.

RQ2: Does the addition of more program constructs increase the effectiveness of bug localization on C# projects?

In order to understand the potential of structured IR techniques completely, we need to analyze their sensibility to particular constructs of a programming language. Therefore, we addressed RQ2 by focusing this analysis on program constructs that were also not considered in previous studies [4] [5], such as string literals, interfaces, and enumerations. In addition, there are language features from C# that do not exist in Java, such as structures and properties. The effects of their explicit consideration on state-of-the-art bug localization are not well understood. Thus, we investigated to what extent the effectiveness of a structured IR technique would benefit from the explicit consideration of these source code constructs.

3.2 Evaluation metrics

This section describes the metrics used to assess the effectiveness of the techniques on selected projects. We focused on the use of two sets of metrics typically used in recent studies [3] [4] [5] and presented in Section 2.2.3:

Hit@N: Percentage of bug reports that have at least one buggy file ranked by the technique in the top N positions. Typical values for N are 1, 5, and 10 [3] [4] [5].

Mean average precision (MAP): Mean of the average precision scores (Equation 4) across all queries. Considers the ranks of all the buggy files, not only the first one.

To measure the effectiveness of a technique, or one of its variations, the average of the results for each project is taken. Finally, the effectiveness of a technique, or one of its variations, corresponds to the average of the results for each project. We use commit and bug report data obtained from the selected projects (Section 3.3) as the oracle against which we compare the results of our implementation. When a bug report explicitly contains a link to a commit, we consider the files modified in the commit as the ones that solved the bug. This is a common assumption in many bug localization studies [3] [4] [5]. When there is no explicit link between a bug report and a commit, we use conventional heuristics [40] to infer this relationship. These heuristics consist of looking for commits that contain messages such as "*Fixes issue 97*" or "*Closes #125*", which usually denotes the ID number of the associated bug report. All these procedures were also important to implement in our study given the lack of C# datasets, which differs from the state-of-the-art on empirical studies of Java projects.

3.3 Project selection

For our experiment, we needed a number of C# projects with available information on their source code, commits, and bug reports. We could not find a bug dataset for C# projects, like iBUGS [36] or moreBugs [44]. Then, we used GitHub search functionality[5] to obtain a list of large C# projects, by searching for

[5] Integrated development environment.

projects with 1,000 or more stars and 100 or more forks. These parameters indirectly allowed us to satisfy the requirement for large projects. The query returned almost 80 projects from various domains, including development tools, compilers, frameworks, and games.

We used GitHub API to download commit and issue data from the projects. We downloaded the 1,000 most recent issues for each project[6], and then all the commits that happened within the period covered by the issues. Next, we processed the data in order to identify (i) issues that could be characterized as bugs and (ii) files modified in order to fix the bug. For characterizing a GitHub issue as a bug, we relied on the labels applied by the users. Issues with at least one label containing terms such as “bug” or “defect” were considered a bug report. As for the files modified to fix a bug, they are determined by the associated commit, as explained in Section 3.2. Since we are focusing on C# code, we excluded from evaluation bugs that do not touch at least one C# file.

After processing downloaded data, only those projects where we could find at least 10 bugs whose resolution modified at least one C# file were kept for the experiment. We processed the projects in the order returned by the query, until we reached 20 projects that met our selection criteria. Table 2 presents a comparison between the dataset of C# projects used in our study and the dataset of Java projects used in recent studies [3] [4] [5] of the same techniques.

Table 2 – Dataset comparison

Dataset details	**Java**	**C#**
Projects	4	20
Files	20,223	46,752
Source files	N/A	28,596
Issues	N/A	16,630
Traceable to commits	N/A	2,839
Classified as bugs	3,479	878

In the table above, we highlight various differences, including the differences between the amount of files and amount of source files present in the corresponding repositories. In our case, about 61% of the files contained in the repositories were C# files, the ones we actually used to search for bugs. This happens because many files represent: (i) configuration or HTML files, or (ii) source files structured with other programming languages, in the case of multi-language projects. Actually, the existence of multi-language projects also highlights the importance of evaluating

[6] GitHub API limits issue searching to 1,000 results per query.

bug localization techniques in different programming languages. For the Java dataset, it was not clear whether the total referred only to Java source files or to all repository files. Therefore, we assumed the latter.

As for the bugs, all of them are treated as issues in GitHub issue tracker, although not all issues are bugs. After downloading all the available issues, we associated them with a commit whenever possible, using the criteria explained in Section 3.2. This step reduced the amount of available bugs to 17% of the original issue count. Then we discarded issues that were not labeled as “bug”, which reduced the number of available bug reports even further, down to 878 (5% of the initial number of issues).

In the next section, we discuss additional preparation steps we applied on the dataset, which were important to guarantee the construct validity of the experiment.

3.4 Dataset preparation

As mentioned in Section 1.2, previous studies suffer from a series of shortcomings regarding their experimental setup. Next, we describe how we handled these shortcomings in our evaluation.

3.4.1 Version selection

Previous studies on bug localization commonly selected only a single release and ran the bug localization for all bugs on the same release. Results reported in this manner cannot be fully trusted [17], because there is a high chance that the bug is not even present on the code being analyzed. To overcome this problem, we identified the version of the source code that was active by the time the bug was reported by searching for the oldest commit that happened before the bug report creation. The source code for every identified version was downloaded, and each bug was localized on its corresponding version.

3.4.2
Bug report selection

Some bug reports already inform the location of the defect in source code, by mentioning the file where the bug was observed. Kochhar et al. [14] demonstrated that including these bug reports on the evaluation of a bug localization technique significantly influences the results by artificially increasing the reported effectiveness. The authors classified bug reports in three categories: *fully localized*, *partially localized*, and *not localized*, which mean that a bug report mentions *all*, *some* or *none* of the files modified to fix a bug; respectively. We removed fully and partially localized bug reports from our evaluation, meaning that we included only those bug reports that contained no mention to the faulty files. Although this step contributes to more realistic results, it reduced the number of available bug reports in 51%, from the 878 reported in Table 2 to 450 (3% of the initial issue count).

3.4.3
Source file selection

Software projects often include test code. Test code may contain bugs, which, in theory, may be reported just like production code. However, bug localization algorithms should not include test code within their scope. Consider, for instance, three bug reports, whose resolution involved the modification of (i) only production code (no test code); (ii) production and test code; and (iii) only test code. In the first case, it is obvious that localization does not benefit from considering test code. When the resolution of a bug requires changing production and test code (second case), it is usually because a test was added or modified in order to catch the referred bug in the future. Test code was not the *source* of the failure, though. Therefore, modified test files are not what developers expect as an answer from the localization algorithm in this case. Finally, when a bug in the test code itself is caught (third case), developers already have detailed information provided by the test framework, which includes the location of the bug. Thus, even if a developer chooses to report a test bug instead of fixing it immediately, it is likely that this report will include the detailed information already provided by the test framework. Therefore, bug reports on test code are rarer (because the developer may choose to fix the bug instead of reporting it) and likely to be localized (because test frameworks already

indicate the buggy files). This rationale led us to restrict the localization to production code.

We excluded test files from the scope of the analysis by ignoring all files that contain the word “test” on its path. We confirmed with manual inspection on two sample projects that this simple heuristic was able to accurately remove the undesired files, since it reflects (in our sample) the common developer practices of naming test files with a “Test” prefix of suffix, or placing test code in a separate directory named “test”.

3.5 Model adaptation

Structured IR demands the extraction of identifiers from source code. For this task, we used the .NET Compiler Platform, also known as Roslyn [45]. As C# is an object-oriented language, similar to Java, it has the same four constructs considered on BLUiR’s original evaluation: class names, method names, variable names, and comments. However, C# also has constructs that either were not considered by BLUiR (and, consequently, neither by BLUiR+ nor AmaLgam) or do not exist in Java. Table 3 summarizes these differences.

Table 3 – List of C# constructs

C# construct	Equivalent in Java?	Considered by BLUiR?
Classes	Yes	Yes
Comments	Yes	Yes
Enumerations	Yes	No
Fields	Yes	No
Interfaces	Yes	No
Methods	Yes	Yes
Namespaces	Yes (packages)	No
Parameters	Yes	No
Properties	No	No
String literals	Yes	No
Structures	No	No
Variables	Yes	Yes

BLUiR breaks a source file into parts. Each part contains identifiers from one kind of construct. To deal with the different kinds of constructs, while keeping the underlying philosophy of BLUiR, we devised three alternative configuration modes to run the experiment:

- *Default*: Strictly uses only the same constructs used by BLUiR, ignoring any other construct.
- *Complete*: Uses all constructs present in Table 3, with each construct mapped to an exclusive file part.
- *Mixed*: All constructs are used, but they are mapped to one of the four file parts corresponding to the constructs originally used by BLUiR.

Default mode is used as a baseline for the sake of comparing our results with the original evaluation in Java projects [4]. *Complete* mode represents the simplest way of including new constructs in BLUiR's algorithm. *Mixed* mode represents an alternate way of computing new constructs, by mapping them to one of the preexisting categories. For example, interfaces, structures, and enumerations are semantically close to classes. Therefore, for the purpose of bug localization, it could be enough to consider code elements of any of these types as "classes". In a similar vein, string literals usually represent plain text inserted into source files, such as comments do. Therefore, string literals and comments could be mapped together in the same file part.

The difference between some constructs is negligible in practice. For instance, variables and parameters are distinct constructs, strictly speaking. However, from the developers' point of view, they are both handled as variables. Although it was not clear, this simplification might have been used in the BLUiR evaluation. Thus, the *Mixed* mode addresses this possible ambiguity, by defining a broader interpretation to the four constructs mentioned in [4]. The mapping strategy for each mode is shown on Table 4.

Table 4 – Construct-mapping strategies

Mode	**Construct mapping**
Default	Classes, Methods, Variables, and Comments (one file part for each)
Complete	All constructs from Table 3 (one file part for each)
Mixed	Part 1: Classes, Enumerations, Interfaces, Namespaces, and Structures Part 2: Methods Part 3: Fields, Parameters, Properties, and Variables Part 4: Comments and String literals

Next section presents the results of the evaluation, thus answering the research questions formulated in Section 3.1. For RQ1, *Default* mode will be applied to the 20 selected projects (Section 3.6.1). In RQ2, *Complete* and *Mixed* modes will also be applied to the same projects in order to compare their results with those from *Default* mode (Section 3.6.2).

3.6 Results

3.6.1 Effectiveness of structured IR-based bug localization in C# projects

Our goal includes performing *"a realistic (…) evaluation of state-of-the-art bug localization techniques on a previously untested programming language"* (Section 3.1). To reach this goal, we apply BLUiR, BLUiR+, and AmaLgam on 20 C# projects. To the best of our knowledge, the mentioned techniques have not been evaluated on C# projects so far. To perform a realistic evaluation, we include the dataset preparation steps discussed in Section 3.4. However, we also apply the techniques without dataset preparation, in order to draw a parallel with results from Java studies. The two sets of results will be presented next: first *without* dataset preparation (Section 3.6.1.1) and then *with* dataset preparation (Section 3.6.1.2).

3.6.1.1 Effectiveness without dataset preparation

Initially, we questioned whether current state-of-the-art bug localization techniques based on structured information retrieval, i.e., BLUiR, BLUiR+, and AmaLgam, would effectively locate bugs in C# projects. Considering results for Java, one would expect similar levels of effectiveness for C# as well, given the apparent similarities between the languages. To answer our first research question, we ran the bug localization techniques on downloaded projects using the reported optimal configuration for each technique (Section 2.3) and the *Default* mode (Table 4). For the sake of comparison, we initially run the algorithms without the preparation steps discussed in Section 3.4, i.e., selection of appropriate source code versions, exclusion of localized bug reports, and exclusion of test files. For each technique, we took the average MAP, which consists of the arithmetic mean of the MAPs from each project. Table 5 presents the average MAP values observed for the set of evaluated projects, and the variation observed over the same measure from Java projects. Considering all the techniques, the average MAP achieved by each technique with C# projects was around 0.307.

Table 5 – C# and Java results – Average MAP

Technique	Java	C#	Variation
BLUiR	0.38	0.302	-21%
BLUiR+	0.39	0.307	-21%
AmaLgam	0.43	0.312	-27%

Opposed to the previous findings in Java projects, the selected bug localization techniques showed lower effectiveness in terms of average MAP. This result should be interpreted carefully, as the projects are different and cannot be compared. Nevertheless, the observed variation is explainable in part due to the higher number of projects analyzed: 4 in the Java studies [4] [5] against 20 in our C# study. Within projects in the same language, the techniques presented similar behavior: AmaLgam performed better than BLUiR+, which outperformed BLUiR. Recall from Section 2.3 that each technique uses a superset of the information used by the previously proposed technique: BLUiR is based on the similarity of bug reports and source code, BLUiR+ adds the similarity of previous bug reports to the equation, while AmaLgam also considers version history. At first glance, this could be considered an indication that, in fact, hybrid techniques which combine additional sources of information tend to perform better than their predecessors do. However, the average values are rather close. Thus, we analyzed additional parameters, which are presented in Table 6.

Table 6 – C# and Java results – Minimum and maximum project MAP

Technique	Minimum MAP		Maximum MAP	
	Java	C#	Java	C#
BLUiR	0.24	0.103	0.56	0.596
BLUiR+	0.25	0.125	0.58	0.596
AmaLgam	0.33	0.120	0.62	0.604

In contrast with data available from Java studies, we observed a high variation on results from C# projects. Table 6 presents highest and lowest MAP scores for each technique and language. The minimum MAPs from the C# group were lower than minimum values from the Java group for all three techniques. The maximum MAPs, on the other hand, were similar. In fact, there was one project where the techniques reached even higher MAP values, but it was considered an outlier, as seen on Figure 1. Average MAPs for the outlier were 0.770, 0.767, and 0.747 for BLUiR, BLUiR+, and AmaLgam, respectively.

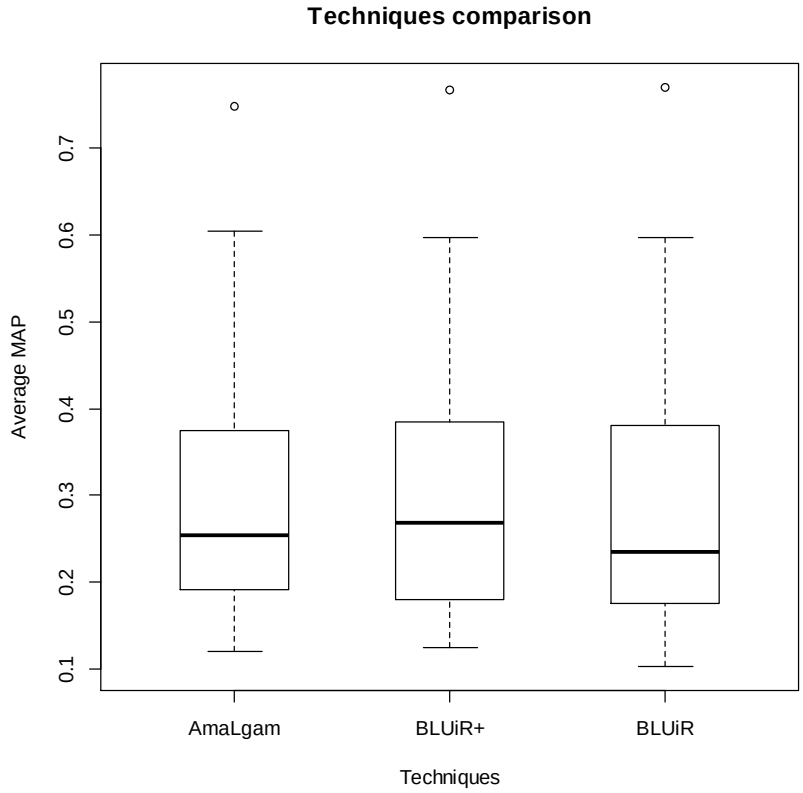

Figure 1 – Effectiveness of the techniques with C# projects

The three techniques performed very similarly on the C# projects, as the averages on Table 5 indicate. Another evidence of the similar performance is the fact that each technique attained a higher score with a different metric: AmaLgam had the higher average (0.312), BLUiR+, the higher median (0.269), and BLUiR, the higher maximum (0.770). This suggests that, for this particular dataset, the additional information considered by BLUiR+ and AmaLgam failed to increase the effectiveness of the techniques significantly. Nevertheless, in spite of the lower averages relative to the Java evaluation, six C# projects still have attained MAP scores superior to the average of their Java counterparts in at least one technique. This implies that, in principle, there is no impediment to the usage of bug localization on C# projects due to features of the language itself, leaving room for the investigation of alternatives to increase the effectiveness of the techniques. In Section 3.6.2 we propose such alternatives by evaluating the effects of different mappings of language constructs on the bug localization algorithm. However, we must evaluate the effectiveness of these alternatives against an accurate baseline. Hence, we also performed an evaluation of the techniques with the dataset preparation steps discussed in Section 3.4, to be presented next.

3.6.1.2 Effectiveness with dataset preparation

We remind the reader that results from the previous section were obtained without considering the dataset preparation steps presented in Section 3.4. Those results were compiled to reproduce the same conditions from the original studies [4] [5]. A more realistic experiment, however, should incorporate these steps. Thus, we have also performed an evaluation of the three techniques including these steps. Table 7 presents minimum, maximum, and average values for each technique, and the decrease relative to results with no preparation steps.

Table 7 – Effect of dataset preparation steps on bug localization – MAP

Technique	Min	Average	Max
BLUiR	0.020	0.183 (-40%)	0.499
BLUiR+	0.048	0.198 (-36%)	0.493
AmaLgam	0.044	0.206 (-34%)	0.565

Wilcoxon Signed-Rank tests with 95% confidence level confirmed that additional preparation steps on the dataset significantly decreased the MAP scores for the three techniques. Complete details about the tests, including p-values, are available at the study website [46]. The maximum values indicate that some projects were still able to achieve reasonable scores. However, compared to the execution with no preparation steps, the effectiveness for all projects decreased, on average, more than 30% for all the evaluated techniques. Figure 2 presents a graphical comparison of the effectiveness with and without the preparation steps. It becomes clear from the data that bug localization studies must not ignore these steps, under the penalty of reporting results incorrectly higher than what would be found in actual settings.

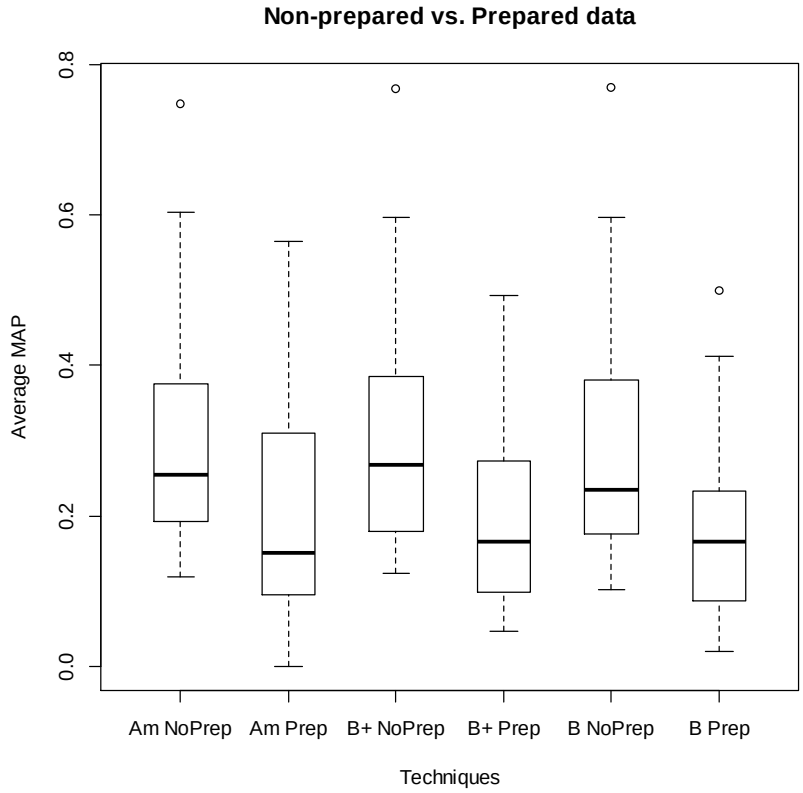

Figure 2 – Effectiveness of techniques with C# projects – Non-prepared vs. prepared data

The importance of removing localized bug reports from this kind of evaluation is not only a matter of construct validity of the experiment. It has indeed practical importance, as non-localized bug reports are exactly the kind of report where developers would need the assistance of a localization technique. Therefore, the effectiveness of IR-based bug localization in terms of mean average precision can still be considered too low for these techniques to be applied in practice.

On the other hand, the expectations for this kind of technique must also be put into context. No matter how effective they are, bug localization techniques do not eliminate the need for the developer to examine and fix the buggy file. Therefore, instead of pinpointing the exact files where the bug is located, it may be acceptable for the technique to provide a list featuring a few candidates. Table 8 shows that the best performing technique – AmaLgam – was able to return a buggy file at the top of the list 20% of the times, and in 57% of the times there was a buggy file among the 10 first files returned by the technique. Analyzing hundreds of files and correctly placing at least one buggy file in a list of 10 candidates for almost 60% of the time, while not ideal, is not as discouraging as the average MAP of 0.206 suggests.

Table 8 – AmaLgam effectiveness with dataset preparation steps

Technique	Hit@1	Hit@5	Hit@10
AmaLgam	20%	46%	57%

Nevertheless, these results reinforce that there is still room for improvement. As discussed in Section 2.3, structured information retrieval is the component that contributes the most to the effectiveness of state-of-the-art bug localization techniques [4] [5]. This remains true even for techniques using multiple sources of information, such as AmaLgam [5]. Thus, we extended the underlying algorithm of AmaLgam's structure component (Section 2.3) in order to assess its effectiveness when using a different set of programming language constructs. We present the results in the next section.

3.6.2 Usage of more constructs to improve bug localization effectiveness

The set of constructs used by BLUiR, BLUiR+, and AmaLgam includes basic constructs from object-orientated languages (classes and methods) and constructs from programming languages in general (variables and comments). However, some subtleties about construct selection were omitted or unaddressed in previous studies. For instance, it is unclear how these techniques deal with interface names, which could be considered equivalent to class names or simply ignored. As for variable names, they might refer only to local variables or include class attributes (or fields) and method parameters. In other words, there are additional types of constructs that could be explicitly considered by bug localization techniques. When considering a different programming language, with a different set of constructs, these questions become more relevant.

To answer whether the consideration of more source code constructs could improve effectiveness of bug localization, we designed the three construct-mapping modes described in Table 4. We selected AmaLgam, the best performing technique according to the evaluation from Section 3.6.1.2, adapted it to use the three mentioned modes, and applied it to the set of C# projects. We present the average MAPs (Table 9) and a box plot (Figure 3) summarizing the performance observed for each mode.

Table 9 – Effectiveness of AmaLgam using different construct-mapping modes – MAP

Mode	Default	Complete	Mixed
Average MAP	0.206	0.244	0.222

The usage of all the 12 constructs associated with the *Complete* mode increased the average MAP of AmaLgam to 0.244, an increase of 18%. *Mixed* mode, which also uses the 12 constructs but maps them into four categories (Table 4), showed a smaller increase on average, to 0.222 (near 8%). From these results, only the improvement associated with *Complete* mode was statistically significant, according to Wilcoxon Signed-Rank tests with 95% confidence level [46]. The effect of the three construct-mapping modes on individual projects was generally the same observed on average values: the higher increase was associated with the *Complete* mode, while *Mixed* mode caused a more modest increase, as shown in Figure 3.

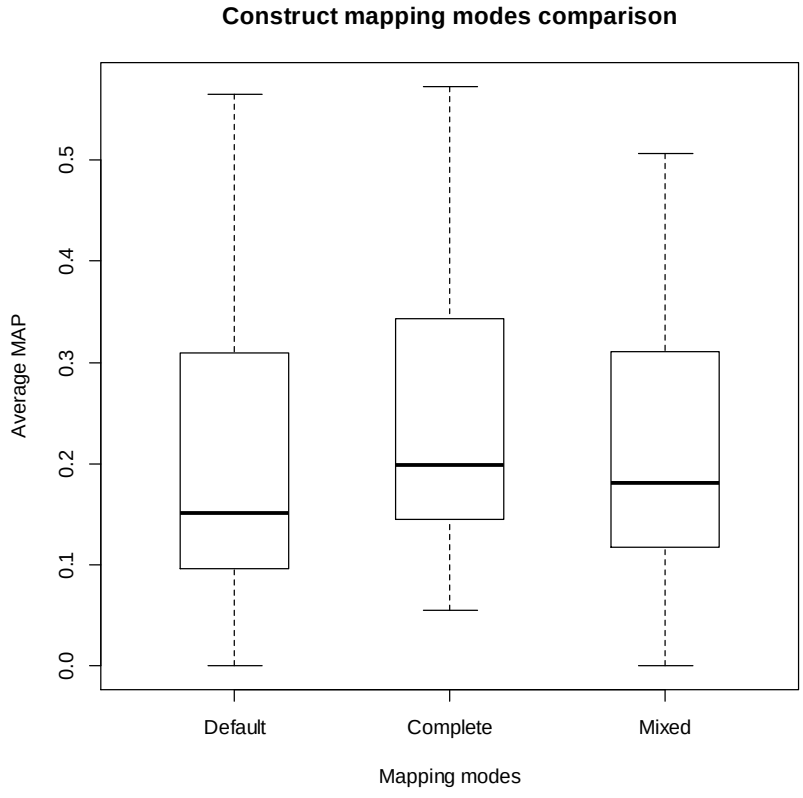

Figure 3 – Effectiveness of construct mapping modes

The reason why *Complete* mode was able to produce better results can be explained by BLUiR formula (also used by AmaLgam) for determining the similarity of a bug report and a source file (Equation 8), which involves the

summation of the similarities of all pairs of bug-file parts. In *Default* mode, the total number of similarities to be summed is 8 – two parts from the bug report multiplied by four parts from the source code files. In *Complete* mode, the number of similarities to be calculated increases to 24 (12 file parts × 2 bug parts). Similarity results are normalized before the rank of files is generated, such that file scores are always between 0 and 1. Therefore, the higher value that would result from the summation of more terms in *Complete* mode is unlikely to be the reason this mode produced better results. In addition, the *Mixed* mode restricts the number of terms to be added up to 8, similarly to the *Default* mode. Since the *Mixed* mode has also produced results higher than those of the *Default* mode, we conclude that the consideration of more source constructs by itself contributed to increasing the bug localization effectiveness.

3.7 Threats to validity

In this evaluation, we carefully handled experimental issues that are recurrent in bug localization studies (Section 1.2.1). These issues are highly relevant, as results from Section 3.6.1.2 demonstrates. We mitigated the threat to construct validity posed by these issues with the dataset preparation steps described in Section 3.4. Nevertheless, some threats to validity are still present. We discuss them in the next subsections.

3.7.1 Construct validity

Given the originality of our study, we could not find an available bug benchmark for C# projects. Instead, we downloaded issues from GitHub and used the existence of a user-applied "*bug*" label as a criterion to identify bugs among those issues. Even following this procedure, we are still subject to misclassified issues, since not all bug reports could be manually verified. However, recent studies suggest that this particular bias does not substantially influence bug localization results [14]. Nevertheless, we make all our dataset available at our study website [46] so that others can refine it and replicate our study. As there was no dataset available for C# projects, one can consider our replication package also as a contribution of our study.

Studies involving retrospective evaluation of bugs should consider the version of the software at the time the bug was found. We addressed this issue by performing the localization on latest version available before the creation of each bug report. Strictly speaking, this does not guarantee that the selected version actually contains the bug reported. However, the selection of a previous version of the code for each bug report is a close approximation. Moreover, this step mitigates a threat ignored in many recent studies on bug localization, including [4] and [5].

Localized bug reports, i.e., reports that mention one or more code elements, significantly influence the evaluation of bug localization techniques [14]. Thus, we excluded such bug reports from the analysis. By performing this exclusion, we remove an important bias that may have led previous studies to unrealistic results.

Finally, the presence of test files also influences bug localization results, since bug reports related to these files are very likely to be localized (Section 3.4.3). We eliminated this bias by excluding test files from the evaluation. The full rationale for performing these exclusions was discussed in Sections 3.4 and 3.6.1.2. The adoption of these dataset preparation steps argue for a strong construct validity in our study.

3.7.2 External Validity

In an attempt to increase generalizability, we attempted to select a higher number of projects compared to previous studies. Given the criteria defined in Section 3.3, we were able to select 20 projects, a considerably higher number of projects compared to other studies on bug localization (5 times more than [3], [4], and [5]). The absence of a standardized bug database, however, greatly reduced the amount of bug reports available for the experiment (Table 2). The relatively low quantity of bug reports and the variation in quantity and quality of bug reports observed on each project are threats to the external validity of our study. However, we consider that bug localization techniques must be assessed under realistic settings, where the amount of available bug reports widely varies from a project to another. No relationship between the number of bug reports of a project and the effectiveness of the technique could be observed. The complete list of evaluated projects and the number of bug reports evaluated for each one is available at the study website [46].

Another threat is the fact that all selected projects were open-source. This kind of project has a characteristic workflow that differs from that found on proprietary projects. Different policies on bug reporting, for example, may significantly influence the results of bug localization techniques. Therefore, results presented in this study are only representative of the workflow typically practiced in open-source projects.

3.8 Conclusion

Structured information retrieval has been successfully applied to the bug localization problem. Techniques based on structured IR have shown to be considerably more effective than other IR-based approaches. However, these techniques are language-specific, as they depend upon the structure of source files. Considering the multi-language nature of most modern software [47], it is important to have effective bug localization models for the different kinds of languages and technologies used in software projects. This study is a step in that direction, where structured information retrieval is evaluated on C# projects for the first time, as far as we know.

The average effectiveness of the evaluated techniques on C# projects was lower than the same metric reported in the original studies on Java. However, some projects have individually yielded results above the average informed by the Java studies. Therefore, we conclude that, in principle, there is no impediment to the usage of bug localization on C# projects due to features of the language itself. The lower average effectiveness compared to previous studies can be attributed to (i) the 5x higher number of projects evaluated, and (ii) the discard of localized bug reports, which artificially increased the effectiveness of bug localization techniques in previous studies. We also demonstrated that using more program constructs, which is a strategy that differs from previous studies [4] [5], increased bug localization effectiveness by 18% on average.

To the best of our knowledge, this is the first bug localization study to implement the experimental steps needed to solve the issue of different versions raised by [17] and the *localized bug report* bias presented in [14]. Besides, we also observed that test files should not be included in the scope of the localization process. These steps are important because they demonstrate that the reported

effectiveness of current state-of-the-art bug localization techniques cannot be achieved in realistic situations. Future studies in bug localization should not skip such steps, as they produce an experimental setup closer to reality and to developers' expectations, increasing the chances of bug localization to become more useful in practice.

In the next chapter, we address the remaining research questions (RQ3, RQ4, and RQ5). These questions will be answered by performing an in-depth evaluation of how bug localization techniques based on structured information retrieval use program constructs.

4
Analysis of the contribution of program constructs to bug localization

Structured information retrieval (IR) has been able to increase the effectiveness of static bug localization techniques [4] [5]. The key feature of structured IR-based techniques refers to how they break up source files, based on constructs available in the adopted programming language. Bug localization techniques based on traditional IR calculate the similarity between a source file and a bug report considering the whole file (Section 2.3.1). Conversely, structured IR-based techniques break source files into multiple parts, one for each construct recognized by the technique (Section 2.3.2). Each of these parts contains only terms that are instances of the corresponding construct in the original source file. Then, instead of calculating similarity using the whole file, the final similarity between a bug report and a source file is the sum of the similarities between each *part* of the source file and the bug report (Equation 8). BLUiR [4] recognizes four Java constructs: class names, method names, variable names, and comments (Section 2.3.2). Thus, it breaks source files into four parts. The same approach is followed by BLUiR+ [4] (Section 2.3.2) and AmaLgam [5] (Section 2.3.3).

However, structured IR has not been thoroughly explored yet. In addition to the limitation of being evaluated only on four projects, the original models of BLUiR [4], BLUiR+ [4], and AmaLgam [5] used only four constructs from the Java language (Section 3.5). Thus, it is unknown whether other constructs, such as interfaces or enumerations, could have influenced bug localization results. This question becomes even more relevant when source files are written in other programming languages, such as C#, which supports constructs inexistent in Java (Table 3).

In this chapter, we investigate the influence of different program constructs on the effectiveness of structured IR-based bug localization. In this investigation, we use results obtained with the *Complete* mode (Section 3.5), as this construct mapping mode increased bug localization effectiveness by including all available

C# constructs into the localization process (Section 3.6.2). Then, we use a statistical procedure called Principal Component Analysis (PCA) to quantify the contribution of each construct to the similarity score attributed to source files. This analysis will reveal the extent of the correlation between those constructs and bug localization results.

Finally, we explore the different contributions from each construct by further modifying the bug localization algorithm. First, we evaluate whether suppressing low-contributing constructs influences the result either positively or negatively. Next, we evaluate whether bug localization effectiveness can be increased by attributing higher weights the most influential constructs in the file score equation (Equation 8), thus emphasizing their contribution.

4.1 Motivation

The key success factor for a bug localization technique based on information retrieval lies on its ability to match terms from bug reports and source files successfully. Once we have observed that using the full set of available program constructs significantly increases bug localization effectiveness (Section 3.6.2), it becomes important to understand in more depth how these constructs individually contribute to bug localization. Such knowledge enables us to discard low contributing constructs, as well as attribute higher weights to the most contributing constructs, possibly increasing effectiveness. The contribution of each program construct is the subject of our third research question, restated below.

RQ3: Which program constructs contribute more to the effectiveness of bug localization on C# projects?

To answer this question, we use principal component analysis (PCA). Principal component analysis is a statistical procedure that transforms a number of possibly correlated variables into a smaller number of variables called *principal components* [48]. According to Jolliffe [48]:

> The central idea of principal component analysis (PCA) is to reduce the dimensionality of a data set consisting of a large number of interrelated variables, while retaining as much as possible of the variation present in the data set. This is achieved by transforming to a new set of variables, the principal components (PCs), which are uncorrelated, and which are ordered so that the first *few* retain most of the variation present in *all* of the original variables [48].

Translating bug localization domain to PCA, the constructs are the variables and the similarity scores are the variable values. As a result, PCA generates a set of new variables – the principal components – with varying degrees of correlation with the original variables. The PCs are presented in decreasing order of contribution to the total variance of the dataset. Therefore, the degree of contribution from a construct to the variance of the dataset can be measured by its correlation with the first PCs.

The reasoning for using PCA to answer RQ3 is that such analysis may indicate that the studied techniques may be more sensitive to a specific construct subset. If this is true, most influential constructs will emerge as highly correlated with the first few principal components (PCs). Furthermore, it is expected that the influence exerted by these constructs could be exploited to increase bug localization effectiveness.

Next section illustrates how the PCA was modeled in order to answer RQ3. All the results presented in this chapter were generated using R version 3.3.0 [49], with additional libraries for analysis [50] [51] and data visualization [52] [53].

4.2 Analysis setup

To perform the analysis, we organize relevant data in the form of a table: variables are laid out in columns, while rows correspond to data points. The 12 C# constructs (Table 3) are the variables. The data points correspond to every buggy file ranked among the top ten positions for every available bug report. The values for each variable are the summands that compose $scoreS$ (Equation 8), the similarity score attributed by AmaLgam's structure component (Section 2.3.3). These values were taken from the best performing construct mapping mode (*Complete* mode, Sections 3.5 and 3.6.2). Table 10 illustrates how data is organized as input to PCA. Afterwards, we discuss in detail the construction of the input table.

Table 10 – Sample of the input for principal component analysis

Bug # / File rank[7]	**Class names**	**Comments**	**Enum names**	**…**	**String literals**	**Struct names**	**Variable names**
Bug #375 / 6th file	0.000	0.141	0.000	…	0.224	0.000	0.354
Bug #535 / 1st file	0.473	0.523	0.000	…	0.148	0.000	0.270
Bug #535 / 3rd file	0.451	0.131	0.000	…	0.177	0.000	0.358
Bug #742 / 3rd file	0.345	0.410	0.282	…	0.085	0.000	0.135
Bug #850 / 1st file	0.514	0.427	0.000	…	0.532	0.000	0.569
Bug #850 / 2nd file	0.355	0.422	0.961	…	0.461	0.000	0.000
…	…	…	…	…	…	…	…

Informally, PCA tries to "explain" a data set consisting of many variables using a smaller number of variables. Since our assumption is that a subset of the constructs is able to "explain" most of the bug localization results, the constructs represent the variables for the analysis. The variable values are the scores attributed by AmaLgam's structure component with respect to each construct. Recall Equation 8:

$$scoreS = sim(f, b) = \sum_{fp \in f} \sum_{bp \in b} sim(fp, bp)$$

AmaLgam's structure component breaks source files and bug reports into parts: bug reports are split into summary and description, and source files are split in parts that contain only constructs from a specific type. In the *Complete* mode adaptation (Section 3.5), 12 C# constructs are considered (Table 4). The score attributed by the structure component ($scoreS$) is the summation of the similarities of each pair of file and bug parts. Thus, for each construct, there is a term from $scoreS$ that reflects its specific contribution to the structural similarity score. These are the variable values used as input for the PCA (Table 10).

Since our goal is to understand the contributions of each construct to the effectiveness of bug localization, we must select effective instances from the data set. Therefore, we selected those files that were both *buggy* according to the oracle (Section 3.2) and *high-ranked* by the technique, i.e., ranked among the top 10 positions. The number of positions, 10, is consistent with the Hit@10 metric (Section 3.2). Thus, data points (rows in Table 10) correspond to every buggy file ranked among the top 10 positions for every available bug report.

[7] Sample taken from project akka.net.

After applying the described selection criteria, 363 data points were selected to compose the PCA input. Table 11 summarizes descriptive statistics for the selected data points.

Table 11 – Descriptive statistics for PCA input – MAP

Variable	Min	Median	Max	Avg.	Std. dev.
Class names	0.000	0.210	1.511	0.269	0.244
Comments	0.000	0.129	0.706	0.156	0.157
Enum names	0.000	0.000	1.208	0.109	0.254
Field names	0.000	0.121	1.204	0.169	0.193
Interface names	0.000	0.000	1.081	0.077	0.185
Method names	0.000	0.166	1.011	0.203	0.171
Namespace names	0.000	0.103	1.678	0.271	0.362
Parameter names	0.000	0.121	1.108	0.174	0.189
Property names	0.000	0.164	1.237	0.214	0.207
String literals	0.000	0.132	1.316	0.190	0.216
Struct names	0.000	0.000	1.567	0.061	0.225
Variable names	0.000	0.138	1.617	0.183	0.198

Once again, we remind the reader that the values presented in Table 10 and Table 11 refer to the scores attributed by the structure component to each file, considering only constructs of a particular type. These scores are summed to compose the structural similarity score (Equation 8), which is then used to calculate the final similarity score for a file (Equation 11). Therefore, it is expected that minimum values for all constructs are equal to zero. This means that, for every construct, there has been at least one file with no construct of that particular type matching any terms from the bug report. This is expected because source files do not usually contain instances of every existing language feature or construct, let alone instances that match terms from a specific bug report.

Some constructs, namely enums, interfaces, and structs, had median values equal to zero. This means that, within the selected sample (which consists of high-ranked buggy files), these were the constructs that matched bug report terms for fewer occasions. This is an indicator of low contribution from these constructs. However, we will proceed to the principal component analysis before formulating a definitive answer to RQ3.

4.3
Contribution of program constructs

This section answers RQ3, i.e., which C# constructs contribute more to the effectiveness of bug localization. The answer is obtained by applying principal component analysis (PCA) to the dataset prepared in Section 4.2. First, the analysis will transform data and express it as a series of dimensions with varying degrees of correlation with the original constructs (Section 4.3.1). Next step is to determine the correlation of each construct with the returned dimensions (Section 4.3.2), thus answering RQ3.

4.3.1
Variances of principal components

PCA transforms input data into a coordinate system such that the highest variance lies on the axis corresponding to the first principal component. Remaining components represent *dimensions* that account for a decreasing amount of variance. In other words, the first components explain most of the variance of the data. In our context, we explore PCA to understand which constructs better explain the data variance on bug localization results.

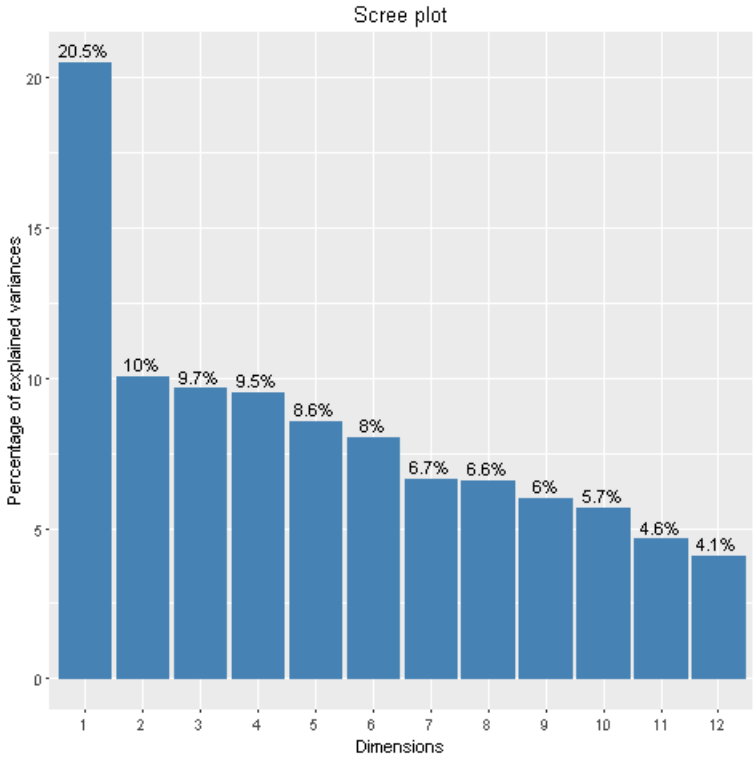

Figure 4 – Variance corresponding to each principal component

Figure 4 presents the degree of variance explained by each of the 12 PCs. While X-axis represents the 12 PCs, Y-axis indicates the percentage of explained variances. Figure 4 shows that the first principal component (PC1) accounts for 20% of the variance in the data, twice as much as PC2. However, the *difference* in the variance of the remaining PCs is much smaller. From PC2 through PC12, percentage of variance smoothly decreases from 10% to 4.1%. Table 12 displays the cumulative percentage of the variance from the first through the last component.

Table 12 – Distribution of variance through the principal components

Component	% variance	Cumulative % variance
PC1	20,5%	20,5%
PC2	10,0%	30,6%
PC3	9,7%	40,2%
PC4	9,5%	49,7%
PC5	8,6%	58,3%
PC6	8,0%	66,3%
PC7	6,7%	73,0%
PC8	6,6%	79,6%
PC9	6,0%	85,6%
PC10	5,7%	91,3%
PC11	4,6%	95,9%
PC12	4,1%	100,0%

As mentioned in Section 4.1, one of the main applications of PCA is to reduce dimensionality from a dataset. This is possible when the first few components account for a high percentage of the variance. What may be considered a high percentage of variation is subjective, although the literature suggests a sensible cutoff is very often in the range 70% to 90% [48]. Considering the distribution presented in Table 12, it would be necessary to retain the seven first PCs to account for 70% of the variance. Likewise, the ten first PCs would have to be retained to account for 90% of the variance. From Figure 4, it becomes clear that, except for PC1, all remaining components present comparable contributions to the structural similarity scores. Although there is no strict rule, typical contributions that allow components to be safely discarded are below 1% [48]. Hence, no component can be confidently discarded due to a negligible contribution.

Although all construct types contribute to the final scores, analysis of the variances (Figure 4) suggest that some constructs contribute more than others do. These are probably associated with PC1, which by itself accounts for 20% of the variance in the results. It remains to be investigated which constructs are associated

with the first few principal components and whether this association can be exploited in order to increase bug localization effectiveness.

4.3.2
Constructs associated with principal components

The degree of relationship between original variables and principal components created by the analysis can be measured by their correlation coefficients. A positive correlation indicates that both values (original variable and PC) increase simultaneously. Therefore, positive correlations reveal constructs that positively contribute to the result. Conversely, negative correlations indicate that while one of the values increases, the other one decreases. This situation could be interpreted as a "wrong clue" to the technique, as the negatively correlated construct would be assigning higher scores to files that, according to the rest of the constructs, should have lower scores. Therefore, constructs with a negative correlation to the PCs are likely to be negatively contributing, i.e., "disturbing" the results. Figure 5 depicts the correlation between constructs and principal components in the form of a correlogram [54].

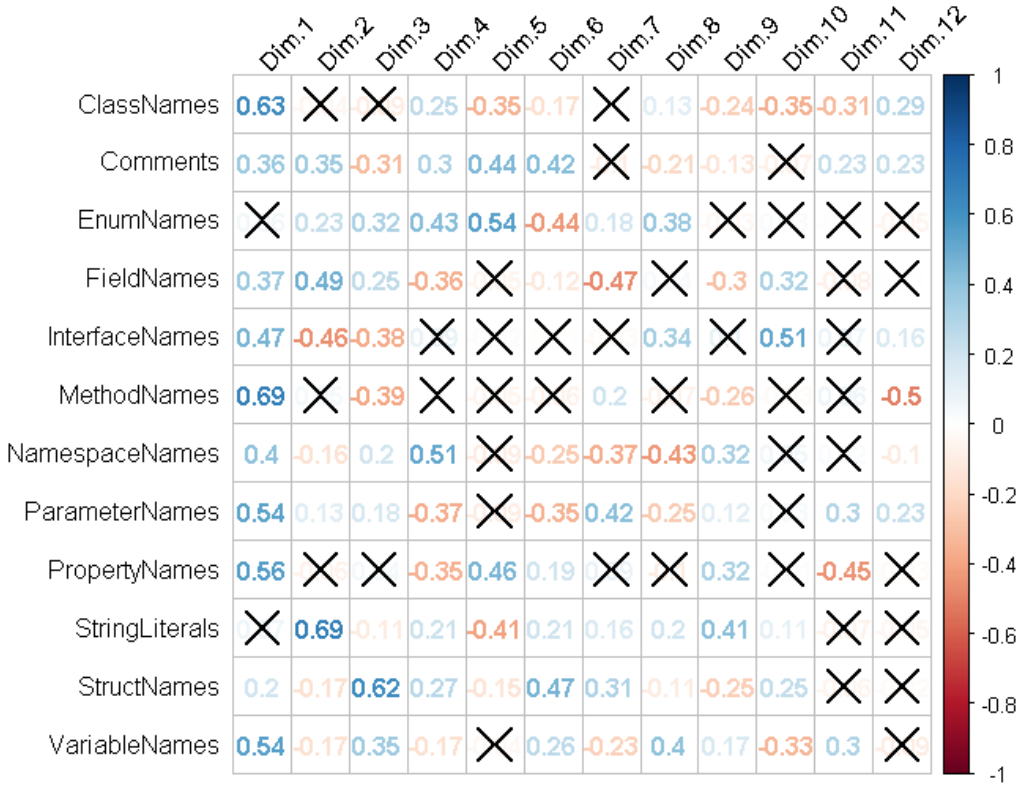

Figure 5 – Correlation between variables and principal components

In Figure 5, blue values represent positive correlations, while red values indicate negative correlations. Higher absolute values indicate stronger correlations. Thus, the closer to +1 the correlation is, the greater the contribution of the construct. Similarly, constructs with correlations close to -1 are more likely disturbing the effectiveness of the technique. The strength of the correlation is also given by the intensity of the color: dark blue and dark red circles indicate strong positive and negative correlations, respectively. Statistically insignificant correlations are signaled with a dark "×".

4.3.2.1 Positive correlations

It is possible to see that many constructs are positively correlated with the first principal component (Dim.1). Method and class names are the ones with the strongest positive correlation. This means that method and class names are the most influential constructs regarding the first dimension extracted by the PCA. This result was expected as classes and methods often represent the most important domain abstractions realized in program files. They embrace some other inner constructs in a file, were the bugs are often "located". Given their importance in the system domain, the names of such (class or method) abstractions naturally have to be reasoned about when someone is either reporting or locating a bug.

The construct with the third highest correlation to the first PC is *Properties*. *Properties*, alongside with *Structures*, is one of the two C# constructs that have no equivalent in Java (Table 3). The contribution of *Properties*, though, was more relevant than that of *Structures*. This can be explained by the fact that *Structures* usually represent simple data structures, with little or no behavior, and therefore are less prone to be associated with bugs. Moreover, *Structures* are independent constructs, while *Properties*, on the other hand, are members of classes. Therefore, it is expected that *Properties* be more closely related to domain abstractions already represented by classes, increasing their chances to be mentioned in bug reports.

After *Properties*, the next constructs more correlated with PC1 are *Parameters* and *Variables*. *Variables* represent a ubiquitous concept of programming languages, and its relevance to IR-based bug localization is no surprise. *Parameters* are used to pass values or variable references to methods [55]. Although *Parameters* are strictly different from *Variables*, their purposes are quite

similar. We have discussed the possibility of considering some constructs equivalent, including *Parameters* and *Variables*, with the *Mixed* construct-mapping mode (Section 3.5). However, we have observed the *Complete* mode, i.e., considering the constructs separately, yielded better results (Section 3.6.2). The high correlation of these two constructs with PC1 may explain the advantage of the *Complete* mode. As both constructs proved to be relevant (Figure 5), considering them separately had the effect of raising the similarity score (Equation 8).

Each PC represents a different dimension of the original dataset. Notice that PC1 is highly correlated to *Methods*, *Classes*, *Properties*, *Parameters*, and *Variables*. In C#, methods and properties are class members. Likewise, parameters and variables occur inside of methods. Therefore, as *Methods* and *Classes* are containers of other constructs, such as *Properties*, *Parameters*, and *Variables*, it is expected that they co-occur, hence their high correlation in the first PC. Notice, however, that the container constructs, i.e., *Methods* and *Classes*, have the highest correlation. This means that, although the inner constructs (*Properties*, *Parameters*, and *Variables*) *do* contribute, their contribution is overshadowed by that of the container ones (*Methods* and *Classes*).

The construct with the strongest positive correlation with the second dimension is *String literals*. This construct had a negligible effect on the first principal component. However, the strong correlation with the second component indicates that, overall, it still has a significant contribution to bug localization effectiveness, as Table 11 suggests. The importance of *String literals* may be explained by the fact that many bug reports include error messages, which are often included in the source code as string literals. This finding actually reinforces that *String literals* should be explicitly considered in structured IR-based bug localization models

Moreover, the fact that *String literals* were more correlated with the second PC, rather than the first, is meaningful. As aforementioned, each PC represents a different dimension of the data. Thus, the contribution of *String literals* occurs in a different dimension than that represented by PC1. This means that files with high scores due to similarity with *String literals* did not have high scores due to method or class name similarity, for example. This fact can be interpreted as an indication that some files would only be located due to the similarity of bug reports with *String*

literals. This is an interesting result, as *String literals* were not considered by BLUiR [4] nor AmaLgam [5], despite being a frequently used construct.

Similar reasoning can be applied to the third PC, where *Structures* are the most relevant construct, and so forth. However, as one advances into the subsequent PCs, one must remember that the relevance of the PCs decreases (Figure 4). Moreover, constructs with negative correlations become more common. Thus, an analysis of the influence of negative correlations is also necessary.

4.3.2.2 Negative correlations

No construct showed negative correlation with the first principal component. However, from the second PC onwards, negative correlations start to appear. The highest negative correlations observed were for *Methods*, on PC12 (-0.5), followed by *Fields* on PC7 (-0.47), and *Interfaces* on PC2 (-0.46). However, the percentage of the variance explained by these components are 4.1%, 6.7%, and 10%, respectively (Figure 4). Thus, the strong negative correlation displayed by *Interfaces* represent a relevant concern.

Interfaces presented a strong negative correlation as early as the second dimension. Although it was also responsible for a similar contribution on PC1, its relative influence within that particular PC was lower than in PC2 and PC3: it has the sixth largest absolute correlation value on PC1 and the third largest value on PC2 and PC3. Apart from PC1, the positive contributions from *Interfaces* appear only on PC8 (fourth largest) and PC10 (first largest). These dimensions, however, account for 6.6% and 5.7% of the variance observed in the scores. Therefore, the positive contribution from *Interfaces* are relatively low, compared to other constructs.

Descriptive statistics presented in Table 11 points at *Interfaces*, alongside with *Enumerations* and *Structures*, as the constructs with the lowest contribution to bug localization effectiveness. This suggests that these constructs are less frequently (i) mentioned in bug reports; or (ii) involved in bug-fixing commits. These are plausible explanations due to the essentially static nature of these construct types. From the three, only structs can contain some sort of dynamic behavior (functions) [55]. Therefore, any bug that stems from these construct types

is likely to be detected at compile time, thus not living long enough to generate a bug report.

PCA confirmed the low contribution from *Enumerations* and *Structures* (`enums` and `structs`). As for *Interfaces*, however, it revealed a strong negative correlation between this construct type and the second principal component. Such observation prompts us to investigate whether bug localization effectiveness could be improved by removing *Interfaces* from the analysis. In the next section, we present the results of another AmaLgam execution, however this time using a different construct-mapping mode. The *Complete* mode (Section 3.5) will be adapted to consider all C# constructs (Table 3) except *Interfaces*, in order to investigate the effects of removing a relatively low-contributing construct type on bug localization effectiveness.

4.4 Effects of constructs on bug localization results

This section explores the effects of program constructs on bug localization. Section 4.3.2 determined constructs of interest for such exploration, i.e., constructs lowly and highly correlated with bugs effectively located by AmaLgam. RQ4 asks whether the suppression of low-contributing constructs could increase bug localization effectiveness. *Interfaces* emerged as the construct with higher negative correlation – thus, less correlated – with effectively located bugs. Therefore, RQ4 will be answered in Section 4.4.1 by adapting AmaLgam to ignore interface names and, then, applying this adapted version on the 20 C# projects that comprise the experimental dataset (Section 3.3).

In contrast, RQ5 inquires about the effects of emphasizing constructs highly correlated with bugs that were effectively located by AmaLgam, namely, *Methods* and *Classes* (Section 4.3.2). RQ5 is answered in Section 4.4.2, which describes how AmaLgam is adapted to emphasize method and class names and presents the results obtained from applying it on the 20 C# projects.

4.4.1 Suppression of low-contributing constructs

The influence of each program construct on the similarity scores attributed by AmaLgam is not homogeneous (Section 4.3). The correlation of these scores with

the dimensions revealed by principal component analysis (Figure 5) made it clear that some constructs exert greater influence on bug localization effectiveness than other constructs do. It is unclear, however, whether *negative* correlations can disturb results. That is the subject of our fourth research question:

RQ4: Does the effectiveness of bug localization increase with the suppression of constructs with the lowest contributions?

Principal component analysis results showed that the constructs with the lowest contribution are *Interfaces*. In fact, *Interfaces* are the constructs with the larger *negative* correlation with the principal components. Thus, while similarity scores from positively correlated constructs increase together, scores from interface names decrease. To assess whether this effect has any influence in bug localization results, we used the dataset of 20 C# projects to run AmaLgam using a slightly modified *Complete* mode (Section 3.5). This modified mode considers all the 12 available C# constructs except for *Interfaces*. Results are summarized in Table 13.

Table 13 – Effect of the suppression of interface names – MAP

Mode	Min	Median	Max	Avg.	Std. dev.
Complete	0.055	0.198	0.573	0.244	0.154
Without interfaces	0.055	0.200	0.582	**0.245**	0.158

Removing *Interfaces* from the localization process increased AmaLgam's average MAP from 0.244 to 0.245 (0.4%). Median and maximum MAPs were also slightly increased, while minimum MAP was unchanged. This is a positive, although negligible, increase on AmaLgam results, with no statistical significance. Therefore, it is not possible to answer RQ4 positively based on our dataset. As it cannot be said that *Interfaces* hamper bug localization, it is not needed to remove this or any other low-contributing construct from a bug localization model based on structured information retrieval.

4.4.2 Emphasis on most contributing constructs

One possible way of increasing effectiveness of bug localization based on structured information retrieval is to assign different weights to the parts in which

source files are split [4]. PCA revealed that *Methods* and *Classes* are the constructs with greater contribution to bug localization results (Section 4.3.2). Thus, our fifth research question (repeated below) asks whether it is possible to increase the effectiveness of a technique by emphasizing highly contributing constructs.

RQ5: Does the effectiveness of bug localization increase with the emphasis on constructs with the highest contributions?

To perform this evaluation, we modify AmaLgam once more, by allowing it to use different weights for each file part generated during source file splitting. The formula originally defined in BLUiR [4], and also used by BLUiR+ [4] and AmaLgam [5] (Equation 8) is replaced by:

$$sim(f, b, w) = \frac{\sum_{i=1}^{n}\left[w_i \sum_{bp \in b} sim(fp_i, bp)\right]}{\sum_{i=1}^{n} w_i}$$

Equation 12 – Weighted structural similarity between a file and a bug report

Equation 12 incorporates weights to the calculation performed by AmaLgam's structure component. Recall from Section 2.3.2 that structural similarity is computed by splitting bug reports and source files in parts corresponding to relevant fields. Bug reports are split into summary and description, while source files are split into as many parts as the construct-mapping mode being used (Section 3.5). Since we are using the *Complete* mode, where all the 12 constructs available in C# are used, $n = 12$ in the above equation.

To answer RQ5, we must choose one or more constructs with high contributions to the results, assign them higher weights (Equation 12), and re-run AmaLgam with this configuration. We selected the two constructs with the highest contribution, *Methods* and *Classes* (Section 4.3.2), and assigned weights of 1.5, 2.0, and 3.0 to each one. These values were arbitrarily chosen to promote a significant variation in the weights, so we could observe to which extent the technique benefits from using higher or lower weights. The results obtained with this execution are displayed in Table 14. The first row repeats AmaLgam results with the *Complete* mode (Table 9), while next rows (referenced by keys) represent the weighted configurations being tested.

Table 14 – Effect of applying higher weights to method and class names – MAP

Key	Mode	Min	Median	Max	Avg.	Std. dev.	p-value
–	*Complete*	*0.055*	*0.198*	*0.573*	*0.244*	*0.154*	–
A	Method weight = 1.5	0.055	0.200	0.574	0.246	0.160	0.0533
B	Method weight = 2.0	0.056	0.195	0.574	0.246	0.160	0.1230
C	Method weight = 3.0	0.050	0.171	0.574	0.238	0.162	0.6186
D	Class weight = 1.5	0.055	0.207	0.582	0.248	0.155	0.0919
E	Class weight = 2.0	0.055	0.207	0.582	**0.265**	0.168	0.2707
F	Class weight = 3.0	0.055	0.194	0.582	0.256	0.171	0.6783

Table 14 shows that, in general, usage of higher weights was able to increase AmaLgam's effectiveness, measured in terms of mean average precision (MAP). *Class* constructs (rows D – F) led to higher MAPs than *Method* constructs (rows A – C) with the same weight for all statistics (minimum, median, maximum and average MAP). As for the weight values selected, best average MAPs were obtained when the emphasized construct had its weight doubled (rows B and E, weight = 2.0).

We used Wilcoxon Signed-Rank tests to assess statistical significance. Unfortunately, none of the results was statistically significant at 95% confidence level, although configurations with weights = 1.5 (rows A and D) came close (94.7% for *Method* and 90.8% for *Class*). The confidence levels decreased drastically as the weights increased. For instance, result for the configuration with the highest MAP, i.e., *Class* weight = 2.0 (row E), had a confidence level of 73% (p-value = 0.2707). As for *Class* weight = 3.0 (row F), not only the MAP dropped, but also the confidence level (32%, p-value = 0.6783). The same was observed for *Method* weight = 3.0 (row C), which means 3.0 is a weight value beyond the threshold both for effectiveness and for significance. Complete statistical analysis is available in the online appendix [46].

The constructs *Methods* and *Classes* presented similar levels of influence to bug localization results, as measured by their correlation to the main component revealed by PCA analysis (Figure 5). Thus, we also tested AmaLgam simultaneously changing the weights of these two constructs. We fixed *Class* weight with a value of 2.0, as it was the best result obtained when constructs had their weights changed individually (Table 14, row E). Then, we applied weights of 1.5 and 2.0 to *Method* constructs. We did not set *Method* weight = 3.0, as this weight value led to smaller MAPs for both constructs evaluated individually (Table 14, rows C and F). Results are presented in Table 15.

Table 15 – Effect of combining higher weights on method and class names – MAP

Key	Mode	Min	Median	Max	Avg.	Std. dev.	p-value
–	*Complete*	*0.055*	*0.198*	*0.573*	*0.244*	*0.154*	–
E	*Class weight = 2.0*	*0.055*	*0.207*	*0.582*	*0.265*	*0.168*	*0.2707*
G	Class weight = 2.0 Method weight = 1.5	0.055	0.198	0.571	**0.266**	0.168	0.0682
H	Class weight = 2.0 Method weight = 2.0	0.056	0.195	0.571	0.253	0.156	0.0412

In Table 15, previous results (in *italics*) are repeated for the sake of comparison. The first row contains results from the *Complete* mode with equal weights for all constructs (Table 9). The second row repeats the result obtained with a weight of 2.0 attributed to *Class* constructs (Table 14, row E). It is possible to see that average MAP increased from 0.244 (*Complete* mode) to 0.266 when *Method* weight is set to 1.5 (row G), and to 0.253 when *Method* weight is 2.0 (row H). Combining *Method* weight = 1.5 and *Class* weight = 2.0 (row G) even increased average MAP compared to using only *Class* weight = 2.0 (row E), although by a negligible amount (only 0.4% higher, from 0.265 to 0.266).

As with the first part of this evaluation, we used Wilcoxon Signed-Rank tests to determine statistical significance. Results for combined weights were closer to the selected 95% confidence threshold: 93% for row G and 96% for row H (p-values of 0.0682 and 0.0412, respectively). Thus, it is possible to state that setting *Class* and *Method* weights to 2.0 (row H) significantly increased bug localization effectiveness, compared with *Complete* mode with equal weights for all constructs.

4.5 Conclusion

In this chapter, we investigated the influence of different program constructs on the effectiveness of structured IR-based bug localization. Initially, we applied principal component analysis (PCA) on results from AmaLgam in the *Complete* mode. This analysis intended to reveal which constructs from the C# language exerted more or less influence on bug localization results.

PCA data suggested that all constructs exerted a significant level of influence on the results (Section 4.3.1). Thus, it was not possible to identify irrelevant constructs just by inspecting PCA data. The analysis also revealed that *Methods* and *Classes* were the constructs with more influence on the results (Section 4.3.2).

In spite of PCA data not having revealed constructs that could be considered irrelevant, some constructs emerged as negatively correlated with bug localization results. The most striking example was *Interfaces* (Figure 5). This negative correlation caused us to investigate what would be the effect of suppressing *Interfaces* from bug localization (Section 4.4.1). Compared to *Complete* mode, results were practically unchanged (Table 13). Thus, we conclude that suppression of low-contributing constructs does not increase bug localization effectiveness.

Another possible way of increasing effectiveness of bug localization is by emphasizing constructs that are more influential, i.e., *Methods* and *Classes* (Section 4.3.2). We investigated that possibility by running AmaLgam with alternative configurations, where different weights were assigned to these two constructs, one at a time (Section 4.4.2). Practically all of these configurations caused the average MAP to increase (Table 14), although none of these improvements reached our statistical significance threshold. Nonetheless, we also tested AmaLgam assigning higher weights to both *Methods* and *Classes*, simultaneously. In this case, a statistically significant improvement was attained when *Methods* and *Classes* were assigned a weight of 2.0 (Table 15). Compared to *Complete* mode, MAP increased from 0.244 to 0.253 (3.7%).

It was previously demonstrated that bug localization based on structured information retrieval benefits from the usage of more program constructs (RQ2, Section 3.6.2). This finding is reinforced by the thorough analysis of the contribution of program constructs performed in this chapter. The answer to RQ3 suggested that all constructs significantly influence bug localization results. RQ4 confirmed this suspicion, by showing there was no significant effectiveness increase when the technique ignored the construct with the smallest contribution.

The usage of weights in the calculation of structural similarity increased bug localization effectiveness. The weight values used in this experiment (1.5, 2.0, and 3.0) were selected empirically. Thus, a possible improvement to this evaluation involves determining optimal weights for each construct. Likewise, we only evaluated the assignment of higher weights to the two most influential constructs, i.e., *Methods* and *Classes*. However, the effect of weighing more than two constructs is still unknown, and could be the subject of future studies. Nonetheless, weighing constructs proved to be a promising way of increasing the effectiveness of bug localization techniques based on structured information retrieval.

5
Conclusion

Determining which parts of the source code need to be modified to remove a bug can be a difficult task. Automated bug localization techniques aim to help developers in this task by providing a list of suspicious files potentially related to the bug. Such techniques can be dynamic or static. Dynamic techniques depend on program execution. Therefore, numerous test cases must be available (Section 2.1.2). Conversely, static bug localization techniques require only source files and a bug report to be applied. Thus, static techniques can be applied to a wider range of scenarios, such as legacy systems where comprehensive test suites are rare or not available (Section 2.1.3).

In recent years, structured information retrieval has been successfully employed by static bug localization techniques, such as BLUiR [4], BLUiR+ [4], and AmaLgam [5]. Some of these techniques incorporate additional data into the localization process, such as bug history (BLUiR+ [4] and AmaLgam [5]) and change history (AmaLgam [5]). Nonetheless, structured information retrieval was still the main responsible for the improvement brought by these techniques (Section 2.3.4).

In spite of the improvements, these techniques are still not effective enough to be widely used in practice. To make matters worse, problems in the dataset preparation (Section 3.4) led these studies [4] [5] to achieve an artificial effectiveness (Section 3.6.1). The lack of realism in empirical studies of the field is likely to become a bottleneck for their adoption. Furthermore, there was a lack of a thorough evaluation of how structured information retrieval could be further explored to increase bug localization effectiveness. These shortcomings motivated us to perform "*a realistic, in-depth effectiveness evaluation of state-of-the-art bug localization techniques*", as stated in our goal (Section 1.3). Our main findings are summarized in the next section.

5.1 Findings

State-of-the-art bug localization techniques are commonly tested in Java software systems only [3] [4] [5]. Thus, in order to contribute to the body of knowledge on bug localization, we decided to conduct our studies in a previously untested OO language. We have selected C#, as it is similar, however significantly different from Java (Section 3.1). In particular, the set of constructs available in both languages is different (Table 3), which enhances the relevance of selecting a different language to evaluate structured IR-based bug localization techniques.

The lack of previous evaluations using C# software systems obliged us to establish a baseline for further comparisons. Thus, we formulated RQ1:

RQ1: Are BLUiR, BLUiR+, and AmaLgam effective to locate bugs in C# projects?

To answer this question, we applied the three techniques on the 20 selected C# projects (Section 3.3). Effectiveness, measured in terms of mean average precision (MAP), was close to the values reported in Java studies, indicating the techniques could be successfully applied to C# projects (Section 3.6.1.1). There was a significant MAP variation across the projects in our sample, which is commonly omitted in previous empirical studies. In certain cases, MAP was even higher than 0.5, while MAP was close to 0.1 in others. This high variation shows that structured IR has already potential to be applied in certain industry C# projects, where: (i) bug reports are used with proper discipline, and (ii) the text produced by bug report authors share some vocabulary with the program itself.

However, a comparison of results obtained in different sets of projects would not be appropriate. Moreover, being aware of the experimental shortcomings of previous studies (Section 1.2.1), we needed to establish a reliable baseline against which subsequent results would be compared. Then, we ran the same techniques on a dataset that implemented the preparation steps necessary to mitigate the mentioned shortcomings (Section 3.4). Results indicated that effectiveness with the dataset preparation steps was, on average, 37% smaller than the effectiveness without those steps (Table 7).

In the original studies, AmaLgam was the most effective from the three techniques [5]. The same happened in our study, both without (Table 5) and with

(Table 7) dataset preparation steps implemented. Thus, we restricted subsequent evaluations to AmaLgam only.

Some of the experimental shortcomings handled in this evaluation had been pointed out by other studies [14] [17]. Our evaluation reinforces findings from these studies, by providing evidence of the effect of those biases on a different programming language, which was not addressed in previous studies. Furthermore, it served as a realistic parameter against which it was possible to compare subsequent results, described in the next subsections.

5.1.1 Usage of constructs

The set of constructs explicitly considered by BLUiR [4] comprises class names, method names, variable names, and comments (Section 2.3.2). Considering that some Java constructs were left out by BLUiR, as well as the fact that C# has a different set of constructs, we formulated RQ2:

RQ2: Does the addition of more program constructs increase the effectiveness of bug localization on C# projects?

We devised two construct mapping modes in addition to the default mapping used by the Java studies (Table 4). Results showed that *Complete* mode, where source files are split in one part for each available C# construct, was the most effective mode, increasing MAP in 18%, from 0.206 to 0.244. This result suggests that structured IR techniques should leverage the usage of program constructs to the maximum possible extent, explicitly including all the available constructs into their process.

5.1.2 Influence of constructs

We became aware that the inclusion of all available constructs increased bug localization effectiveness (Section 3.6.2). However, it was not clear to which extent each construct contributed to the effectiveness increase. To investigate this matter, we formulated RQ3:

RQ3: Which program constructs contribute more to the effectiveness of bug localization on C# projects?

We answered this question using principal component analysis (PCA). The analysis focused on the similarity scores attributed to effectively localized buggy files. These scores were tabulated and transformed into a different dataset, composed of dimensions (or principal components) sorted in decreasing order of relevance to the original dataset. The correlation of each construct to the most relevant dimensions (Figure 5) determines the degree of influence of the constructs. Thus, it was possible to verify that *Methods* and *Classes* were the most influential constructs regarding bug localization results.

PCA also revealed some constructs negatively correlated with the principal components. In particular, *Interfaces* showed a strong negative correlation in the second dimension of the data (Figure 5). This negative correlation led us to formulate RQ4, questioning whether there would be any construct disturbing bug localization.

RQ4: Does the effectiveness of bug localization increase with the suppression of constructs with the lowest contributions?

To address this question, AmaLgam was adapted to consider all the 12 C# constructs (Table 3), except *Interfaces*. In fact, MAP increased without considering interfaces (Table 13). However, the improvement was negligible, and not statistically significant. Therefore, we conclude there is little gain in removing constructs from bug localization techniques based in structured IR.

5.1.3 Weighted similarity calculation

The suppressing of constructs with little influence on bug localization results did not significantly increase bug localization effectiveness. However, we still needed to evaluate how the most relevant constructs could influence bug localization. This was the subject of RQ5:

RQ5: Does the effectiveness of bug localization increase with the emphasis on constructs with the highest contributions?

This question was answered by performing another adaptation to AmaLgam and allowing it to run with different weights assigned to each program construct. Similarity calculation splits source files into parts that contain only constructs from

a specific type. Then, the final similarity score of a file is the sum of the similarities of its parts (Equation 8). In the weighted variation, the final similarity score is a weighted average of the similarities of its parts (Equation 12).

We assigned various weight values to *Method* and *Class* constructs, both in isolation (Table 14) and combined (Table 15). Most of the values tested increased bug localization effectiveness. However, a statistically significant improvement was achieved with weight values of 2.0 for both *Classes* and *Methods*. The weight values were empirically determined, suggesting there might be optimal values that lead to even better results. Nonetheless, this result shows that structured IR-based techniques can be fine-tuned to increase effectiveness even further.

5.2 Contributions

Structured information retrieval allows bug localization techniques to exploit language features – constructs – in order to increase their effectiveness. Java is the programming language that more often appears in bug localization studies [2] [3] [4] [5] [6] [7]. Our study contributes by performing an evaluation of state-of-the-art bug localization techniques on a set of programs written in a previously untested language, namely C#. This evaluation provides evidence of the effectiveness of bug localization techniques on another important and widely used language [18] [19]. Nonetheless, while the evidence we provide is language-specific, the findings of our study may as well be applied to different programming languages, including Java.

Next subsections discuss our contributions in further detail.

5.2.1 Alternatives to increase bug localization effectiveness

We have evaluated structured information retrieval aspects that were unexplored in previous bug localization studies. Particularly, we performed an in-depth study on the influence of program constructs on bug localization effectiveness. In summary, our findings indicate that bug localization techniques should (i) consider the entire set of program constructs that can be extracted from source files and (ii) attribute higher weights to constructs that are more relevant. We provided evidence of the effectiveness of these measures on a set of 20 C#

projects. Nevertheless, these measures could be applied in future techniques designed to work with different programming languages as well.

Studies on bug localization have been trending towards the adoption of hybrid models, which aggregate multiple sources of information as a strategy to increase bug localization effectiveness [3] [4] [5] [7] [32]. Literature indicates that the usage of hybrid models is an assured way to evolve bug localization techniques. However, structured IR-based techniques can still benefit from the improvements brought by our study, regardless of how much additional information their models incorporate. Therefore, it is important to exhaust the possibility of improvement associated exclusively with structured information retrieval. Our study contributes to this goal, by providing alternatives to increase the effectiveness of techniques based on structured IR.

5.2.2 First bug localization study using C#

Most bug localization techniques have been tested on software projects written in Java [2] [3] [4] [5] [6] [7] or C [6] [15]. We were unable to find studies applying bug localization to other object-oriented languages. In particular, this is the first bug localization study involving the C# language, to the best of our knowledge.

Preparation steps (Section 3.4) were applied to the experimental dataset in order to mitigate bias (Section 1.2.1), therefore ensuring the realistic evaluation which was part of our goal (Section 1.3). In addition to these preparation steps, developing the study in a language other than Java can also be considered a decision that favors realism. While structured IR *techniques* are language-specific, the structured IR *approach* itself is not. Therefore, implementing techniques that apply the principles of a structured IR technique to different programming languages helps to strengthen the confidence in their results.

5.2.3 Replication package

Since we could not find other similar studies applied to the C# language, we had to develop a set of tools to support the experiments. These tools are described in the next paragraphs.

GitHub data extractor. It uses the GitHub public API to download issue and commit data. It saves downloaded data in JavaScript Object Notation (JSON) format. It also downloads the appropriate program version according to issue creation dates (Section 3.4.1).

Preprocessor. It performs the text preprocessing steps (Section 2.2.1) on downloaded issues and source code and generates some term frequency statistics (Section 2.2.2). It saves the result in XML files, in order to avoid repeated computation on each execution of the bug localization.

Bug localizer. It applies one of the bug localization techniques to a list of preprocessed bug reports, generating lists of suspicious files. The technique to be applied (BLUiR, BLUiR+, or AmaLgam) is determined by parameters *a* and *b*, which determine weights of the scores generated by each component (Section 2.3.3). Results – such as file names, rank, and suspiciousness scores – are saved to CSV files (comma-separated values). The program also generates evaluation metrics (Section 3.2) and save them to CSV files as well.

In addition to the aforementioned tools, resources used in this study – downloaded issues, commits, preprocessed source code, raw results and statistical analysis – are available online [46].

5.3 Future work

To conclude our study, we highlight possibilities for future work that might develop some of our findings, thus advancing the bug localization field.

Weighted similarity calculation was shown to increase bug localization effectiveness (Section 4.4.2). However, our evaluation selected weight values empirically. The effectiveness could be further increased if an optimal set of weights could be found. The calculation of the weights could be automated and performed on a per project basis, which might lead to even better results.

In fact, project characteristics significantly influence bug localization effectiveness. We have observed in our dataset a wide variation on effectiveness across different projects. In some projects, the average MAP was above 0.5, while in others it was below 0.1 [46], suggesting that, for particular projects, IR-based bug localization is feasible. However, there is a need to perform further analysis of project characteristics that affect bug localization effectiveness. A key success

factor for the success of IR-based bug localization is that both bug reports and source files contain terms that relate to the domain of the application. Therefore, practices followed by developers may have a significant influence on bug localization. Projects with stricter policies regarding naming conventions in source code and bug reporting might be those where IR-based bug localization is more effective.

Therefore, to be successfully applied, IR-based bug localization techniques require the adoption of practices that encourage developers and users to share the same vocabulary. These practices could be supported through recommendation systems, which could analyze the code to identify relevant terms and suggest the use of these terms to the author while the bug report is being written. Likewise, IDEs could be improved to advise programmers to use a consistent set of terms. For instance, if a software project uses the term *customer* to represent a domain concept, when a developer creates a class named *Client*, the IDE could recommend the usage of the first term. This consistency would improve term matching, which is vital to the success of information retrieval models.

We focused our exploration of structured IR-based bug localization on source files – specifically, on program constructs. Nevertheless, similar exploration could be carried out with bug reports. Regarding bug report contents, the same kind of recommendation for ensuring terminological consistency that was suggested for programmers could also be directed to users writing bug reports. As for bug report structure, we followed the approach defined in [3] [4] [5] and considered only bug report summary and description. However, additional information could be considered. For example, many platforms, such as GitHub, allow developers to carry out a discussion about the bug, saving the exchanged messages within the bug report. These discussions could become a third part in which bug reports are split (Section 2.3.2). As these discussions can become lengthy, bug report summarization could be applied to restrict discussion contents to more meaningful terms.

Regarding C#, future work includes the creation of a standard bug dataset, similar to iBUGS [36] or moreBugs [44], containing bugs from C# projects. This would allow studies with better potential for generalizability involving the C# language. Although we have made downloaded issues available online [46], only raw data is available. Ideally, a standard dataset would have all or most of its

constituent bug reports manually verified in order to avoid misclassification, i.e., regular issues wrongly classified as bugs. Other desirable features of a bug dataset include online search and visualization functionalities and the possibility of downloading parts of the dataset according to some criteria, e.g., bugs opened after a specific date, bugs closed more than 30 days after being opened, bugs opened and closed by the same person.

Another possible line of work involves conducting analytical studies to improve bug localization knowledge on different languages and different types of files. For instance, bugs are not always located in source files. Sometimes they can be found in different kinds of files, such as configuration files. Developers could benefit from having specific localization techniques for these kinds of bugs.

Finally, it is also important to assess the usefulness of bug localization when actually applied by developers. This is a fundamental step to promote the adoption of bug localization techniques. Therefore, these techniques should be assessed via controlled experiments involving developers, such as the one reported in [10]. Controlled experiments would reveal how developers use bug localization results to locate buggy files. Nevertheless, the bug localization field still demands effectiveness increase of existing techniques, as observed in this dissertation. Otherwise, developers would not be confident enough in the techniques to use them, even in controlled experiments. Thus, increasing the effectiveness of the techniques is an important first step towards widespread adoption of automated bug localization.

References

1. ISO/IEC/IEEE. **Systems and software engineering – Vocabulary**. ISO/IEC/IEEE 24765:2010. Switzerland: ISO/IEC/IEEE. 2010. p. 1-418.

2. LUKINS, S. K.; KRAFT, N. A.; ETZKORN, L. H. Bug Localization Using Latent Dirichlet Allocation. **Information and Software Technology**, 52, n. 9, September 2010. 972-990. Available: <http://dx.doi.org/10.1016/j.infsof.2010.04.002>.

3. ZHOU, J.; ZHANG, H.; LO, D. **Where Should the Bugs Be Fixed? - More Accurate Information Retrieval-based Bug Localization Based on Bug Reports**. 34th International Conference on Software Engineering (ICSE). Zurich, Switzerland: IEEE. 2012. p. 14-24.

4. SAHA, R. K. et al. **Improving bug localization using structured information retrieval**. 28th International Conference on Automated Software Engineering (ASE). Palo Alto, California, USA: IEEE. 2013. p. 345-355.

5. WANG, S.; LO, D. **Version History, Similar Report, and Structure:** Putting Them Together for Improved Bug Localization. 22nd International Conference on Program Comprehension (ICPC). Hyderabad, India: ACM. 2014. p. 53-63.

6. RAHMAN, F. et al. **BugCache for Inspections:** Hit or Miss? 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE). Szeged, Hungary: ACM. 2011. p. 322-331.

7. SISMAN, B.; KAK, A. C. **Incorporating Version Histories in Information Retrieval Based Bug Localization**. 9th Working Conference on Mining Software Repositories (MSR). Zurich, Switzerland: IEEE. 2012. p. 50-59.

8. LEWIS, C.; OU, R. Bug Prediction at Google, 14 December 2011. Available: <http://google-engtools.blogspot.sg/2011/12/bug-prediction-at-google.html>. Accessed: 05 September 2015.

9. MURPHY-HILL, E. et al. **The Design of Bug Fixes**. 35th International Conference on Software Engineering (ICSE). San Francisco, California, USA: IEEE. 2013. p. 332-341.

10. WANG, Q.; PARNIN, C.; ORSO, A. **Evaluating the Usefulness of IR-based Fault Localization Techniques**. 2015 International Symposium on Software Testing and Analysis (ISSTA). Baltimore, Maryland, USA: ACM. 2015. p. 1-11.

11. KOCHHAR, P. S. et al. **An Empirical Study of Adoption of Software Testing in Open Source Projects**. 13th International Conference on Quality Software (QSIC). Nanjing, China: IEEE. 2013. p. 103-112.

12. RAO, S.; KAK, A. **Retrieval from Software Libraries for Bug Localization:** A Comparative Study of Generic and Composite Text Models. 8th Working Conference on Mining Software Repositories (MSR). Waikiki, Honolulu, HI, USA: ACM. 2011. p. 43-52.

13. MANNING, C. D.; RAGHAVAN, P.; SCHÜTZE, H. **Introduction to Information Retrieval**. New York, NY, USA: Cambridge University Press, 2008. ISBN 0521865719, 9780521865715. Available: <http://dl.acm.org/citation.cfm?id=1394399>.

14. KOCHHAR, P. S.; TIAN, Y.; LO, D. **Potential Biases in Bug Localization:** Do They Matter? 29th International Conference on Automated Software Engineering (ASE). Vasteras, Sweden: ACM. 2014. p. 803-814.

15. SAHA, R. K. et al. **On the Effectiveness of Information Retrieval Based Bug Localization for C Programs**. 30th International Conference on Software Maintenance and Evolution (ICSME). Victoria, British Columbia, Canada: IEEE. 2014. p. 161-170.

16. PARNIN, C.; ORSO, A. **Are Automated Debugging Techniques Actually Helping Programmers?** 2011 International Symposium on Software Testing and Analysis (ISSTA). Toronto, Ontario, Canada: ACM. 2011. p. 199-209.

17. RAO, S.; KAK, A. A Serious Issue with Some Current Publications on IR-Based Approaches to Automatic Bug Localization. **moreBugs**, 2013. Available: <https://engineering.purdue.edu/RVL/Database/moreBugs/#C5>. Accessed: 14 May 2016.

18. TIOBE SOFTWARE BV. TIOBE Index for April 2016, April 2016. Available: <http://www.tiobe.com/tiobe_index>. Accessed: 19 April 2016.

19. GITHUB, INC. Language Trends on GitHub. **The GitHub Blog**, 19 Aug. 2015. Available: <https://github.com/blog/2047-language-trends-on-github>. Accessed: 19 April 2016.

20. GITHUB, INC. Mastering Issues. **GitHub Guides**, 2014. Available: <https://guides.github.com/features/issues/>. Accessed: 03 July 2016.

21. ATLASSIAN. What is an Issue. **JIRA User's Guide**, 2015. Available: <https://confluence.atlassian.com/jira064/what-is-an-issue-720416138.html>. Accessed: 03 July 2016.

22. LUKINS, S. K.; KRAFT, N. A.; ETZKORN, L. H. **Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation**. 15th Working Conference on Reverse Engineering (WCRE). Antwerp, Belgium: [s.n.]. 2008. p. 155-164.

23. ABREU, R.; ZOETEWEIJ, P.; GEMUND, A. J. C. V. **On the Accuracy of Spectrum-based Fault Localization**. Testing: Academic and Industrial Conference Practice And Research Techniques - MUTATION (TAICPART-MUTATION). Windsor, UK: IEEE. 2007. p. 89-98.

24. JONES, J. A.; HARROLD, M. J.; STASKO, J. **Visualization of Test Information to Assist Fault Localization**. 24th International Conference on Software Engineering (ICSE). Orlando, Florida, USA: ACM. 2002. p. 467-477.

25. ABREU, R.; ZOETEWEIJ, P.; GEMUND, A. J. C. V. **An Evaluation of Similarity Coefficients for Software Fault Localization**. 12th Pacific Rim International Symposium on Dependable Computing (PRDC). Riverside, CA, USA: IEEE. 2006. p. 39-46.

26. WONG, W. E. et al. A Survey on Software Fault Localization. **IEEE Transactions on Software Engineering**, 42, n. 8, 1 August 2016. 707-740. Available: <http://dx.doi.org/10.1109/TSE.2016.2521368>.

27. JIN, W.; ORSO, A. **F3:** Fault Localization for Field Failures. 2013 International Symposium on Software Testing and Analysis (ISSTA). Lugano, Switzerland: ACM. 2013. p. 213-223.

28. WOTAWA, F.; STUMPTNER, M.; MAYER, W. Model-Based Debugging or How to Diagnose Programs Automatically. In: HENDTLASS, T.; ALI, M. **Developments in Applied Artificial Intelligence**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. p. 746-757. ISBN 978-3-540-48035-8. Available: <http://link.springer.com/chapter/10.1007/3-540-48035-8_72>.

29. ZELLER, A. **Isolating Cause-effect Chains from Computer Programs**. 10th Symposium on Foundations of Software Engineering (FSE). Charleston, South Carolina, USA: ACM. 2002. p. 1-10.

30. ZELLER, A.; HILDEBRANDT, R. Simplifying and isolating failure-inducing input. **IEEE Transactions on Software Engineering**, 28, n. 2, February 2002. 183-200.

31. PAPADAKIS, M.; LE TRAON, Y. Metallaxis-FL: mutation-based fault localization. **Software Testing, Verification and Reliability**, 25, n. 5-7, 1 August 2015. 605-628. Available: <http://onlinelibrary.wiley.com/doi/10.1002/stvr.1509/abstract>.

32. CHAPARRO, O. et al. **Improving Text Retrieval Based Bug Localization Using Code Authorship Information**. 32nd International Conference on Software Maintenance and Evolution (ICSME). Raleigh, North Carolina, USA: In press.

33. DIT, B. et al. **Can Better Identifier Splitting Techniques Help Feature Location?** 19th International Conference on Program Comprehension (ICPC). Kingston, Ontario, Canada: IEEE. 2011. p. 11-20.

34. PORTER, M. F. An algorithm for suffix stripping. **Program**, 14, n. 3, 1980. 130-137. Available: <http://www.emeraldinsight.com/doi/abs/10.1108/eb046814>.

35. PORTER, M. F. **The Porter Stemming Algorithm**, 2006. Available: <https://tartarus.org/martin/PorterStemmer/>. Accessed: July 2016.

36. DALLMEIER, V.; ZIMMERMANN, T. iBUGS. Available: <https://www.st.cs.uni-saarland.de/ibugs/>. Accessed: July 2016.

37. DALLMEIER, V.; ZIMMERMANN, T. **Extraction of Bug Localization Benchmarks from History**. 22nd International Conference on Automated Software Engineering (ASE). Atlanta, Georgia, USA: ACM. 2007. p. 433-436.

38. HOVEMEYER, D.; PUGH, W. Finding Bugs is Easy. **ACM SIGPLAN Notices**, December 2004. 92-106. Available: <http://doi.acm.org/10.1145/1052883.1052895>.

39. NGUYEN, A. T. et al. **A topic-based approach for narrowing the search space of buggy files from a bug report**. 26th International Conference on Automated Software Engineering (ASE). Lawrence, Kansas, USA: IEEE. 2011. p. 263-272.

40. BACHMANN, A.; BERNSTEIN, A. **Data Retrieval, Processing and Linking for Software Process Data Analysis**. University of Zurich. Zürich, Switzerland, p. 11. December 2009. (IFI-2009.0003b).

41. POSHYVANYK, D. et al. **Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification**. 14th International Conference on Program Comprehension (ICPC). Athens, Greece: IEEE. 2006. p. 137-148.

42. POSHYVANYK, D. et al. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. **Transactions on Software Engineering**, 33, n. 6, June 2007. 420-432. Available: <http://dx.doi.org/10.1109/TSE.2007.1016>.

43. KIM, S. et al. **Predicting Faults from Cached History**. 29th International Conference on Software Engineering (ICSE). Minneapolis, Minnesota, USA: IEEE. 2007. p. 489-498.

44. RAO, S.; KAK, A. **moreBugs: A New Dataset for Benchmarking Algorithms for Information Retrieval from Software Repositories**. Purdue University. West Lafayette, IN. 2013.

45. MICROSOFT..NET Compiler Platform ("Roslyn"). **GitHub**, 2014. Available: <https://github.com/dotnet/roslyn>. Accessed: 14 May 2016.

46. GARNIER, M. Bug localization in C#, 2016. Available: <http://www.inf.puc-rio.br/~mgarnier/bug_localization/>.

47. KARUS, S.; GALL, H. **A Study of Language Usage Evolution in Open Source Software**. 8th Working Conference on Mining Software Repositories (MSR). Waikiki, Honolulu, HI, USA: ACM. 2011. p. 13-22.

48. JOLLIFFE, I. T. **Principal Component Analysis**. Secaucus, NJ, USA: Springer, 2002. 518 p. ISBN 978-0-387-22440-4. Available: <http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10047693>.

49. R CORE TEAM. **R: A Language and Environment for Statistical Computing**. R Foundation for Statistical Computing. Vienna, Austria. 2015.

50. KORKMAZ, S.; GOKSULUK, D.; ZARARSIZ, G. MVN: An R Package for Assessing Multivariate Normality. **The R Journal**, 6, n. 2, 2014. 151-162. Available: <http://journal.r-project.org/archive/2014-2/korkmaz-goksuluk-zararsiz.pdf>.

51. LÊ, S.; JOSSE, J.; HUSSON, F. FactoMineR: An R Package for Multivariate Analysis. **Journal of Statistical Software**, 25, n. 1, 2008. 1-18.

52. WEI, T.; SIMKO, V. **corrplot: Visualization of a Correlation Matrix**. [S.l.]. 2016. R package version 0.77.

53. KASSAMBARA, A.; MUNDT, F. **factoextra: Extract and Visualize the Results of Multivariate Data Analyses**. [S.l.]. 2016. R package version 1.0.3.

54. FRIENDLY, M. Corrgrams: Exploratory displays for correlation matrices. **The American Statistician**, 56, n. 4, 2002. 316-324. Available: <http://dx.doi.org/10.1198/000313002533>.

55. MICROSOFT CORPORATION. C# Language Specification 5.0. **Microsoft Download Center**, 2012. Available: <https://www.microsoft.com/download/details.aspx?id=7029>. Accessed: 06 July 2016.