UNIVERSITÀ DEGLI STUDI DI CATANIA

Dipartimento di Matematica e Informatica

Dottorato di ricerca in informatica (Internazionale) XXXVI Ciclo

Alessandro Midolo

ReFrame: A Refactoring Framework to Support Developers in Improving and Optimising their Source Code

Ph.D. Thesis

Supervisor: Prof. Emiliano Tramontana

Anno Accademico 2022 - 2023

Declaration of Authorship

I, Alessandro Midolo, certify that I have written this thesis independently and have not used any sources, resources or technical tools other than those specified. All statements that were taken from other publications, either literally or analogously, are marked. I have not submitted the work in the same or a similar form to any other examination authority. I agree that this work may be checked with anti-plagiarism software. Parts of this thesis have been published in conferences, and journals. In particular, Chapter 3 is described in an ICCCM-2022 paper [87] and *Future Internet* journal paper [88], Chapter 4 in a CSSE-2022 paper [90] and SaTToSE-2023 paper [86], Chapter 5 in a WETICE-2020 [41] paper and ICSE-2023 paper [89]. These chapters have been extended and revised when writing this dissertation.

Abstract

In order to better employ the computational power of modern hardware, many sequential programs can be changed to encapsulate parallel instructions. Concurrent executions have the potential to significantly enhance overall application performance, reducing both the execution time and the resource requirements. Nevertheless, manually transforming a sequential application to a parallel version is a complex undertaking, involving a meticulous inspection of several lines of code which could be tedious and time-consuming, an in-depth analysis of the data dependencies to preserve the correctness of the transformations and a balance of the workload among threads to optimise the performance boost. In this context, I propose four distinct refactoring approaches to support developers in this complex task, providing a set of fully automatic tools tailored to different scenarios, encompassing data and control dependence analysis, parallelisation of method calls, transformation of for loops to stream pipelines and matching of algorithms for parallel optimisations. An efficacious test suite is necessary to preserve the source code from inadvertent changes during refactoring. Such changes may involve the editing, addition or deletion of code fragments, resulting in poor consistency between the refactored code and the testing suite. To address this issue, I additionally propose two automatic approaches to generate test cases, catering to both classes and methods. All six proposals are integrated into a framework named ReFrame, which serves the dual purpose of validating these approaches with real word scenarios, and providing a unified, modular interface for developers. Extensive testing has been conducted to demonstrate the correctness and efficiency of these approaches, revealing optimal results in each of the related tasks.

Contents

Chapter 1

Introduction

1.1 Refactoring for Software Optimisation

In the realm of software development, it is common to encounter errors in the delivered product, necessitating subsequent correction upon discovery. It is important to note that these maintenance activities are not driven by wear and tear, but rather by the need to both rectify concealed defects and optimise obsolete or poorly performing features. Over the entire lifespan of a software system, a substantial portion of the budget is allocated to its maintenance rather than its initial development [143].

Refactoring stands as a practical and constructive course of action to guide developers in improving program functionalities while preserving its behaviour, thus reducing the resources needed for software maintenance. The term "restructuring" is employed as a general reference to any form of code base reorganisation or cleanup, with "refactoring" representing a specific type of restructuring. The concept of refactoring was initially introduced in Opdyke's doctoral thesis to address code-level restructuring [96]. Fowler provides a formal definition of refactoring: "A change made to the internal structure of software to make it easier to understand

and cheaper to modify without changing its observable behaviour" [42]. Refactoring has been employed in the realm of software evolution to augment the quality of object-oriented software, including aspects like flexibility, availability, reliability, and maintainability [53]. Ongoing refactoring efforts support the notion that code restructuring reduces developer time by 60% [27].

The focus of most refactoring efforts has predominantly been on enhancing internal structure, with limited emphasis on performance improvement [153]. With the proliferation of multi-core and many-core processors, concurrent programming has gained prominence. A growing number of researchers have begun applying software refactoring to concurrent programming, optimising the concurrent structure to improve performance. Concurrent-oriented software refactoring has thus garnered increased attention. Nevertheless, experienced developers do not always possess the knowledge required to optimise concurrent programs effectively.

While manual refactoring is feasible, tool support is considered indispensable. In today's landscape, a wide array of tools exists to automate various facets of refactoring. Automation offers the advantage of reducing costs and enhancing efficiency. The primary benefit of an automated approach is its independence from direct human intervention, which can be error-prone and exceptionally time-consuming. The automatic transformation of sequential programs to perform operations in a parallel fashion is still an open issue since it requires an in-depth data dependence analysis, which could be significantly complex. Moreover, the transformation into a parallel version should be guided by some strategies that could likely bring performance gains once the parallel version executes.

The state of the art shows some notable approaches that can automatically

change applications into parallel versions. Some approaches propose intrusive transformations, failing to consider the number of changes required to refactor code, focusing solely on inserting concurrent operation within the code, irrespective of the cost of change [21, 20, 64, 34, 152]. Consequently, applying a particular refactoring may entail a substantial overhaul of the system or even necessitate a complete reimplementation. To aid developers in understanding the proposed modifications and their functionality, it is imperative that the changes are syntactically coherent and minimal [97].

Previous approaches solely focus on specific types of instructions, i.e. recursive algorithms and Collections [64, 34], Array [35] and streams [69]. Other approaches treat just some categories of statements [74, 49], or optimise concurrent instructions without introducing further parallelism [148, 120, 98, 156, 155, 6].

Concurrent libraries are made available to developers for supporting parallel programming [67, 57], however it is up to the developers to determine which code snippets are prone to parallelism and how APIs can be used. Several other approaches perform automatic parallel transformation in the executable code [70, 68, 37]. Although the performance gain is superb, the parallel version is hidden from the developer who cannot further modify it, e.g., to solve further requirements, improve some parts, etc.

To address these limitations, I propose a set of automatic approaches to support developers in transforming programs from sequential to parallel. These approaches provide developers with a set of APIs and tools to analyse, improve and optimise their code while preserving its structure, modularity and clarity. The main strengths of these works are the following.

- The proposed refactorings do not alter the structure of the code, indeed the

changes done are minor and syntactically coherent with the overall codebase. This preserves the readability and comprehensibility of the source code.

- The applicability of the approaches includes a wider spectrum of instructions, increasing the number of refactoring opportunities, hence the suitability for different software applications.

- The code is automatically analysed and some portions are selected for the refactoring. This frees the developer from the choice of which code snippet could be prone to parallelism.

- The analysis undertaken by the approaches is complex, detailed, and satisfies all the conditions required to safely run a code in parallel. This guarantees the generation of accurate and reliable code.

- An a priori assessment is performed to evaluate the potential advantages of introducing new threads, by analysing execution paths, and optimising the performance boost gained by the parallelisation.

Chapters 3 and 4 broadly describe the aforementioned approaches, giving details of how the analyses are performed and the methodology behind each approach.

One of the defining attributes of refactoring is its preservation of the program's observable behaviour. Achieving this goal necessitates the capability to execute a comprehensive test suite swiftly, enabling frequent testing without hindrance. Therefore, in most scenarios, the presence of self-testing code is essential for successful refactoring. Frequent test runs result in a minimal code differential, greatly simplifying the bug detection process. Consequently, this dissertation addresses concerns about the potential risks of introducing bugs during refactoring, underscoring the pivotal role of robust testing practices.

1.2 Testing for Validation

Refactorings are typically conceived as small and relatively straightforward adjustments, designed to minimise the likelihood of introducing issues. However, unit tests serve as a protective barrier against the inadvertent introduction of regressions during the refactoring process [146]. Unit tests not only act as a safety net but, owing to their compact size, they also enable rapid evaluation of code modifications, providing prompt feedback to developers.

It becomes necessary to devise new test cases to encompass the modified API, while also debugging existing tests post-refactoring. Refactorings may also entail the relocation of functionality implementations, leading to a situation where test code for a specific functionality may now reside in a test associated with a different program unit. The preservation of test suite quality in the context of refactoring is a pivotal concern in application development [103].

During a refactoring operation, existing code can be restructured into new program units. This can disrupt the consistency between the source code and the corresponding unit tests [85]. For instance, relocating a method from class A to class B introduces incongruity between the structure of the refactored code and the corresponding unit tests, where the test code for the moved method might still reside in class A's test. In essence, the challenge of maintaining consistency between refactored code and tests remains an unsolved issue and calls for tool-supported solutions. Furthermore, empirical evidence suggests that refactorings are often inadequately tested [110].

For assisting developers in generating the code of a test suite, many tools have been proposed. Certain tools are designed to randomly generate sequences of calls and corresponding values to form a test case [15, 36, 99, 119]. Typically, these

random testing tools produce numerous test cases, leading to a significant increase in execution time. Consequently, developers require specific criteria and tools to sift through this extensive set of test cases and select a more manageable subset [4, 66]. Additionally, test cases generated through random approaches often exhibit various test smells, diminishing the effectiveness and quality of the overall test suite [102].

One notably effective test generation tool is Evosuite [46, 145], which employs a search-based approach, utilising evolutionary search to automatically generate test suites with the goal of maximising code coverage [101]. Despite Evosuite's superior coverage and accuracy compared to other generation tools, the process of test generation is time-consuming. Furthermore, successful execution of Evosuite demands a substantial amount of computational resources and time [145].

Recognising the importance of improving the usability and readability of generated tests, it has been noted that such tests are generally less readable than manually implemented ones [51, 115].

The proposed approaches aim to support developers with tools specifically designed to generate test cases for classes and methods. The primary contributions of these approaches include: (i) the generation is targeted to those classes and methods that have not been previously tested. This reduces the number of redundant test cases, which could negatively affect execution times, forcing developers to select a smaller set of test cases [4, 66]; (ii) the generated test cases increase the code coverage of the existing test suite, thereby enhancing its overall effectiveness; (iii) the generation is fully automatic and it allows to select the classes and methods for which generates the tests. This guarantees the possibility to generate tests for refactored code, preserving the consistency of the test suite; (iv) the approaches use a static analysis to get the data needed for the generations. As a result, the execution

time is directly associated with the number of lines of code analysed. This approach eliminates the need for multiple iterations, thereby avoiding time-consuming and resource-intensive processes [145].

These approaches are presented in chapter 5, where methodologies and implementations are widely discussed.

1.3 Proposed Framework

Most of the refactoring approaches assume that developers either already possess a test suite or must manually generate one or use an external tool to ensure that the applied refactorings have not altered their application's behaviour.

While these automated refactoring approaches simplify the application of complex refactoring techniques to enhance code quality, they also impose the burden of updating and maintaining test suites. This can be a demanding and time-consuming task for developers, as they must understand how to properly test the refactored sections and update existing tests to guarantee code consistency and correctness. Consequently, while these tools provide valuable capabilities, they also present a challenge in terms of maintaining test suites, which can be complex and labour-intensive.

Another noteworthy consideration is that many of these approaches focus on individual refactoring techniques. To apply multiple refactorings, developers may need to use various tools, each with its own set of requirements and dependencies. These tools can sometimes be complex to use, necessitating a significant learning curve to master their operation.

The combination of these factors can potentially increase the complexity of a developer's tasks and even slow down the development process [42]. Inexperienced developers might be hesitant to use these tools due to the challenges they pose in ensuring code consistency and correctness, which could result in missed opportunities for optimising and refactoring their code.

In order to provide a unified interface for the approaches introduced in the previous sections, I propose a Java framework called *ReFrame*. The creation of this framework is motivated by several factors: (i) the framework serves as an infrastructure for experimenting with and validating the introduced approaches; (ii) since all the approaches rely on static analysis to collect the necessary data, the framework can perform a single parsing of the source code, making the data available to each approach. This optimisation minimises the execution time of each module; (iii) providing developers with a unified interface both for refactoring and testing during the development process of their application; (iv) by offering a consolidated platform, the framework streamlines the development processes related to refactoring and testing, ensuring the efficacy and accuracy of the proposed methods.

The framework has two main components: the first one is a refactoring unit which provides three different modules to apply concurrent refactoring to source code; the second one is a testing unit with two modules to generate a test suite. The modules provided by the framework are entirely automatic, reducing the time required for the processes and improving the effectiveness of the generations. Figure 1.1 shows an overview of the proposed framework. The first unit is composed by three different modules:

- *Methods Parallelisation*: it provides a library that automatically refactor sequential method calls to a parallel version. The module offers an automated

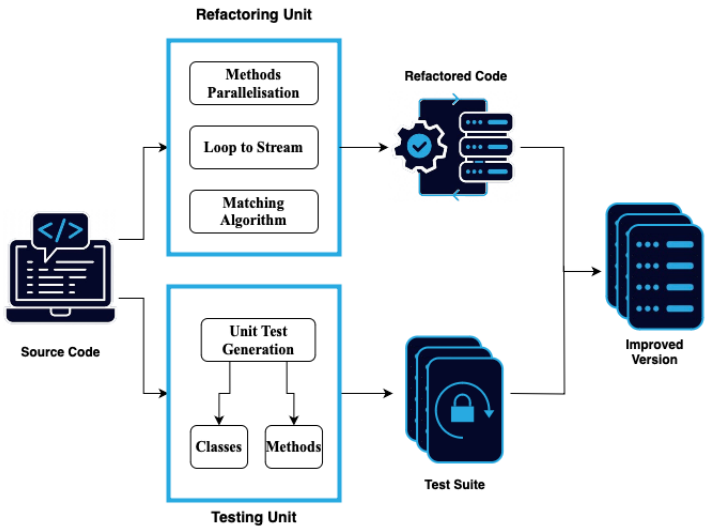


Figure 1.1: Overview of the proposed framework

method to boost code performance by analysing source code. The approach uses control flow analysis, data dependence analysis, and a control flow graph to identify code fragments suitable for parallel execution. The process involves three steps: (i) initial code analysis to find parallelisable sequences of statements; (ii) evaluating data dependencies among statements; (iii) identifying paths with potential for performance improvement based on statement count in a control flow graph. Once a suitable path is found, it is automatically refactored, creating a new thread and inserting synchronisation points as needed. Developers can modify or execute the resulting code

- *Loops to Stream*: this module introduces an innovative approach to refactor for loops into Java stream APIs by establishing templates designed to facilitate code analysis and transformation. Five innovative templates have been introduced that encompass various common loop use cases, encompassing a wide range of statements within their body, including but not limited to conditionals, temporary assignments, secondary pathways, and return statements.

In this module, we have meticulously accounted for the translation of conventional loops to ensure a secure transition from imperative to functional programming, while preserving the original behaviour of the code. Notably, these templates exhibit a higher level of generality compared to prior methodologies, enabling their automatic application. The transformations within stream pipelines facilitate significantly the parallelisation of these functionalities thanks to the APIs provided by the Stream class itself

- *Matching Algorithm*: this module centres on algorithm recognition and introduces an innovative method for automatically identifying algorithms by inspecting their source code. This approach utilises static analysis to extract data from the source code and calculate a similarity score by comparing it with templates of well-known algorithms, facilitating the precise identification of the correct algorithm. The utilisation of templates ensures several advantages: the seamless inclusion of new algorithms in the recognition process, the potential utilisation of multiple versions of the same algorithm to enhance identification accuracy, and the ability to recognise a wide array of algorithm categories (e.g., sorting, searching, traversing, etc.). Once an algorithm is identified, suggestions of different versions of the same algorithm can be made in order to propose code that can better satisfy the conditions required for parallelisation

The second unit is organised in two different modules, one to generate test mainly for classes and the other one for methods:

- The first module automates the generation of test cases that are finely tuned to the behaviour of the target class. This method accomplishes this by utilising

static analysis to understand the functionality of the class under examination. Subsequently, this identified functionality is compared with data regarding the operational characteristics obtained from other classes. As a result, tests are automatically generated by expanding upon existing tests related to classes displaying comparable traits. These generated tests are tailored to the code under examination, rendering them exceptionally effective in improving both code coverage and bug identification

- The second module automates the creation of a set of test case templates for untested methods within an application. Each template encompasses the specific method to be invoked, a corresponding JUnit assertion, and an array of input parameters. The proposed approach relies on statistical data derived from extensive static analyses of software repositories, encompassing test suites. These statistics pertain to the most commonly used JUnit assertions for each return type associated with a given method. Notably, we have gathered statistics for the return types of a significant portion of standard Java library methods. Consequently, when tasked with generating (additional) test cases for an application, the module crafts personalised test case templates for each method. These templates are curated by selecting the assertion that statistically aligns most closely with the return type of the method to be invoked.

The proposed framework offers developers a versatile and user-friendly tool for enhancing and optimising their source code. In contrast to other approaches in the current body of literature, our refactoring unit presents a range of refactoring techniques that yield numerous benefits, including enhanced performance, improved code readability, and increased code quality. The integration of diverse refactoring

approaches within a single tool empowers developers with a consolidated solution, eliminating the need to configure and navigate various distinct tools.

The testing unit provides the capability to validate the alterations made by the refactoring unit or to create a new test suite if necessary. The test generation module for methods facilitates the generation of test templates for methods that have undergone refactoring via the refactoring unit. This not only delivers developers a comprehensive package of refactored code but also provides associated tests to verify its correctness. Furthermore, developers have the flexibility to fully customise the generated code to align with their specific requirements and characteristics.

The framework is modular, allowing each module to function as a standalone library or be employed as an integrated unit. This empowers developers to make informed decisions about which modules to utilise based on their unique needs and software specifications. For instance, if an application already has an existing test suite, developers can opt to exclusively employ the refactoring framework and validate the transformations using their current test suite. Another scenario is when an application already incorporates Java Stream APIs, rendering the module for transforming loops to streams unnecessary.

1.4 Organisation

The rest of this dissertation is organised as follows: In Chapter 2, the existing state of the art is explored, alongside a comprehensive comparison with the approaches detailed in each module. Chapter 3 delves into the *Methods parallelisation* module, with the first segment focusing on data dependency analysis, followed by an in-depth examination of the tool designed to identify and transform methods from

a sequential to a parallel paradigm. Chapter 4 unfolds in two parts: the initial part introduces the *Loop to Stream* module, while the latter part delves into the *Matching Algorithm* module. Chapter 5 is dedicated to the *Testing Unit*, encompassing two modules. The chapter initiates with a description of the class test generation, followed by the presentation of the methods test generation. Chapter 6 accentuates the array of experiments executed to validate the correctness and efficacy of the proposed approaches. Each module has been rigorously assessed in real-world scenarios, with all results meticulously presented and discussed. Ultimately, Chapter 7 draws together the conclusions derived from the preceding chapters.

Chapter 2

Related Works

This dissertation delves into multiple facets of software engineering, with a particular focus on streamlining developers' tasks through the automation of various refactoring and testing techniques. While numerous integrated development environments (IDEs) like Eclipse, IntelliJ, and Visual Studio offer hints for applying refactoring during code development, they typically feature the most commonplace and straightforward refactorings. These include actions such as removing deprecated functions, relocating methods, and substituting temporary variables with queries.

One widely-used refactoring tool, SonarLint, is an open-source IDE plugin designed to perform real-time analysis of source code during coding[1]. It supplies a default rule set for identifying vulnerabilities, errors, and bugs in the code. Developers have the flexibility to craft their custom rules and integrate them with the default ones. Despite offering certain refactoring capabilities, such as code clone detection, its primary emphasis lies in pinpointing vulnerabilities like potential stack overflow errors and null pointer exceptions.

To execute more intricate refactorings, certain prerequisites must be met to prevent unintended alterations in the system's behaviour. For instance, transforming

[1]https://docs.sonarsource.com/sonarlint/intellij/

a for loop into a Java stream necessitates a series of checks to confirm that the loop meets specific criteria, including the condition that all variables within it are locally scoped and not defined beyond the loop's boundaries. Another scenario involves converting sequential code into a parallel one, necessitating a thorough examination of data dependencies among statements before proceeding with the refactoring.

Analysing code for potential refactoring can pose considerable challenges, often leaving developers uncertain about its suitability for improvement. The cutting-edge landscape presents a range of methodologies designed to automatically enact distinct refactorings across various domains. In Section 2.1, we explore several approaches dedicated to transforming sequential programs into concurrent ones. Section 2.2 delves into various works that revolve around the refactoring of loops into streams. Section 2.3 provides insights into approaches employed for identifying algorithms within source code. Lastly, Section 2.4 highlights methodologies tailored for automated testing.

2.1 Concurrent Refactoring

Many approaches provide tools to ease developers' work to achieve parallelism in their program. In [67], a new Java library for parallel programming is presented; these APIs provide many features to ensure multicore parallel programming, cluster parallel programming, GPU accelerated parallel programming and big data parallel programming. Developers should manually integrate its code with the appropriate construct to apply parallelism. Moreover, an approach that makes use of annotations to guide parallelism is presented in [144]; the author's custom annotations for C guide the tool to determine which code fragments could be run in parallel. In addition,

in [52] the author presents a programming model to ensure parallelism for C/C++ programs by taking advantage of GPU hardware; similarly, in [57], the authors built a programming model that is portable across different hardware architectures. These techniques provide parallel features to developers if they know which code snippet is prone to parallelism. Our proposal analyses the entire source code, identifying and refactoring code fragments that are suitable for concurrency. This frees the developer from the identification of proper statements, which could be time consuming and error-prone since a manual data dependence analysis could miss some dependencies, leading to an expected behaviour of the system.

Many papers have proposed automated tools designed to efficiently refactor sequential code into its parallel version. In [21], the authors outline an automatic approach to refactor sequential programs into parallel procedures and validate these changes with appropriate test suites using combinatorial testing [14]. They use the JSCAN tool[38] to identify a pair of statements within the same method that could be executed in parallel. The authors use Bernstein conditions [11] to find parallelisation opportunities. Once identified all the pairs of feasible parallel statements, they use ASM [16], a tool for manipulating Java bytecode, to inspect the source code and select the identified statements; then, the statements are moved in two different methods and their instructions are replaced with the method call created injected with the thread call to execute in parallel the two instructions. Additionally, in [20], the authors present an automatic approach to apply parallelism using the mutation testing technique. A mutant operator is defined using MUSIC [105], a mutation testing tool that allows developers to define and operate with custom mutant operators, hence a mutant operator is defined that will take as input a portion of code and it encapsulates it in a proper thread of execution. To properly identify the right

group of statements, a set of precondition is defined to check whether the set of statements selected is feasible for parallelisation; finally, the code is mutated to its parallel version. A test case is automatically generated to validate the mutation. The MUSIC tool produces a source code for each mutant operator identified, leading to a fragmented code across multiple files, therefore the developer should manually merge all the files to rebuild the code.

Our work distinguishes itself from these approaches in two fundamental ways. Firstly, they establish their data dependence analysis to determine whether two distinct statements share the same data, whereas our study delves deeper into the analysis, scrutinising both internal and external method calls as well as variables. This comprehensive examination ensures that method calls do not share any global state. Secondly, our approach applies a less intrusive transformation. In fact, we perform the transformation directly on the same source code without the need to relocate or insert numerous lines of code within the method. The only additional statement is the one employed for synchronisation, which amounts to just a single line. The code produced by our approach is ready for compilation and execution without necessitating further efforts from the developer.

In [64, 34], the authors presented two approaches to apply Atomic refactoring and Collection refactoring [33] to refactor synchronised statements. They proposed the following transformations: converting Int to AtomicInteger, Long to AtomicLong, HashMap to ConcurrentHashMap, WeakHashMap to ConcurrentWeakHashMap, and HashSet to ConcurrentHashSet. The authors focused on modernising existing parallel code by using the new libraries provided since Java 5. They carried out an effectiveness test to check the correctness of their modifications. In addition,

Dig et al. [34] proposed a refactoring approach changing sequential recursive algorithms into a parallel version using ForkJoinTask; they assessed popular recursive algorithms to show the benefits in execution time. Moreover, in [152], the authors presented a tool to substitute the Thread class with the Executor class to allow the use of a thread pool at runtime.

Conversely, our approach proposes several innovative aspects: (i) the application of a concrete refactoring opportunity from sequential to parallel, injecting new threads into the execution; in comparison, the transformations discussed by the aforementioned approaches just update some class types; (ii) the transformation shown by our approach is definitely less invasive than the one proposed with the ForkJoinTask; indeed, our approach requires the update of the instruction that calls a method with the use of a CompletableFuture and the addition of synchronisation statements with the join() call; (iii) the applicability of our approach accepts all method calls that meet the shown preconditions, while the aforementioned approaches are relevant for recursive algorithms, primitive variables, some collections, and Java synchronised blocks.

Another refactoring approach was presented in [120], where the authors proposed a Lock refactoring approach to automatically refactor built-in monitor locks for Java's synchronised blocks, with the locks provided by the `java.util.concurrent.locks` library: ReentrantLock and ReadWriteLock types. An analysis was performed to check whether the transformations preserved the behaviour of the application and if the updated locks guaranteed a performance boost. A similar approach was presented in [156, 152], where a tool was developed to automatically transform synchronised locks to reentrant locks. In [154], the authors presented an automated approach to convert a synchronised statement lock into a StampedLock. In [155],

the authors proposed a prototype to automatically convert a coarse-grained lock into a fine-grained lock to reduce lock contention and, hence, improve performance and scalability. These approaches focus on modernising and optimising existing parallel code; hence, the developer has to decide which part of the code should be run in parallel. Differently, our approach takes a sequential code as input and finds and introduces parallel constructs to boost performance and scalability.

A practical eclipse-based tool was presented in [148], which replaced the global mutable state with a thread-local state, and introduced a thread to run the refactored code in parallel. The aim of the tool is to reduce the number of executions that share the same input and, hence, increase the parallelisation opportunities. This approach involves invasive changes in the source code since global states are removed/moved, and the boilerplate code is inserted to create a thread. Conversely, our approach fits the code analysed, inserting the synchronisation in the proper position, without moving fields or variables; in addition, as stated before, the use of CompletableFuture widely reduces the number of instructions required to handle threads.

In [69], the authors presented an approach to analyse Java streams; their proposed tool statically analyses a stream pipeline and verifies whether to run the stream in parallel or not. This approach is strictly related to Java streams, while our tool covers a wider set of possible optimisations, since any method call could be evaluated for parallel execution. Other examples of refactoring activities for specific instructions are available in the literature: in [35], Java arrays and their loops were refactored to ParallelArray by using anonymous classes; in [71], an optimised compiler was proposed for the automatic parallelisation of loops; in [83], a refactoring tool for the X10 programming language was presented to introduce additional

concurrency within loops.

In [67, 57], the authors proposed two different libraries, for Java and C++, respectively. These APIs provide many features to ensure multicore parallel programming, cluster parallel programming, GPU-accelerated, and big data parallel programming. Developers can manually integrate code with these features to change it from sequential to parallel. However, one of the main difficulties that developers face when developing parallel applications is to understand which code fragment can be suitable for parallel execution [1] and which concurrent APIs could better fit a particular instruction [34]; instead, our proposal aims to solve these issues automatically. Indeed, the needed statements were selected according to an accurate analysis and a refactored parallel version generated via appropriate APIs.

Many approaches focus on the optimisation of the compiled code to achieve parallelism [70, 68, 37]. These approaches automatically analyse executable code to identify coarse grain tasks, such as the iteration of a large loop, and execute them in parallel. These approaches prove that the performance gain is considerable; however, the optimisation process is not visible to the developer. We propose a tool that shows the changes made to the source code, providing a clear view of which sections are selected and how they are properly refactored to achieve parallelism; moreover, the generated code is available to developers, who can add, modify, or remove it according to their will.

Asynchronous programming is widely used in Android applications, because of UI access and I/O operations [32]. In [74], a tool was proposed to automatically detect long-running operations and refactor them into asynchronous operations. Moreover, in [98], a tool was described to identify the improper use of asynchronous constructs and change them. Unlike our approach, the previously mentioned approach focused

on analysing code that was already parallel, aiming to identify defects. Moreover, JavaScript ecosystems provide synchronous and asynchronous calls to handle several I/O operations. In [49], the authors proposed a refactoring approach to assist developers in transforming operations from synchronous to asynchronous. Our approach is more comprehensive, as it is not just focused on I/O operations, which can be handled as well as other instructions.

Arteca et al. [6] presented an approach to reorder asynchronous calls to be executed as early as possible, yielding significant performance benefits. For this approach, the input code is parallel, and the developer chooses the parts that can run in parallel, unlike our more automated approach.

2.2 Loops Refactoring

Lambda expressions are used as parameters in most Java stream APIs, they are a key construct for functional programming and can be used for different tasks. E.g. in [140], the authors proposed an approach to ensure clone refactoring using lambda expressions. For this, it analyses whether a pair of clone fragments can be refactored [139]. Once it was ensured that two fragments can be unified in a common function, an appropriate functional interface containing the common method was created. As they employ lambda expressions to merge code fragments with similar behaviour in a common functional interface, their work differs from ours that instead uses lambda expressions to replace traditional for loops with proper expressions.

In [112], three templates have been defined to characterise loops [8]: accumulating loops, searching loops and selecting loops. A loop is frequently employed to aggregate specific elements from a collection into a singular value through the

application of various accumulation operators, such as addition, multiplication, and concatenation. This particular usage of loops is encapsulated by the Accumulating pattern. Typically, the type of the resulting value, in other words, the accumulated value, aligns with the type of the elements within the collection that the loop is iterating over. The loop's intended function is established by referencing the accumulated value, the criteria for element selection, and the iterator used to access the collection's elements; loops are frequently employed for element retrieval within a collection, such as locating the maximum value in an array of numbers. The Searching pattern encapsulates this specific utilisation of loops, where the loop iterates through collection elements to find a particular element or seek one that fulfils specific criteria. Typically, the outcome of such a loop is the identified element. However, other potential outcomes include the position or index of the located element and a flag that signifies whether the element has been found or not. The intended purpose of the loop is established by referencing the search criteria and the iterator employed to access the collection's elements; lastly, loops are commonly employed to extract or filter specific elements from a collection, and the Selecting pattern documents this particular application of while loops. The element type of the resulting collection matches that of the original input collection. The loop's intended purpose is delineated by examining the input and output collections, along with their respective iterators, and the criteria governing the selection of elements. Iterators serve as the means to access and store collection elements. Loops are analysed and labelled according to the three templates. The proposed approach does not provide any transformation of the source code, indeed they simply discuss what are the difficulties in trying to transform a loop into a stream pipeline.

The definition of these three templates could help the analysis for the transformation, trying to define a custom transformation for each specific pattern. However, loops within the same pattern could have different characteristics, leading to different types of transformations; e.g., the accumulating pattern could be identified with a stream pipeline whose final operation could be a count, or a reduce, or even a max/min for numeric Streams. Ultimately, the definition of these patterns has more of an educational and clarifying purpose instead of giving hints for possible transformations.

In our approach we propose a concrete and effective way to transform the loops into appropriate Stream pipelines. The templates defined by our approach are all presented with a corresponding transformation, indeed if a loop matches one of the patterns, it can be automatically converted to a stream pipeline following the template guidelines.

An approach and tool to refactor loops to stream code was proposed in [45, 56]. The approach is structured as follows: firstly, a loop is checked on a set of preconditions to ensure whether it can be refactored to a stream pipeline without violating the java language constraints; hence, the code of the loop is broken in a set of potential operations in order to annotate the accesses to variables; accordingly, the operations are merged maintaining the access to the needed references; finally, the operations are chained and the pipeline is formed. The main proposed categories to transform loops into streams are: one that comprises a condition to check whether an element having some property is in a list; one that generates a new list; and one updating an accumulator after evaluating a condition.

For the first category the only return value admitted is boolean, indeed they have defined a precondition (precondition **P5** at page 6 of [56]), which literally states:

"The body of the initial for loop does not contain more than one return statement. LambdaFicator can deal with loops with only one return statement as long as they return a boolean literal and LambdaFicator can infer that they can be refactored to an anyMatch or noneMatch operation." [56]. This precondition is partially true, since the Stream API provides the *findAny()* and *findFirst()* methods which returns an *Optional* value which encapsulated type is the type of the stream; accordingly, for loops with a non boolean *return* statement can be refactored with one of the two above methods. In order to properly select the right method between the two, a further analysis is needed to check whether the return needs the first element encountered in the stream or any one. Moreover, a for loop with two or more return statements could be refactored using a new stream pipeline for each return statement encountered, indeed hence removing the limit proposed by their approach.

For the second category the refactoring proposed still uses the imperative style (i.e. operation add() on a list, instead of collect() on a stream). In the example 3 in figure 4 [56], they show a for loop that after two if statement, it adds an element to a list of *String*; the refactoring proposed by their approach is to use the *forEach* method provided by the Stream API to refactor the loop, keeping the if statement and the add like the original for loop. However, the use of the *forEach* construct has strong similarities with the original for loop, indeed this refactoring has just partially refactored the loop.

In our approach, we have identified five further categories, and the most similar among these is more general than the one previously proposed (in our first template we handle any type of return value). Hence, thanks to our identified templates and transformations many more loops could be refactored and take the advantages of functional programming. An in-depth comparison with the examples shown in [56]

is discussed in section 6.3.1.

One of the advantages of streams is the possibility to easily make it parallel. However, several factors should be considered that could negatively influence parallel execution. In [109], a tool using functional programming was proposed to ensure deterministic parallel dataflow computations that are lock-free and linearisable. In [132], an approach was proposed to analyse accesses in stream-based code to check if they are thread safe. In [69], stream operations were analysed in order to determine whether they could be parallel. A set of conditions were defined to ensure that the mutation to parallel operations improves performance. They assume that the original code analysed uses streams. This work can be integrated with our approach, therefore we can refactor loops into accurate stream pipelines, hence their approach can check whether the generated stream can be run in parallel or not. The combination of these two works would bring both an improvement in the readability and the structure of the code, and a boost of the performance of the application if the generated stream will be optimised for a parallel execution. In [24], an approach was proposed to check the proper use of classes. The above approaches are complementary to ours, and it could be advantageous to combine them with ours, in order to gain parallelism for code not using stream APIs. In [12], an approach was proposed to extend Stream APIs with new operations, without changing the library code, in order to achieve high performance and highly-optimised computations. They have implemented an alternative stream library for Java.

2.3 Matching Algorithm

Algorithm recognition has been tackled by several approaches in the state of the art, both for the software engineering industry and for academic settings. In [123], the authors present a solution for automatic algorithm recognition using machine-learning techniques; they extract a dataset of source code containing algorithms, then a feature extraction is carried out to collect all the characteristic data (e.g. count-vars, operators, constructs etc.), finally a tag updating is done to remove redundant tags. They have trained the dataset and built a group of classifiers to identify the algorithm. In order to add a new algorithm or category of algorithms to the classification, all the previous steps have to be executed again in order to properly train the new dataset, which it could be time consuming; conversely, since our approach uses static analysis to match templates, a new template can be added to the collection without the need for further operations. The approach is proposed for an educational environment, where the codes submitted by the students are analysed to check if the approach is able to recognise a specific algorithm. However, the presented approach uses a "black-box" classifier to perform the identification, hiding the entire recognition process to students or developers. Conversely, our approach highlights all the steps followed, giving a clear view of the entire process.

In [134], the authors propose an algorithm recognition method that detects sorting algorithmic schemas; these schemas consist in a set of loops, features, operations etc. They present two different contributions. Firstly, a unified approach that initiates by extracting the models associated with the target algorithm. Subsequently, it calculates software metrics and various features by leveraging these schemas for classification purposes. This integrated approach segregates application code, which is code unrelated to the target algorithms, from the algorithmic code. Consequently,

it exclusively refines the algorithmic code, ensuring that irrelevant code does not influence the computation of features and indicators. Secondly, they address the utilisation of student-developed algorithm implementations in constructing a classification tree. They encompass the inclusion of challenging solutions created by students. This approach facilitates instructors in recognising these problematic solutions, enabling them to provide feedback. Simultaneously, it empowers students to gain insights into their misconceptions and receive assistance from the system when they encounter difficulties during their implementations.

Another approach discussing sorting algorithms is presented in [135, 133], where numerical (number of blocks, number of loops etc.) and descriptive (iterative, recursive) characteristics are extracted from the source code. The recognition process involves two key aspects: firstly, the calculation of the frequency of numerical features within an algorithm, and secondly, the examination of the descriptive attributes of that algorithm. When a new algorithm is submitted, it begins by tabulating its numerical features and scrutinising its descriptive attributes. Subsequently, a comparative analysis is conducted on this information against the corresponding data from algorithms stored in the database. If a match is detected between the characteristics of the algorithm to be recognised and one retrieved from the database, the recognised algorithm is categorised under the type of the matched algorithm, and its information is then recorded within the databases. A C4.5 decision tree classifier is builded to guide the detection process.

These approaches focus on sorting algorithms under some assumptions such as algorithms are expected to be implemented in a well-established way, e.g. quicksort algorithm should be implemented in a recursive way since it is more common.

Moreover, they define a set of characteristics that are mostly related to sorting algorithms, indeed their approach is suitable just for sorting algorithms, in particular just five sorting algorithms. Furthermore, since the approach is mostly based on the numerical characteristics of the source code, it's highly sensitive to differences in terms of number of statements, statements types etc.

In contrast, our approach can identify a wider spectrum of algorithms since the static analysis can be performed to any Java source code, and we consider different versions of the same algorithm to increase the identification ability. Moreover, our approach is able to identify a source code with a specific algorithm even if there are differences in terms of numerical characteristics because the similarity score computed takes into account both numerical and descriptive characteristics, without depending exclusively on one of them in particular.

Many tools have been presented to measure source code similarity. Most of these approaches address problems such as code clone detection, software licensing violation and software plagiarism [111]. The Levenshtein distance is often used for clone detection. In [126], a hybrid technique is presented where source code is lexically analysed to detect and extract sub-blocks, then similar blocks are grouped and hashed, finally the Levenshtein similarity and the Cosine Similarity are used to compute similarity between blocks and find Type-3 clones. In [5], the authors propose a cross-language clone detector for C, C++ and Java, the input code is tokenized to obtain the keywords of the corresponding language, then these keywords are compared with the Levenshtein distance and finally the clone types are classified based on similarity of keywords match. In [63], the authors present a tool to detect clones of a faulty code fragment, a Normalised Compression Distance is defined to detect duplicate code fragments. The above said approaches use the Levenshtein distance

to detect code clone fragments, whereas our proposal defines a tailored distance to match methods with algorithm templates in order to achieve algorithm recognition. Furthermore, these tools are sensitive to differences in term of statements between code fragments, because the presence or absence of multiple statements can lead to a misidentification of a code similarity; whereas our approach proposes a similarity score which, despite the statements differences, is able to define a matching grade for all templates, where the highest one is the most similar to the implemented algorithm.

Another difference between our approach and these clone detection algorithms is the level of abstraction used to characterise the statements. As illustrated in figure 1, page 3 in [126], the first step performs a lexical analysis to remove information like variable names from the source code; this differs from our approach because they keep the variables types and the names of methods, whereas we extract just the type of the statement analysed.

2.4 Automatic Test Generation

Automatic test generation has been tackled by several approaches in the state of the art, since developers and companies can benefit from it during software development. Random, search-based and symbolic execution are the main approaches employed.

Starting from random generation, Randoop [99], with its current optimisations [77, 94], is one of the first and most popular random testing generators. It generates sequence of method calls by incrementally finding and executing random input values; feedbacks are collected by executions to properly tune the next generations to avoid

generating redundant and illegal inputs. JPF-Doop [36] is a tool that combines random testing with concolic execution, a formal technique that interleaves symbolic and concrete execution: it defines symbolic path constraints by repeatedly executing the application, to reduce the search space over input parameters. Random generation aims at producing many test cases, where the input values are chosen randomly. Our proposed approach aims at generating accurate test cases that are tailored to tested classes and methods, based on the method itself. Moreover, the generation is based on optimising the test suite, covering classes and methods that have not been previously tested, reducing redundant test cases or even impractical ones.

Search-based approaches handle test generation as a search problem: actual solutions are found by using a heuristic approach guided by a fitness function. Evosuite [46] is a search-based generation tool that uses an evolutionary algorithm called DynaMOSA [101] to optimise the generation of test cases based on code coverage. A many-objective optimisation problem is formulated where one fitness function is defined for each target (e.g a statement) to cover. The multiobjective algorithm performs test generation by executing the target system several times to check the fitness of each test sequence; indeed, the greater the number of runs, the greater the coverage of the generated test suite. SUSHI [15] is a combination of search-based and symbolic execution approaches that symbolically executes the target application to define the best paths covering the structure of the inputs, then the search-based process aims at optimising paths by generating test cases that correspond to each path retrieved. In [119], the authors propose a search-based approach for test-data generation, where static analysis determines the set of methods reaching the path of the test target. The above approaches need executing the system several times to obtain a level of coverage that suffices.

In contrast, our approach relies just on static analysis to generate the templates, hence our tool is executed once, and its execution time depends on the number of lines of code analysed. Furthermore, we look for methods that have not been previously covered, and generate tests for them, increasing code coverage. Although the above approaches can produce test cases that automatically run on an application, they cannot ensure a very high code coverage and are computationally demanding during the test generation. Moreover, generated tests are often redundant when a high code coverage has been set as a parameter during generation; therefore running such generated tests can be also computationally costly. Our approach is very quick to find methods that need to be tested, as well as to generate test case templates. Moreover, the generated templates have a clean code that can be easily read by developers and customised when needed.

COFFEe [14] is a comprehensive framework for Combinatorial Interaction Testing and Fault Characterisation. It is based on Junit5 and can model input parameters, generate and execute tests and make a fault characterisation. To properly configure the framework, the developer should provide the model for generating the combinations and a unit test describing the steps to be followed for every given parameter combination. Our approach can provide the artefact needed for the execution of this tool since we generate tests for uncovered classes and methods.

In [95], test cases are generated in order to cover as many paths as possible. Moreover, the number of test cases are reduced by finding paths covered by more than one test case. Their approach is computationally expensive, compared to ours, since it requires the execution of each test case. Moreover, it could be advantageously complemented by ours, in order to reduce computation time: along the lines we introduced, one could select pairs of classes with similar behaviour, pass only one

class of the pair to the tool of [95] to generate test cases, and then enlarge these. Other approaches have been proposed to reduce the number of test cases, e.g. once they have been generated automatically, in order to reduce execution time, or when considering that code coverage has not been increased [50, 65]. In our approach, by analysing the test code and the application code, we can reveal which classes have not been tested and generate proper test cases selectively.

Chapter 3

Automatic Parallelisation of Methods Calls

To fully harness the capabilities of modern processors, which often feature multiple cores, achieving parallel execution is a paramount objective. While developers can attain finely tuned parallelism, the effort required becomes overwhelming for very large applications due to the intricacies of their components and control flow [7, 125]. Nonetheless, numerous software systems can enhance their execution speed by initiating new threads at various points. However, to ensure proper execution, access to shared data among these threads must be meticulously protected.

Concurrent programming, on one hand, can significantly boost performance when handling extensive data sets. On the other hand, developers encounter various challenges, including ensuring thread safety, program correctness, and optimising performance [1, 108]. To maintain program integrity, multiple threads must be appropriately synchronised to prevent uncontrolled concurrent access to shared data. Additionally, the overhead associated with initiating and managing several threads needs to be scrutinised to ensure a net performance gain [43].

The automatic transformation of sequential programs into parallel ones remains

an open issue, as it necessitates an in-depth analysis of data dependencies, which can be notably complex. Furthermore, the transformation into a parallel version should be guided by strategies that have the potential to yield performance improvements once the parallel version is operational.

The subsequent sections are organised as follows: Section 3.1 outlines an API for analysing data dependencies among variables and methods, while Section 3.2 presents an automated approach to refactor method calls from the sequential version into a parallel one.

3.1 An API for Analysing and Classifying Data Dependence in View of Parallelism

Parallel computing is a powerful technique that leverages modern processors to enhance software performance. However, incorporating parallelism into large legacy applications can be an arduous challenge for developers. One of the primary difficulties lies in identifying shared data that require protection against concurrent access. According to data dependence theory, two statements exhibit data flow dependence when the output set (i.e., the variables being written) of the first statement contains data that are also present in the input set (i.e., the variables being read) of the second statement [84, 104, 149].

To address this challenge, we propose an automated approach for identifying data dependencies within methods, which proves invaluable for developers tasked with introducing statements to enable parallel execution. Our method involves conducting data dependence analysis on all statements within a method. Additionally, we determine whether each variable used within the method has a local or external

scope and whether it belongs to the output set or the input set. This information helps identify the operations (read or write) a method performs on each variable.

When executing multiple methods in parallel, special attention is required for operations performed on variables with external scope. Failure to properly synchronise such operations could lead to race conditions. Automatic parallelisation of large-grained portions of code remains a complex challenge, demanding in-depth data dependence analysis. While several approaches have been proposed to automatically transform sequential code fragments into appropriate parallel versions while ensuring correctness [20, 43, 68, 81], it remains a significant challenge.

In our approach, we provide a high-level library that allows developers to assess opportunities for parallelism by analysing the usage of variables with external scope across multiple methods. Our analysis involves parsing the static source code to extract information related to classes and methods, including variables, parameters, method calls, and attributes. Subsequently, our library offers an intuitive API for developers to identify data dependence (i.e., dependencies on external variables) and control dependence (i.e., dependencies on called methods) for each method. Moreover, it categorises methods to facilitate the evaluation of safe parallel execution.

Armed with this information, developers can then make informed code modifications, such as reducing external variable dependencies or implementing access safeguards when executing selected methods in parallel

This Section is organised as follows. Section 3.1.1 describes the proposed classification of methods based on data and control dependence analysis performed by our tool. Section 3.1.2 presents the proposed API that developers can use to find opportunities for parallelism. Finally, Section 6.1 shows our experiments to several open source projects and their results.

3.1.1 Classifying methods based on data dependence analysis

Our proposed approach conducts both data and control dependence analyses for each method. Subsequently, each method is categorised into one of three groups: stateless (indicating no usage of external data), read (denoting the use of external data exclusively in the input set), or write (signifying the presence of external data in the output set). This categorisation is based on a thorough examination of all the variables used and all the methods called within the code.

For variables, our tool determines whether they are local or external and whether they belong to the input set or the output set. The method is assigned a category based on the operations it performs on these variables and the categories of the methods it calls.

The initial step of the analysis involves inspecting all variables used in the statements within each method to identify external (non-local) variables. Subsequently, based on the operations performed, we determine whether they belong to the input or output set. We also identify the set containing all local variables, i.e., those with scope confined to the method's body. This encompasses all variables declared within the method and parameters of primitive data types. Notably, in Java, certain classes (e.g., Integer, Double, Long) wrap primitive types, but they are passed by value to method parameters; hence, we treat them as local.

Furthermore, we identify the set of external variables, i.e., those declared outside the method and accessible by other methods. Additional cases considered include: firstly, local variables holding elements of a collection accessed via an external variable, where changes to these local variables affect the collection; secondly, variables

referenced using 'this' or 'super' keywords, where 'this' refers to fields and methods of the class and 'super' to fields and methods of the parent class; thirdly, local variables with non-primitive types assigned the return value of a method call, which makes the instance a shared one.

This analysis yields two sets for each method: local variables and external variables. The latter is used to categorise the method based on the operations performed on them, with the category also influenced by the methods called.

To evaluate the global state effects of a method calling other methods, we must consider the effects of the called methods. If our analysis labels the called method as a write method, the caller method is likewise labelled as a write method, regardless of other operations. Similarly, if the called method is labelled as a read method, caller methods that would otherwise be stateless are categorised as read methods.

Figure 3.1 illustrates a state machine encompassing the three categories a method can belong to and the transitions between these tentative categories, which are temporarily assigned when not all statements have been analysed. This creates a priority hierarchy, with 'write' being the highest priority; once this state is entered due to a write operation, it cannot be exited regardless of other operations. 'Read' is the intermediate category, while 'stateless' has the lowest priority.

After determining the label for each called method, we check for any method calls within the method body with a higher priority label than the temporary one assigned. If found, the method is assigned the higher-priority label.

In cases where our tool lacks access to the source code for analysing libraries, we rely on developer-provided classifications. To facilitate this process, we offer: (i) a manually curated classification for the most commonly used methods in the standard Java library; (ii) a generated list of methods found within the analysed

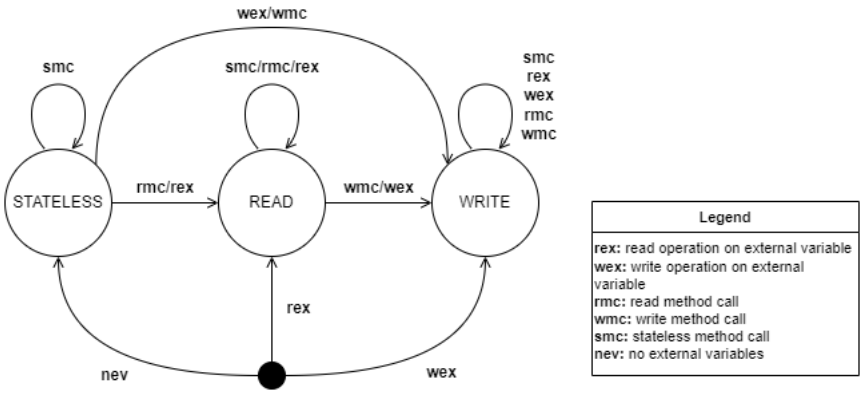


Figure 3.1: The state machine representing the three possible categories for a method

code.

The developer's responsibility is to decide whether each method should be categorised as 'write,' 'read,' or 'stateless'.

In Figure 3.1, we present an illustrative example of the analysis performed by our approach. The left portion of the figure displays the method under analysis, while the right side exhibits the output generated by our analysis. The analysis reveals that the method's input set is empty, while the output set (refer to 'External write variables') includes only the 'counter' variable. Consequently, the method is categorised as 'WRITE.'

Furthermore, the control dependence analysis identifies two method calls within this method. The first call is to 'Math.round()' from the 'java.lang' library, which our tool categorises as a 'STATELESS' method. The second call is to 'Example.logger()', a method within the same class as the one under analysis, classified by our tool as a 'READ' method.

To obtain the displayed output on the right side, you can utilise the provided API.

Specifically, the 'getMethod(String qualifiedName)' function within the 'MethodAnalyzer' class returns a 'ContainerMethod' instance that encapsulates the analysed method, along with all the extracted data. You can access category information, variables, method calls, and external methods by invoking the 'getCategory(),' 'getVariables(),' 'getMethodCalls(),' and 'getExternalMethods()' functions, respectively, on the 'ContainerMethod' instance.

Regarding parallelisation, since the output set ('External write variables') is not empty and includes the 'counter' variable, it should be locked when the 'foo()' method is executed in its own thread, as well as when another method that accesses the same variable runs concurrently. To identify all variables requiring a lock, you can simply find the intersection of the common external variables, adhering to Bernstein's conditions [11]. For locking purposes, you can use a data type like 'AtomicReference¡V¿' from the 'java.util.concurrent.atomic' package to automatically synchronise concurrent accesses. To initiate the execution of the 'foo()' method in its own thread, you can employ a construct like $'CompletableFuture.runAsync(()->o.foo(tmp))'$.

Listing 3.1: At the top, an example method *foo(Double tmp)*, at the bottom, all the data extracted by our library. Method's category is given by a write operation on the external variable *counter*

```
public void foo(Double tmp){
 Double bar = Math.round(tmp);
 counter += bar;
 logger("Counter updated with tmp value: " + tmp);
}

Qualified name: Example.foo
Category: WRITE
External write variables: [counter]
Read Method Calls: Example.logger
External read methods: java.lang.Math.round
```

3.1.2 API of the developed library

We have developed a Java library to facilitate the dependence analysis described above, providing developers with comprehensive insights extracted from their applications and categorisation of methods based on the dependence analysis. To accomplish this, we leveraged the Javaparser library [129] for parsing the source code and extracting essential data. Javaparser offers robust APIs for parsing source code and generating an Abstract Syntax Tree, granting developers the tools needed to navigate and modify the tree structure.

Our tool comprises seven classes, with five serving as the core for code parsing and dependence analysis, while the remaining two house the API accessible to developers for initiating the analysis and accessing the obtained data. To commence the analysis and gather methods classified according to dependence analysis, developers need to instantiate the 'MethodAnalyzer' class. Each analysed method is stored as an object

of the 'ContainerMethod' class, encapsulating all the extracted dependence data. The enumeration 'Category' defines the three possible categories that a method can fall into.

The *ContainerMethod* class encapsulates a method along with all the data collected during the dependence analysis. We provide developers with both the categorisation of each method, according to the previously defined three categories, and the data extracted during the analysis. Table 3.1 presents a list of available methods along with their descriptions. Each method identified during the analysis is uniquely identified by its qualified name, retrievable using the 'getQualifiedName()' method. In this context, the qualified name consists of the type followed by the method name. For instance, the qualified name of the 'length()' method in the 'String' Java class is 'java.lang.String.length'.

The method's category is obtainable through the 'getCategory()' method. Additionally, we offer several other methods to empower developers with a deeper understanding of the code. The 'getVariables()' and 'getExternalMethods()' methods return a hashmap where each key-value pair consists of the variable name or the qualified name of the external method as the key and its corresponding category as the value. For variables, the category is determined by the operation applied to them, with 'write' indicating data updates and 'read' indicating data retrieval.

Moreover, the 'getMethodCalls()' method returns a hashset containing the respective 'ContainerMethod' instances of all methods called within the current method's body. The 'getCallers()' method provides the 'ContainerMethod' instances of all methods that contain a call to the current method within their bodies. Finally, the 'getSharedVariables()' method offers a hashmap where keys represent the intersection of the output set of the current method with the output set of another method,

Table 3.1: Methods provided by the ContainerMethod class

Name	Description
getQualifiedName	Return the qualified name of the wrapped method
getCategory	Return the category of the method
getVariables	Return a map of external variables labelled as READ or WRITE
getExternalMethods	Return a map of external method calls labelled as STATELESS, READ or WRITE
getMethodCalls	Return the internal methods called by the wrapped one
getCallers	Return the methods that contain a method call of the wrapped method
getSharedVariables	Returns a map containing write variables shared among methods

as indicated in the value of the respective pair. Each pair in the map highlights data shared by two or more methods.

The *MethodAnalyzer* class serves as the entry point for our API, allowing you to initiate the analysis and gather all labelled methods. To instantiate this class, you need to provide the source path of the project to be analysed. This source path corresponds to the root of the package structure containing the source files. For a comprehensive list of methods offered by 'MethodAnalyzer,' please refer to Table 3.2.

To begin the analysis, call the 'startAnalysis()' method. This function parses the source code based on the provided source path. Subsequently, other methods within the class can be invoked. However, please note that invoking these methods before 'startAnalysis()' may result in empty values.

The first four *getter* methods return methods categorised according to the labels established through the earlier analysis. The 'getMethod(QualifiedName: String)' method returns the method specified by its qualified name.

The following five methods require further explanation. A method can contain multiple read and write operations on different external variables, as well as method calls to stateless, read, or write methods. As indicated by the state machine in Figure 3.1, the initially suggested category may change during the analysis of subsequent lines. When all lines have been considered, the method is assigned a final label. This

Table 3.2: Methods provided by the MethodAnalyzer class

Name	Description
startAnalysis	Starts the analysis for the specified directory (project root directory).
getSlessMethods	Return all methods labelled as STATELESS
getReadMethods	Return all methods labelled as READ
getWriteMethods	Return all methods labelled as WRITE
getAllMethods	Return all analysed methods
getMethod	Return a specified method
getVarMethods	Return all methods having at least an external variable
getMethCallMethods	Return all methods having at least a method call
getExternalMethods	Return all methods having at least an external method call
getMethExclMethods	Return all methods whose category is due to method calls only
getVarExclMethods	Return all methods whose category is due to external variables only
estimateWriteVars	Return a map containing all methods with n write variables

label is determined by the highest category of operations or method calls encountered within the method's body. For example, if a method includes a read operation on an external variable, its category will be updated from 'STATELESS' to 'READ.' If a subsequent method call is found within the same method, and the called method is categorised as 'WRITE,' the label of the analysed method will be updated to 'WRITE,' as method calls with higher priority categories take precedence over read operations.

Lastly, the 'estimateWriteVars()' method returns a map where keys range from 1 to 7, each corresponding to a list of methods. These keys represent the size of the output set for each method within the respective list. Therefore, methods with only one write variable are categorised under key 1, methods with two write variables under key 2, and so on. We provide this mapping to assist in understanding the synchronisation requirements for parallelising the method. The larger the size of the shared data, the greater the effort required to ensure proper synchronisation. All methods retrieved through this function are encapsulated within the 'ContainerMethod' class.

3.1.3 Summary

This study introduces a library that equips developers with a versatile API for conducting data dependence and control dependence analysis within software systems. Each method under examination is enriched with valuable data that developers can readily access via the provided API. Our data dependence analysis encompasses variables, while we also perform control dependence analysis.

The proposed API is designed to be user-friendly and high-level, simplifying its utilisation. It can be applied, for instance, to automatically assess the feasibility of parallel execution for a given method. Our custom library and API have been applied to analyse various open-source systems, ranging in code complexity. The results of these experiments have validated the precision of our method classifications, demonstrating our capability to accurately identify data dependencies among methods. Dependency analysis is a pivotal step towards enabling parallel execution of code, potentially enhancing software performance without introducing unforeseen issues.

Understanding the interplay between variables and methods is of primary importance when considering parallel execution of specific code segments. The subsequent phase of this module involves leveraging this analysis to pinpoint method calls that can be transformed into parallel versions, provided their context, data dependencies, and computational complexity align favourably with parallelisation goals.

3.2 An Automatic Transformer from Sequential to Parallel Java Code

Sequential programs can benefit from parallel execution to enhance their performance. When developing a parallel application, various techniques are employed to achieve the desired behaviour, such as identifying segments suitable for parallelism, synchronising access to shared data, and optimising performance. However, manually transforming a sequential application into a parallel one can be a daunting task due to several challenges. These include the need to inspect a large volume of code, the potential for errors resulting from inaccurate data dependence analysis, which can lead to unpredictable behaviour, and addressing imbalances in workload distribution between parallel threads.

The current state of the art offers some noteworthy approaches that can automatically convert applications into parallel versions [153]. However, many of these approaches have limitations. Some exclusively focus on recursive algorithms, which represent only a narrow category of algorithms [64, 34]. Others concentrate on specific data structures, like arrays and streams, limiting their applicability [35, 69]. In contrast, our proposed approach offers a broader spectrum of parallel execution opportunities. While prior approaches often focus solely on identifying specific categories of statements [74, 49], or refrain from introducing parallelism [148, 6, 120, 98, 156, 155]; our analysis is more comprehensive in scope.

Furthermore, most existing approaches rely on concurrent libraries that predate Java 8, whereas our work leverages the latest Java libraries for concurrent computations. This integration offers the added benefit of incorporating high-level API calls into the resulting source code, ensuring minimal overhead when initiating new

threads. Our fine-grained transformation only modifies a small portion of the initial source code, providing developers with a clear and readable new version. While libraries for parallel programming exist for Java and C++ [57, 67], developers must manually determine which code segments are amenable to parallelism and how APIs can be utilised. In contrast, our approach automatically generates a parallel version of the source code.

Several other approaches perform automatic parallel transformation directly in the executable code [37, 70, 68]. While these approaches achieve impressive performance gains, they hide the parallel version from the developer, limiting their ability to modify it to meet additional requirements or make improvements. In contrast, our approach generates the modified source code, making the transformations transparent and the code amenable to further changes.

We propose an automated approach that statically analyses source codes to look for fragments of code that can be safely run in parallel to provide performance gain. The approach and corresponding tool use control flow analysis, data dependence analysis, and a control flow graph to identify execution paths that can be safely run in parallel (safety is checked according to Bernstein's conditions [11]), showing that such paths require considerable computational efforts before synchronisation is needed, so that a performance gain is expected. Firstly, code is analysed to find sequences of statements that could run in parallel, while adhering to some syntactic constraints (e.g., blocks of code in the same conditional branch, etc.). Secondly, data dependence analysis and control dependence analysis are carried out to check data dependencies among statements that could run in parallel. Two statements have a data flow dependence when the output set (i.e., the set of variables written) of the first statement contains data that are in the input set (i.e., the set of variables

read) of the second statement [149, 84, 11]. Thirdly, a control flow graph (CFG) [3] is built, which represents all the paths that could be executed by a program. Such a CFG is then used to determine the two paths that might execute in parallel and to evaluate whether the statement numbers in each path are sufficiently large to possibly improve execution time. Finally, once a path that has passed all previous analysis steps has been found, the source code is refactored to run a new thread and insert synchronisation points where needed. The code of the new version is generated automatically for developers to further modify or simply run it.

Our primary original contributions are as follows. We present a comprehensive approach that (i) automatically transforms sequential Java source code into parallel code (in previous literature, the authors mainly tackled executable code); (ii) undertakes a detailed static analysis of the source code, comprising control flow, data dependence analysis, evaluation of Bernstein's conditions, and the assessment of the computational effort required; (iii) employs an a priori evaluation of the potential benefits of new threads by computing execution paths; and (iv) evaluates the benefits performed according to the context of the instructions that are potential candidates for parallel execution and the estimated computational effort required by such instructions. We performed several experiments to assess the benefits and correctness of our approach. We report on an experiment where the analysis steps were automatically executed on an application; the results of several tests executed on the refactored parallel version show that our transformation preserves correctness.

The subsequent sections are organised as follows: Section 3.2.1 introduces the general approach and provides a high-level algorithm for code analysis. Sections 3.2.2 and 3.2.3 delve into the static code analysis, elucidating method call context and data dependence. Section 3.2.4 details the construction of the CFG for the

analysed method and the evaluation of instructions for potential parallel execution. Finally, Section 3.2.5 sums up the proposed approach and concludes the module.

3.2.1 Proposed Approach

The proposed approach employs static analysis to extract data from the source code of the analysed software. Parsing activities rely on the JavaParser library, which serves as an automatic parser. It takes one or more .java files as input, generates an abstract syntax tree (AST) for each file, and offers functionality for various operations on ASTs, such as reading, inserting, deleting, and updating [129].

Algorithm 1 provides a high-level pseudo-code representation of our analysis. The procedure takes the source code of the analysed application as input, parses it, and returns instances of CompilationUnit for the application. A CompilationUnit represents a single Java file in the form of an AST. Subsequently, the method declarations are extracted from each CompilationUnit instance to create a list containing all the methods implemented in the source code. While a method body generally contains numerous instructions, our analysis focuses on method calls since they are the instructions evaluated for potential parallel execution.

Each method call undergoes analysis based on its context, data dependence, and the number of instructions it involves. All three aspects are covered in the following three sections. Once a method call successfully passes all the checks, it is transformed into a parallel version, and the method is updated with the appropriate synchronisation statement, based on the identified data-dependent statement. Finally, all compilation units containing at least one transformed method are written to new Java files.

Algorithm 1 The algorithm of the proposed approach

```
procedure TransformSequentialToParallel(Sc)
    compilationUnits ← parseAllPaths(Sc)
    for cu, compilationUnits do
        methods ← visitMethods(cu)
    end for
    for m, methods do
        methodCalls ← m.getMethodCalls()
        for mCall, methodCalls do
            if ContextAnalysis(mCall) then
                ddStatement ← DataDependenceAnalysis(m, mCall)
                if CFGAnalysis(m, mCall, ddStatement) then
                    mCall ← transformToParallel(mCall, ddStatement)
                end if
            end if
        end for
    end for
    printParallelCu()
end procedure
```

3.2.2 Method Call Analysis

Executing methods can consume significant time, especially if they contain a large number of instructions or encapsulate nested method calls. To enhance execution speed, certain methods can be executed in dedicated threads. We conducted three distinct analyses to ensure that our proposed automatic source transformation into a parallel version preserves the behaviour and correctness of the original version.

Firstly, we analyse the method call's context to assess the safety and suitability of inserting a parallel construct. Secondly, we examine data dependencies between concurrent instructions to prevent race conditions. Thirdly, we evaluate the number of instructions that threads would execute before requiring synchronisation. This evaluation helps determine whether the workload is balanced between threads and if there are enough instructions to achieve a performance gain.

The context of the method call is critically important when assessing the potential benefits of running the method call in parallel. The context refers to the statement in which the method call is located. This statement could either be the method call itself or a more complex statement containing it. In the former case, further analysis of the same statement is unnecessary since the method call is the only instruction in the statement. Therefore, we can directly check additional conditions to determine the feasibility and desirability of parallel execution. Conversely, in the latter cases where the method call is within another statement, there may be situations where parallel execution is unsuitable. In such cases, we avoid further analysis aimed at parallelisation.

Algorithm 2 outlines the steps performed when analysing the context of a method call. The context is extracted by identifying the ancestor of the method call.

Algorithm 2 The instructions performed by the context analysis

```
function ContextAnalysis(mCall)
    context ← getAncestor(mCall)
    suitableContexts ← defineSuitableContext()
    return suitableContexts.contains(context)
end function
```

The ancestor corresponds to the parent node on the AST (Abstract Syntax Tree) of the node containing the analysed method call. If the method call is within a more complex statement, the ancestor will be the instruction containing it. Otherwise, the ancestor will be the statement containing the block of instructions with the method call. For example, it could be the body of a method declaration or the body of a for statement. Once the ancestor is retrieved, it is compared to a predefined set of feasible contexts. The function returns true if the context of the method call passed as input is included in the feasible ones, false otherwise.

Listing 3.2 provides examples of method calls that are unsuitable for parallel execution. These include method calls that are part of the condition expression in if, while, and do constructs (examples 1 and 2 in Listing 3.2), as well as statements such as switch, for, throw, assert, and synchronised (example 3). In such cases, parallel execution is unsuitable because the method call's return value is immediately used to determine whether to execute the following statements. Similarly, method calls within a return statement are not advantageous for parallel execution since the need for the return value would cause the calling thread to wait for the result.

Listing 3.2: Method calls whose context is such that parallelisation is unsuitable. The method call is part of: (i) an *if* condition, (ii) a *while* condition, and (iii) a *for* loop iterable.

```
// Method call in a condition statement
1. if ( checkForFileAndPatternCollisions() ) {
    addError("File property collides with fileNamePattern. Aborting.");
    addError(MORE_INFO_PREFIX + COLLISION_URL);
    return;
}
// Method call in a while statement
2. while ( isRunning() != state) {
    runningCondition.await(delay, TimeUnit.MILLISECONDS);
}

// Method call passed as iterable for a foreach statement
3. for (StatusListener sl : sm.getCopyOfStatusListenerList() ) {
    if (!sl.isResetResistant()) {
        sm.remove(sl);
    }
}
```

Listing 3.3 shows six examples of method calls that can be transformed to run

in parallel. Line 1 shows a method call used as the value for an assignment expression; line 2 shows a method call used as the value for an assignment in a variable declaration expression; line 3 shows a method call chained with other calls; line 4 shows a method call passed as the argument for other method calls; line 5 shows a method call without any other instruction; and line 6 shows a method call passed as the argument for an object creation expression. In such cases, the result of the method call is not used to determine whether to execute the following statements. Indeed, our aim is to evaluate whether the whole statement (which could consist of several expressions) could run in a thread parallel to the thread that runs the following statements.

Listing 3.3: Method calls whose context is such that parallelisation is suitable. The method call is: (i) part of an *assign expression*, (ii) part of a *variable declaration expression*, (iii) chained with other method calls, (iv) passed as argument for a method call, (v) called without other instructions and (vi) passed as argument for an object creation expression.

```
// Assign Expression
1. le = makeLoggingEvent(aMessage, null);

// Variable Declaration Expression
2. BufferedReader in2 = gzFileToBufferedReader(file2);

// Chained method calls
3. context.getStatusManager().getCount();

// Method call passed as argument for another method call
4. buf.append(tp.getClassName());

// A simple method call
5. implicitModel.markAsSkipped();

// Method call passed as argument for an object creation
6. new Parser(tokenizer.tokenize());
```

3.2.3 Data Dependence Analysis

To ensure proper guarding of shared data among threads, we employed the methodology detailed in section 3.1, which offers a collection of APIs for extracting data dependence within methods. This approach scrutinises both variables and method calls within a method to establish its input set (i.e., the set of variables read) and output set (i.e., the set of variables written). Algorithm 3 outlines the steps executed during the data dependence analysis. Initially, the method call statement is

retrieved, allowing us to define the input and output sets of the statement. Subsequently, for each subsequent statement, we compute the intersection between sets. If the intersection contains at least one element, the statement is identified as the data-dependent statement requiring synchronisation; otherwise, we proceed to the next statement. In the absence of a data-dependent statement, the last statement of the method is returned.

Algorithm 3 The instructions performed by the data dependence analysis

```
function DataDependenceAnalysis(m, mCall, ddStatement)
    s1 ← getStatement(m, mCall)
    inSetS1 ← getInputSet(s1)
    outSetS1 ← getOutputSet(s1)
    index ← indexOf(m, s1)
    for i ← index + 1, m.getStatements().length() − 1 do
        s2 ← m.getStatements().get(i)
        inSetS2 ← getInputSet(s2)
        outSetS2 ← getOutputSet(s2)
        if checkIntersections(inSetS1, outSetS1, inSetS2, outSetS2) then
            return s2
        end if
    end for
    lastStatement ← getLastStatement(m)
    return lastStatement
end function
```

Listing 3.3 illustrates various statements. In the case of lines involving assignments, such as lines 1 and 2, the output set encompasses the variable being assigned and any variables modified by the called method, as determined by analysing the called method itself. For example, in line 1, the output set comprises the variable *le*, while in line 2, it includes the variable *in2*. We will assume that no variables are written in the called methods for the sake of this explanation.

On the other hand, the input set consists of all variables passed to the called method and those read within the called method. Thus, in line 1, the input set

includes the variable *aMessage*, and in line 2, it contains the variable *file2*. Again, let's assume that no variables are read in the called methods for this illustration.

In cases where methods are called on an instance using a variable that holds the reference, like the calls in lines 3, 4, 5, and 6, the variable holding the reference is part of the input set. This is because when executing the method call, such a variable is read to properly dispatch the message. As a result, variables *buf* and *tp* constitute the input sets for line 4, while variables *implicitModel* and *tokenizer* are the input sets for lines 5 and 6, respectively. For lines 3, 4, 5, and 6, their respective output sets will include all the variables written in the called method, in addition to the variables used to call the method (e.g., *context* for line 3, *tp* and *buf* for line 4, etc.), if the called method writes some of the attributes within the same instance.

Once these two sets are defined for the relevant statement, the analysis is reiterated for each subsequent statement to identify any data dependencies. Listing 3.4 provides an example of the data dependence analysis. In this instance, for line 74, we are analysing the method *getRandomlyNamedLoggerContextVO()*, and the method call is located within a variable declaration expression. As previously mentioned, we consider this instruction feasible. Subsequently, commencing from line 75, the data dependence analysis is executed to pinpoint a data-dependent statement, which is found on line 79 where the variable *lcVO* is assigned to the *e.loggerContextVO* field. This data-dependent statement signifies the point at which the main thread will need to wait for the forked thread to complete before resuming its execution.

Listing 3.4: Data dependence analysis: *lcV0* is declared at line 74 (the statement has it in its output set) and is used at line 79 (the statement has it in its input set), hence the two statements are data dependent as the intersection between their output and input sets is not empty.

```
74  LoggerContextV0 lcV0 = corpusModel.getRandomlyNamedLoggerContextV0();
75  PubLoggingEventV0[] plevoArray = new PubLoggingEventV0[n];
76  for (int i = 0; i < n; i++) {
77      PubLoggingEventV0 e = new PubLoggingEventV0();
78      plevoArray[i] = e;
79      e.loggerContextV0 = lcV0;
80      e.timeStamp = corpusModel.getRandomTimeStamp();
    ....
}
```

3.2.4 Control Flow Graph Analysis

Once all data dependencies among statements have been identified, the analysis proceeds to pinpoint the potential paths that can be executed in parallel. These parallelisable paths are primarily identified by scrutinising the control flow graph (CFG), a graph-based representation that encompasses all possible program execution paths [3]. Subsequently, a comprehensive evaluation is conducted to assess whether introducing parallelism would indeed lead to performance enhancements. This final aspect of the analysis determines whether the workload allocated to both threads is both balanced and substantial enough to justify the overhead associated with starting a new thread.

Numerous studies have underscored the significance of the number of statements when evaluating the computational complexity of an algorithm [54, 128]. Our approach leverages control flow graph theory to inspect execution paths and evaluate

the number of instructions encompassed by each path. For the CFG analysis, we employed the JGraphT library (https://jgrapht.org, accessed on July 21, 2023). Algorithm 4 outlines the steps undertaken during CFG analysis.

To begin, the CFG is constructed, and special attention is given to conditional and loop branches to ensure the graph's acyclicity. Next, the nodes containing the method call that could be executed in parallel and the data-dependent statement (as previously defined) are gathered. Finally, the paths amenable to parallel execution are computed and compared. The function returns true if these two paths are deemed suitable for parallelisation and false otherwise.

Algorithm 4 The instructions performed by the control flow graph analysis

function CfgAnalysis(m, mCall, ddStatement)
  $cfg \leftarrow$ BuildCfg(m)
  $cfg \leftarrow handleConditionalAndLoops(cfg)$
  $node_1 \leftarrow getGraphNode(cfg, m)$
  $node_2 \leftarrow getGraphNode(cfg, ddStatement)$
  return comparePaths($node_1, node_2$)
end function

A CFG represents the set of instructions that form a program and all the interactions that intervene. In a CFG, each node corresponds to a basic block, which is a line of code encompassing a specific statement, while edges depict the transitions from one instruction to another. CFGs can include nodes with multiple incoming or outgoing edges, such as those found in conditional statements. Additionally, they may contain cycles, often arising from loop statements like "for" and "while" loops or recursive method calls.

Listing 3.5 showcases the source code of a method named "start()," and Figure 3.2 illustrates its corresponding CFG. This method is part of the "LevelChangePropagator" class within the Lombok library (https://github.com/projectlo

mbok/lombok, accessed on April 17, 2023). During the analysis, since this method contains two method calls, each with its own CFG, the CFG of "start()" was constructed with four nodes in addition to all the nodes from the CFGs of the called methods, denoted as CFG_{96} and CFG_{98}. In cases where a single node is found to have multiple method calls during the analysis, the CFGs of the called methods are interconnected in the order of execution.

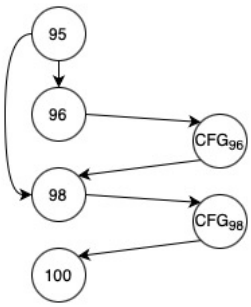


Figure 3.2: Control Flow Graph of the code shown in Listing 3.5.

Listing 3.5: Method start() has two method calls, one at line 96 and the other at line 98. The method is part of the *lombok* library.

```
    public void start() {
95      if (resetJUL) {
96          resetJULLevels();
97      }
98      propagateExistingLoggerLevels();
99
100     isStarted = true;
    }
```

Algorithm 5 provides a step-by-step outline of the instructions followed to build a CFG based on a method declaration. Our approach constructs a CFG for a given method, and for each method call identified within the method, its CFG is integrated with the main CFG of the caller. These identified method calls can either correspond

to methods implemented within the analysed application, in which case they are analysed, or they can be methods from external libraries without accompanying source code, which are not analysed. Consequently, when a method's body includes a call to a library-implemented method, we represent it as a single node within the CFG. The first and last nodes are preserved to facilitate the merging process between CFGs.

Algorithm 5 The algorithm used to build the control flow graph analysis for a method

```
procedure BuildCFG(m)
    cfg ← createEmptyCfg()
    for statement ← m.getStatements() do
        node ← createNode(statement)
        if firstNode = null then
            firstNode ← node
        end if
        cfg.addNode(node)
        if previousStmt ≠ null then
            cfg.addEdge(previousStmt, node)
        end if
        if statement.contains(MethodCall) then
            mCall ← statement.getMethodCall()
            if mCall.getCfg() = null then
                buildCfg(mCall)
            end if
            cfg.addEdge(node, mCall.getCfg().getFirstNode())
            previousStmt ← mCall.getCfg().getLastNode()
            if statement.next() = null then
                lastNode ← previousStmt
            end if
        else
            previousStmt ← node
        end if
        if statement.next() = null then
            lastNode ← node
        end if
    end for
end procedure
```

To accurately assess the workload assigned to each thread, it's crucial to establish clear and unambiguous execution paths within the Control Flow Graph (CFG). This requirement necessitates that between any pair of nodes in the graph, only a single path should exist connecting them. This condition must hold true for all nodes in the CFG. To achieve this, we need to handle conditional branches and loops appropriately.

For conditional statements with two alternative paths, we employ the worst-case execution time (WCET) approach [55]. This approach involves selecting the branch with the highest number of instructions, as it is likely to have the lengthiest execution time. The other branch is then disconnected from the graph. Consequently, pruned nodes are no longer connected to any other node in the graph. However, we retain these pruned nodes within our representation, as they may need to be reconnected to the graph for further analysis.

Listing 3.6 provides an example of a conditional statement with method calls in each branch. If the analysis determines that the method call within the "then" branch (line 79) should be parallelised due to the higher WCET, the "else" branch will be removed from the graph. The resulting CFG is shown on the left side of Figure 3.3. Conversely, if the "else" branch is selected, the "then" branch will be removed, as depicted on the right side of Figure 3.3. In the example from Listing 3.5, even though there is no "else" branch, the corresponding CFG will still be modified by removing the edge connecting node 95 to node 98. This is done because the branch 95→96→CFG96→98 is longer, with more instructions and a higher WCET value, than the branch 95→98.

Regarding loop management, some approaches aim to statically estimate the impact of loops on program execution [79, 78]. In our approach, to maintain efficiency

while keeping the analysis manageable, we treat loops as simple block statements. We analyse them without estimating the number of iterations that could be executed. Such a detailed analysis of loops is outside the scope of our proposal. However, we do keep track of these occurrences when evaluating the workload of a branch.

Finally, we consider cases where a chain of method calls introduces cycles, meaning a method can call a previous method in the chain, creating a loop. When we insert the corresponding CFGs into the main CFG, this can disrupt the assumption of a single path between any pair of nodes. However, such occurrences are very rare, and if encountered, we simply remove the edge that creates the loop in the graph.

Listing 3.6: A code fragment of a if/else statement with a method call (*apply(...)*) in both blocks. Both method calls are in the context of a *Variable Declaration Expression* (line 79 and 81).

```
78  if (ruleEntry.type == RuleEntry.TYPE_TEST_RULE) {
79      result = ((TestRule) ruleEntry.rule).apply(result, description);
80  } else {
81      result = ((MethodRule) ruleEntry.rule).apply(result, method, target);
82  }
83  return result;
```

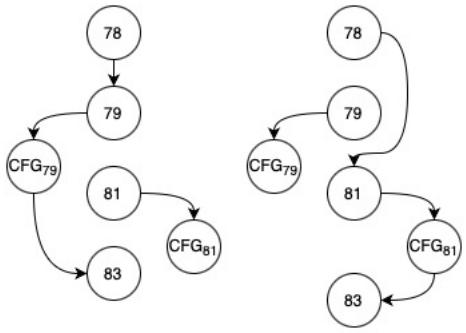


Figure 3.3: Control Flow Graphs of the code with conditions shown in Listing 3.6.

The final step in our approach involves evaluating the instructions that make up

the two paths, each of which can be assigned to a separate thread. The first path extends from the node containing the instruction we wish to execute in parallel to the subsequent instruction. The other path extends from this subsequent instruction to the data-dependent statement. Algorithm 6 outlines the procedure carried out in this step.

The paths are determined using Dijkstra's shortest path algorithm, utilising the API provided by the third-party library *JGraphT*. By design, the shortest path is the only path between two given nodes in the graph. The length of a path corresponds to the number of nodes (instructions) within it. Therefore, we examine the lengths of both paths, filtering out paths with short lengths and paths with length differences exceeding a predefined threshold. The function returns true if both paths meet these two criteria, and false otherwise.

Figure 3.2 illustrates the CFG of the method in Listing 3.5. In this example, node 96 is earmarked for parallel execution, while node 100 represents the data-dependent instruction. The first path extends from node 96 to node 98, while the second path extends from node 98 to node 100. The length of the first path is 17, whereas the length of the second path is 42 (accounting for the instructions within the called methods). When the main thread reaches the instruction at line 100, it will pause until the other thread completes its execution. Given that both paths contain a sufficiently large number of instructions, the method call is refactored to execute in parallel.

Listing 6 displays the code of the method refactored for parallel execution, utilising *CompletableFuture* to enable the execution of the method at line 96 asynchronously (using *runAsync()*). The instruction at line 99, *future.join()*, denotes the point at which the main thread awaits the completion of the task defined at line

96.

Algorithm 6 The algorithm used to define and compare two parallel paths

function COMPAREPATHS($node_1, node_2$)
  $m \leftarrow node_1.getStatement().getMethodCall()$
  $cfg_m \leftarrow m.getCfg()$
  $lastNode \leftarrow cfg_m.getLastNode()$
  $nextNode \leftarrow lastNode.getNext()$
  $path_1 \leftarrow DijkstraShortestPath.findPathBetween(node_1, lastNode)$
  $path_2 \leftarrow DijkstraShortestPath.findPathBetween(nextNode, node_2)$
  if $path_1.getLength() \geq 5 \wedge path_2.getLength() \geq 5$ **then**
    if $|path1.length() - path_2.length()| < 10$ **then**
      return $true$
    end if
  end if
  return $false$
end function

Listing 3.7: Method start() updated with a *CompletableFuture* executing the method called at line 96, where *future* is initialised, and for line 99, where *future.join()* waits for its completion.

```
    public void start() {
 94         CompletableFuture<Void> future;
 95         if (resetJUL) {
 96             future = CompletableFuture.runAsync(() -> resetJULLevels());
 97         }
 98         propagateExistingLoggerLevels();
 99         future.join();
100         isStarted = true;
    }
```

3.2.5 Summary

This study introduces an approach and a corresponding tool designed to perform static analysis of an application's source code in order to identify opportunities for

parallel execution. The approach is grounded in the analysis of control flow and data dependence, allowing us to assess the feasibility of executing certain statements in parallel while maintaining program correctness.

In this proposed approach, we automatically acquire the control flow graph of the target application under analysis. This graph serves as the foundation for identifying potential parallel paths and estimating potential performance enhancements. Subsequently, we generate a transformed parallel version of the application, where selected methods are executed in new threads through the use of Java CompletableFuture. The modifications made to the source code are minimal, ensuring a clean and effective transformation process.

To evaluate the effectiveness and correctness of both the approach and the developed tool, we conducted a series of experiments. The resulting parallel version consistently executes in a fraction of the time required by the sequential version. Furthermore, we rigorously assessed the transformation's correctness by subjecting it to multiple tests. Our comprehensive test suite provided coverage for 94% of the code, and the execution of these tests consistently confirmed that the transformations preserved the results of the executed methods.

These two studies, discussed in this chapter, comprise the first module of our refactoring framework. They provide developers with the opportunity to enhance their application's performance by executing computationally intensive methods in parallel. The subsequent chapter will delve into the remaining two modules of our proposed framework: "Loop to Stream" and "Matching Algorithm."

Chapter 4

Loops To Stream and Matching Algorithms

Modern software applications can have millions of lines of code. Moreover, while several developers collaborate to develop code, each of them could have their own coding style, e.g. follow a personal code convention, prefer some data structures, etc. Having a large amount of code and several coding styles can negatively affect the readability of the code, especially when this has not been documented properly. A somewhat incorrect version of an algorithm would lead, besides to the presence of bugs, an involute, complex and inefficient code.

Furthermore, Java 8 introduced the functional programming paradigm into Java language. This allows developers to implement more expressive and concise code, and at the same time reduces the possibility to introduce defects, such as e.g. off-by-one errors, etc. Java 8 streams and lambda expressions cater for higher-level statements, expressing some goal, while the underlying library takes care of lower-level details [142]. Since lambda expressions allow further parameterisation, refactoring techniques can be leveraged to handle e.g. clone fragments [140].

In this chapter, we introduce two different modules to enhance the readability

and the efficiency of source code by analysing the structure of the code in order to suggest improvements. We propose a module that automatically refactor for loops into Java Streams (Section 4.1) and a module that perform a similarity analysis to identify known algorithms within the code and suggest optimised versions (Section 4.2).

4.1 Refactoring Java Loops to Streams Automatically

Java, starting from version 8, adopted the functional paradigm, comprising lambda expressions, streams, functional interfaces, etc. Java stream APIs provide developers with the possibility to parallelise the stream-based fragment just by calling the parallel() operation (or by replacing stream() operation with parallelStream()). Of course, it should be checked whether the parallel version changes the expected behaviour [69], avoid performance drop due to thread contention [132] and deadlocks [109]. Therefore, stream APIs can be effectively applied to Big Data in order to achieve scalability [25].

Despite Java 8 being released in 2014, a large amount of code available presents traditional for loops, and some developers have not fully embraced the functional paradigm (as it would require a shift in the reasoning). This paper aims at assisting the conversion of for loops into stream-based fragments. The literature suggests two main approaches for refactoring for loops into appropriate stream code [45, 112]. The first approach presents an automatic tool that checks whether a few proposed transformations can be performed. However, we identify several additional and novel transformations that, while can be handled automatically, are applicable to

widespread fragments of code. In the second approach, practical rules to transform loops have not been given.

The aim of this paper is twofold: (i) guiding the developer to use Java stream APIs by devising several formats and their equivalence to loops; (ii) providing an automatic tool supporting the analysis, identification, and refactoring of loops to streams. Aiding the developer could make the newer paradigm mainstream, and automatic conversion gives higher quality code.

This section is structured in the following way. Section 4.1.1 presents the proposed templates that guide the subsequent transformations. Section 4.1.2 reports our approach for parsing, analysing, matching and converting loops. Section 4.1.3 draws the conclusion.

4.1.1 Refactoring Templates

We have identified five templates for recognising loop occurrences and proposed the transformations to refactor them to suitable stream-based code. The following subsections describe the templates.

The first template handles for loops that contain a conditional statement and one or more return statements giving back a primitive value or an object. For such a template the equivalent stream-based version comprises a *filter()* and a *findFirst()* operation. The latter is a terminal operation that performs short-circuiting, i.e. blocks evaluations, and gives the first value that satisfies the predicate in *filter()*. Once an element has been found, or no more elements in the input stream exist, *findFirst()* returns an *Optional¡T¿*, where T is the type of the elements in the stream. For performing the transformation, the conditional expression in the original code constitutes the predicate for *filter()*. An Optional variable is assigned with the value

extracted by *findFirst()*. Finally, a check is performed on the optional value in order to call the return statement with the proper value. Figure 4.1 shows a sample code in both versions.

Listing 4.1: For loop having a conditional statement and a return statement.

```
// original code before refactoring
for (Marker m : markers) {
    if (getLineOfOffset(m.getPosition()) == line)
        return m;
}

// code after refactoring
Optional<Marker> mark = markers.stream()
  .filter(m -> getLineOfOffset(m.getPosition()) == line)
  .findFirst();
if (mark.isPresent()) return mark.get();
```

The second identified template is a *for* loop whose body has a temp variable declaration and assignment, then a conditional statement whose expression uses the assigned variable and a return statement using the assigned variable. Note that after the first assignment, the current value of the iteration is not used anymore. The stream-based version of it has a *map()* operation whose mapper function is the right part of the assignment in the original statement. Though, after a *map()* has been called the current element of the stream cannot be referenced anymore, it is appropriate in this case as the subsequent statements need not use it. The remaining statements in the for loop are handled analogously as in the first conversion template. Figure 4.2 shows a sample code using the traditional loop and its equivalent stream-based version.

Listing 4.2: For loop having an assignment, a conditional and a return statement.

```
// original code before refactoring
for (Matchable m : getMatchTypes(startCharacter)) {
    SymbolMatch match = matcher.makeMatch(m);
    if (match.isMatch()) return match;
}

// code after refactoring
Optional<SymbolMatch> tmpMatch = getMatchTypes(startCharacter).stream()
    .map(candidate -> matcher.makeMatch(candidate))
    .filter(match -> match.isMatch())
    .findFirst();

if (tmpMatch.isPresent()) return tmpMatch.get();
```

The third identified template is a for loop having in its body a temp variable and some other statements, such as a conditional statement using the current value of the iteration, and a method call. Note that the main difference with the second template is that the current value of the iteration is needed in the assignment statement and in the subsequent statements, hence a map() operation cannot be used as it would consume the current value.

Before transforming this loop, we apply the Replace temp with query refactoring technique [42] to get rid of the temp in the loop body. This technique extracts the expression on the right side of the assignment into a method, in order to make the temp no longer local to a method.

Now, we transform the loop body into a stream-based code by using a filter() corresponding to the conditional statement, and a forEach() to perform the method call on each value of the stream. We avoided using map() that would have made it impossible to further operations to reference the current element of the stream, and

at the same time we have no fragment repetitions as it would have been by replacing each occurrence where the variable is read with the right side of the assignment. Figure 4.3 shows a code sample of this template and its transformation.

Listing 4.3: For loop having a temp and a condition on the current value.

```
//original code before refactoring
for (Map.Entry<String, Integer> e:rawPackCount.entrySet()){
    String packageName = e.getKey();
    if (e.getValue() > 5) {
        System.out.printf("%5d %s%n",e.getValue(),packageName);
    }

    // proposed code after refactoring
    rawPackCount.entrySet().stream()
        .filter(e -> e.getValue() > 5)
        .forEach(e -> System.out.printf("%5d %s%n", e.getValue(), packName(e)));

    // method obtained after applying refactoring Replace temp with query
    String packName(Map.Entry<String, Integer> e){
        return e.getKey();
    }
}
```

The fourth template consists of for loops where some operation is called on each element of the iteration to create a new value, then such a new value is inserted into a collection. Hence, there is an assignment to a temp variable holding the created value resulting from a method call using as a parameter the current value of the iteration. The transformation into a stream-based code consists in calling: a map() for obtaining a newly created value (as the current value of the iteration is not needed in subsequent statements), then a collect() to convert the output stream into a collection. Figure 4.4 shows an example of code and its transformation. Operation map() creates an object of type DetectorNode, then operation collect() makes all

the resulting values available as a HashSet. Of course, the result could be e.g. a List if needed in the original code.

Listing 4.4: For loop having a temp and producing a collection.

```
// original code before refactoring
HashSet<DetectorNode> result = new HashSet<DetectorNode>();
for (DetectorFactory factory : chosenSet) {
    DetectorNode node = addOrCreateDetectorNode(factory, nodeMap, constraintGraph);
    result.add(node);
}

// proposed code after refactoring
HashSet<DetectorNode> result = chosenSet.stream()
    .map(factory -> addOrCreateDetectorNode(factory, nodeMap, constraintGraph))
    .collect(Collectors.toCollection(HashSet::new));
```

The fifth template consists of an if-then-else statement, hence a secondary path of execution exists, followed when the condition evaluates false. We transform a conditional statement in a filter(), however since no secondary path can be executed in the same stream, we use two different streams, having complementary conditions. Therefore, the conditional statement is replaced with a filter(), and the condition is used as a predicate, then we handle the transformation of statements on the primary path of execution as in the above templates. Additionally, another stream is created having a filter() using the complementary condition. Figure 4.5 shows an example of code having a stream for each path, and a further condition for the secondary path of execution, i.e. a if-then-else-if statement. Here, the effect of the assignment operations, start = node and end = node is obtained using findFirst() operations, then by extracting a value, if it exists, from the Optional variables.

Listing 4.5: For loop having a if-then-else-if statement.

```
// original code before refactoring
for (Node node : nodeList){
    if(startNode.compareTo(node.name) == 0) start = node;
    else if(endNode.compareTo(node.name) == 0) end = node;
}

// proposed code after refactoring
Optional<Node> tmpstart = nodelist.stream()
    .filter(node -> startNode.compareTo(node.name) == 0)
    .findFirst();
Optional<Node> tmpend = nodelist.stream()
    .filter(node -> endNode.compareTo(node.name) == 0)
    .findFirst();
if(tmpstart.isPresent()) start = tmpstart.get();
if(tmpend.isPresent()) end = tmpend.get();
```

Figure 4.6 shows a further code sample having a if-then-else statement. Here, the statements on each path are transformed using operation forEach().

Listing 4.6: For loop having a if-then-else statement.

```
// original code before refactoring
for (String attribute : attrs) {
    if(isChecked(request, attribute))
        data.setAttribute(attribute);
    else
        data.removeAttribute(attribute);
}

// proposed code after refactoring
attrs.stream()
    .filter(attribute -> isChecked(request, attribute))
    .forEach(attribute -> data.setAttribute(attribute));
attrs.stream()
    .filter(attribute -> !(isChecked(request, attribute)))
    .forEach(attribute -> data.removeAttribute(attribute));
```

4.1.2 Tool for Refactoring Loops

We perform static analysis, by means of JavaParser library, in order to analyse the code and gain knowledge necessary for the proposed replacements. JavaParser represents the source code as an abstract syntax tree (AST) and allows performing operations on it [129]. The main four steps followed in our approach to refactor for loops to stream-based code are: (i) parsing the original source code to gather all the data needed for the following analysis; (ii) check whether for loops satisfy all our defined preconditions; (iii) match loops with templates and then build the stream-based code; (iv) replace each loop with the appropriate stream-based statements.

Firstly, we search all the method declarations present in each class file, we extract the name of the class that contains the methods, and finally search all the for loops within them. We consider the two versions: standard, i.e. for loops with indexes,

as in *for (int i = 0; i<n; i++)*, and advanced, i.e. having the format *for (T item: list)*, to iterate lists, sets, etc. This classification is necessary in order to recognise the loop and perform its replacement. Loops are checked to determine whether they iterate over an instance conforming to Collection, since the transformation would convert the object into a stream by using default method stream(). Moreover, a check is performed to verify that in standard loops, the index is only used to access the list. In the standard for loop, it is necessary to analyse both the initialisation of the index variable and the condition. The straightforward case is when the index is given initial value 0 and iterations scan the list sequentially till the last element. Conversely, when the index is given initial value as the maximum size then iterations start from the last value and the stop condition occurs when the index reaches value 0; then a reconstruction would be needed to access the elements in reverse order, before using them in a stream.

The analysis of the for statement consists in verifying whether it has all the functional requisites for becoming a stream and whether operations using lambda expressions can be performed on it. The checked preconditions are the following.

- The number of statements inside the body of the loop has to be at most five, where in an if-then-else statement both paths are considered as a single statement. More than five statements likely means the functional style is not appropriate for such a loop.

- The body of the loop cannot contain another for loop, a switch, or a while loop. These three types of statements are excluded since it would not be possible to replace them using streams.

- The body of the loop cannot contain more than one reference to non-effectively final variables defined outside the loop; indeed we previously have shown an example of a loop with a single non-effectively final variable converted to an appropriate stream.

- The body of the loop cannot contain any break statement; this statement is characteristic of imperative programming, indeed there is not a functional construct in Java streams which closely mimic such a behaviour.

Once the loops satisfying the above preconditions have been selected, then matching and conversion steps can be executed. The matching step follows a top-down approach: it analyses the body of a loop starting from the first expression, and it continues till the end. The approach compares each expression with the expression in the same position in the templates available: if there is no match, the template is discarded and a match to another will be attempted until all templates are checked. Conversely, if all statements inside the loop match a template, then the conversion step will be performed. This step firstly extracts needed data, i.e. the name of the collection in the iteration, the name of the current element, all the expressions inside the loop body together with variable names, etc. Therefore, the template can be filled with all data extracted and the new compound statement created.

Once the replacement expression has been created, we move on to the actual replacement. For each loop, the replace() method provided by JavaParser is used, which allows us to replace a node of the AST with a new node passed as a parameter. Thanks to this operation it is possible to insert the new expression in the same place as the original loop, so as not to modify other parts, keeping indentation and lexical correctness. For inserting additional expressions, such as the check of

Optional variables, then it suffices to have the position of the loop and insert the new expression in the next position using the *add()* method provided by JavaParser.

4.1.3 Summary

This study presents an innovative approach to refactor for loops to Java stream APIs by defining templates to guide code analysis and transformation. We have proposed five novel templates that consider some of the most typical uses of loops, and include in their body several categories of statements, such as conditionals, assignments to temp, secondary paths, return statements, etc. We have handled the correspondence to traditional loops in order to safely transform the code from imperative to functional without changing the behaviour. Our templates are more general than some previous approaches and can be automatically applied.

When analysing existing code, the significant number of template occurrences found has confirmed the validity of our approach, showing that the definition of such templates could help to transform actual code used in popular repositories from the imperative version into the functional one. The use of Java stream APIs makes the source code more readable. Besides the possibility to parallelise stream pipelines could lead to an increase in the software performance itself.

The following Section describes the last module of the refactoring framework, the *Matching Algorithm*, an approach to identify and suggest optimised versions of the same algorithm to improve readability, performances, and reusability of the source code.

4.2 A Robust and Automatic Approach for Matching Algorithms

The bigger the repositories the greater the demanding effort for developers when trying to understand code implemented by their colleagues. Visually inspecting source code to understand its structure and the functionalities could be time consuming and error-prone, since developers could misidentify some code behaviour. Generally, this occurs when the source code documentation is poor, or missing, and when many developers contribute to the same project.

Automatic Program Comprehension (PC) tries to solve these issues by proposing several approaches that automatically assist developers to understand source code [127, 80]. Several applications of the PC have been presented: source code classification according to specific categories [137, 141, 131], code clone detection, and algorithm recognition.

The state of the art shows several techniques to automatically identify algorithms. Machine learning approaches have been presented [123, 29], where different classifiers are used to label code fragments. Other proposals use a hybrid approach, mixing static analysis to extract data from the code and machine learning classifiers to identify the algorithm [135, 133, 134]; their classifier is based on a set of structural and truth value characteristics that are strictly related to sorting algorithms, therefore new beacons should be defined for different categories of algorithms. Furthermore, these approaches require a new training of the dataset if new labels are included in the classification, which can be time consuming and complex.

This study focuses on algorithm recognition and presents an innovative approach that automatically matches algorithms by inspecting the source code. Our approach

uses static analysis to collect data from the source code and compute a similarity score with templates of known algorithms to identify the correct one. The use of templates guarantees that: new algorithms can be easily added for the recognition step; multiple versions of the same algorithm can be used to improve the accuracy of the identification; many, if not all, categories of algorithms can be recognised (sorting, searching, traversing, etc.). The proposed approach consists in four main phases: firstly, a code parsing tool collects all the statements of the algorithm analysed; secondly, a statement transformation is performed to extract the data required for the similarity match; thirdly, the Levenshtein distance is computed to attribute a similarity score between the algorithm and a set of known templates, representing other algorithms; finally, the template with the highest similarity score is selected.

Levenshtein distance has been widely used in program analysis, especially for code clone detection when evaluating the similarity between code fragments [126, 63, 5]. However, such approaches are strictly related to detecting clone fragments. Such approaches focus on detecting type-3 clones that are portions of code that differ in terms of whitespaces, comments, layouts and identifiers, and can have some modifications like addition or removal of statements [2]. Conversely, we propose another approach in which different statements of the algorithms under analysis are reflected on the matching score, hence having a varying degree of matching; moreover, the above said approaches refer to types and names (e.g. methods and fields names) to detect clones, whereas our approach focuses on the statements freed from the developer chosen names, providing a more generalised identification.

This section is structured as follows. Section 4.2.1 describes the proposed approach with all the steps followed by the analysis. Section 4.2.2 shows three examples of the use of the approach and how the similarity score is computed in real scenarios.

Section 4.2.4 sums up the presented approach.

4.2.1 Proposed Approach

We propose an approach that gathers data by parsing the source code and then evaluates the similarity score of an algorithm and a set of known algorithms. The proposed approach makes a proper generalisation of the algorithms to avoid depending on naming conventions or on statements that are not contributing to the main goal of the analysed algorithm. We make use of the static analysis of the source code, hence executable files are not needed for the analysis. Algorithm 7 shows the main steps, as pseudo-instructions, followed by the proposed approach to match algorithms. The procedure takes as input the source code, variable SC, and parses it, extracting the compilation units. Then, method declarations are extracted and, for each, all the statement types are collected and compared to the algorithm templates to compute the similarity score. The approach can be structured in four steps: (i) parsing the source code by means of a tailored Visitor to gather all the data needed for the following analysis; (ii) selecting and transforming the most relevant statements; (iii) computing an adapted Levenshtein distance to determine the similarity score; (iv) evaluating the matching degree of the algorithms.

We perform code parsing, by means of the Javaparser library, to extract all the data required to evaluate the similarity score. Javaparser is an automatic parser that generates an abstract syntax tree (AST) from source code and provides a set of APIs to perform operations on it [129].

The root of the AST is the CompilationUnit (representing a Java file) to which all code elements are connected, e.g. package declaration, class and methods declarations, etc. Code inspection has been performed by using the VoidVisitorAdapter

Algorithm 7 The algorithm of the proposed approach

```
procedure MatchingAlgorithm(SC)
    compilationUnits ← parseAllPaths(SC)
    for cu, compilationUnits do
        methods ← visitMethods(cu)
    end for
    for mDecl, methods do
        stmts ← mDecl.getStatementsType()
        for tmp, templates do
            Tstmts ← tmp.getStatementsType()
            score ← computeLVD(stmts, Tstmts)
            mDecl.collectScore(tmp, score)
        end for
    end for
end procedure
```

class, which lets us define a Visitor class to search for a specific property. In the Visitor class, the method visit() was implemented, which takes as parameters the type of object being searched (e.g. method declaration, statement, field), and the container in which data are stored; the body of the method contains all the instructions that are executed for every object found of the type specified as a parameter.

We have defined a Visitor which looks for MDs (method declarations); once a MD is found, the visit() method extracts from its body all the statements, and stores them in a List preserving the order. The list of statements provided by Javaparser is further expanded, as it would otherwise miss: (i) nested statements (e.g. all the statements present in the body of a for loop, defined as ForStmt), and (ii) expressions, such as assignments, method calls, variable declarations, etc. (defined as ExpressionStmt).

The statements initially gathered by Javaparser are transformed to better serve the following analysis. Javaparser provides a function, getStatements(), to get the statements contained in the body of a method declaration; nested statements, e.g.

statements contained in a for loop, are omitted be the said getStatements(), and just the parent statement is inserted in the list. To collect all the statements inside the method, we further extract nested statements and we place them in the list of statements right after their parent, preserving the order of the block. Whereupon, for each statement in the list, we extract the class type, avoiding collecting other parts such as names, types, comments and expressions, to better generalise the approach, so as to recognise different versions of the same algorithm; e.g. statement for(int i=0; i¡size; i++) is represented in Javaparser by a ForStmt type, whereas all the other parts, such as the variable declaration (int i=0), the binary expression (i¡size) and the unary expression (i++) are omitted.

For all the statements that can contain nested statements in their body, e.g. for, if, while, a custom statement is inserted at the end of the nested statements; its name is given by the concatenation of the "End" prefix with the type of the statement containing the nested ones, e.g. EndIf, EndFor, EndWhile. This custom statement allows us to identify which statements are nested, thus improving the precision of the approach when comparing algorithms.

The extracted types are defined by the Javaparser library1. Table 4.1 shows some examples of statement types: the column Statement Type represents the type defined by the library, and the column Code shows an example of the code associated with the type; further types can be found in the documentation. The ExpressionStmt type does not have a code example because it can represent any type of expression: AssignExpr (a = b +c;), MethodCallExpr (method();), VariableDeclarationExpr (int a = 0;) etc. This class type is too generic, hence we select and insert in the statement list the type of the expression contained in the statement. If there are nested expressions, e.g. a method call with argument an ObjectCreationExpr

Table 4.1: Some examples of statement types defined in the Javaparser library (see the documentation for the complete list[0]). The *ExpressionsStmt* is handled differently from others, since it can represent more types of expressions (method calls, assignments, declarations, etc.).

Statement Type	Code
BreakStmt	break;
ContinueStmt	continue;
DoStmt	$do\{...\}while(a > 0);$
ExpressionStmt	*
ForEachStmt	$for(Object : objects)\{...\}$
ForStmt	$for(a = 3; a < 99; a++)\{...\}$
IfStmt	$if(a == 0)\{...\}$
ReturnStmt	return a;

(e.g. method(a, new object())), we select the parent expression, in this example the MethodCallExpr.

Listing 4.7 shows an example of data extracted from a method: the code on top shows the method bubbleSort(); the bottom part displays the list of statements extracted by our visitor. The statement list extracted by Javaparser contains just a ForStmt because all the other instructions are nested into it, whereas our approach has properly handled this occurrence and the statement list is defined as follows: the first two instructions are for statements, the third is a if statement, the fourth is a variable declaration expression and the last two are assign expressions.

Listing 4.7: The upper part shows an iterative version of the *bubblesort* algorithm implemented in Java, whereas the bottom part displays the list of statements extracted by our approach, using the class types defined in Javaparser.

```
public static void bubblesort(int [] sort_arr, int size){
  for (int i=0;i<size-1;++i){
    for (int j=0;j<size-i-1; ++j){
      if (sort_arr[j+1]<sort_arr[j]){
        int tmp = sort_arr[j];
        sort_arr[j] = sort_arr[j+1];
        sort_arr[j+1] = tmp;
      }
    }
  }
}

Statements Type:
ForStmt,
ForStmt,
IfStmt,
VariableDeclarationExpr,
AssignExpr,
AssignExpr
EndIf
EndFor
EndFor
```

The Levenshtein distance is a string metric for measuring the difference between two sequences [19]. It is defined as the minimum number of operations (replace, insert and delete) required to change a sequence into the other. A string can be seen as a list of single characters; the Levenshtein algorithm iteratively compares all the characters and finds the minimum number of operations (insertion, deletion or substitution) required to make the two sequences equal. We have implemented a custom version of the algorithm where two lists of statements are the compared

sequences, and every character represents a single statement. Once defined the number of minimum instructions, the similarity score is computed as [20]:

$$Similarity(S_1, S_2) = 1 - \frac{levDist(S_1, S_2)}{max(size(S_1), size(S_2))}$$

where S_1 and S_2 are the two sequences of statements, *levDist()* gives as output the Levenshtein distance, and *size()* gives the number of elements in a sequence.

Once all the data needed for the analysis have been obtained, we can compute the similarity scores according to the extracted list of statements, i.e. the analysed method is compared with a set of known algorithms that have been gathered and parsed beforehand. We have created a set of Java files containing the source code of several known algorithms, and for each one at least two versions are stored: iterative and recursive. For some algorithms, more versions have been implemented as the aim is to improve the accuracy of our analysis. E.g., the bubblesort algorithm has two different versions, besides the iterative and recursive versions: the one shown in Listing 1, and an optimised version where a boolean flag breaks the execution if no elements are swapped in the inner loop.

Listing 4.8 shows one of the templates of the iterative version for the bubblesort algorithm used in our analysis: the code on top displays the implementation of the algorithm, while the list below represents the statements extracted by our approach. We show this template because, according to our approach, it is the most similar to the code shown in Listing 4.7. However, the two methods have several textual differences: firstly, the name of some variables is different, e.g. the variable representing number of elements, size and length, the array containing the elements, $sort_arr$ and list, and the variable used for the swap, tmp and swap; secondly, the condition in the IfStmt is inverted; finally, the template has an additional statement compared

to the example, the first statement VariableDeclarationExpr.

Despite these differences, our approach correctly identifies the algorithm implemented. According to the Levenshtein distance, the number of operations needed to match the two sequences is one (an insertion because the list of statements differ in only one element). Indeed the similarity score between these two sequences is computed as $Similarity = 1 - (1/7) = 0.857$, where the the size of the longest sequence is 7.

Listing 4.8: The upper part shows one iterative version of the *bubblesort* algorithm stored in our template db, whereas the bottom part displays the list of statements extracted by our approach, according to the types defined in Javaparser.

```
void iterativeBubbleSortTemplate(int list[], int length) {
  int length = list.length;
  for(int i=0; i < length; i++) {
    for(int j=1; j < length-i; j++) {
      if(list[j-1] > list[j]){
       int swap = list[j-1];
       list[j-1] = list[j];
       list[j] = swap;
     }
   }
  }
}

Statements Type:
VariableDelcarationExpr,
ForStmt,
ForStmt,
IfStmt,
VariableDeclarationExpr,
AssignExpr,
AssignExpr
EndIf
EndFor
EndFor
```

4.2.2 Evaluation

We tested our approach on four different algorithms, each implemented as a method. All the templates used by our approach can be found on a public repository[1]. The

[1]https://github.com/AleMidolo/MatchingAlgorithms

first example is shown in Listing 1 previously discussed; here, we discuss three other algorithms: a version of factorial and two versions of quicksort.

Listing 4.9 shows a method implementing the factorial algorithm for an integer value in a recursive form. The list of statements is as follows: IfStmt, ReturnStmt, ReturnStmt. The analysis carried out by our approach has identified the method as the recursive version of the factorial algorithm with a similarity of 1.0. In such a case, the similarity is the maximum possible value since given the simple structure of the algorithm, the types of statements used by such a method match 100% the factorial algorithm template.

Listing 4.9: An example of the recursive factorial algorithm implemented in Java.

```
public static int factorial(int n) {
  if (n == 0 || n == 1)
    return 1;
  return n * factorial(n - 1);
}
```

Moreover, two versions of the quicksort algorithm are considered to test the approach on different versions of the same algorithm; both versions propose an iterative solution. Firstly, Listing 4.10 shows an iterative version of quicksort algorithm that uses a stack as a support to sort the elements contained in the array passed as argument. The method consists in twelve statements, in order: two VariableDeclaration, MethodCall, WhileStmt, VariableDeclaration, AssignExpr, MethodCall, VariableDeclaration, and two IfStmt with a MethodCall in their body. Therefore, the method is characterised by such statements, indeed comments, names and types will be ignored for the purpose of the identification. Our template storage includes

an implementation of the quicksort using a stack to sort the elements[2]; the differences between the template and this method are: the template has an additional VariableDeclarationExpression before the first push() call; in the example, the first instruction after the WhileStmt is a VariableDeclarationExpr, while in the template it is an AssignExpr; the partition() method call takes one more argument in the template; types and names of the variables are different. Considering the said differences, our analysis has computed a similarity score of 0.85, correctly identifying the algorithm.

[2]The templates in the db can be found in the github repository above

Listing 4.10: An iterative version of the quicksort algorithm using a stack of objects to sort the elements.

```
public void quickSortStack(short[] array) {
  // create a stack for storing
  // subarray start and end index
  Stack<Pair> st = new Stack<>();
  short finish = array.length -1;
  // push the start and end index
  // of the array into the stack
  st.push(new Pair(0, finish));
  // loop till stack is empty
  while (!s.empty()) {
    // remove top pair from the list and get
    // subarray starting and ending indices
    short begin = st.peek().getX();
    finish = st.peek().getY();
    st.pop();
    // rearrange elements across pivot
    short pv = partition(array, begin);
    // push subarray indices with elements
    // less than the current pivot to stack
    if (pv - 1 > begin) {
        st.push(new Pair(begin, pv - 1));
    }
    // push subarray indices with elements
    // more than the current pivot to stack
    if (pv + 1 < begin) {
        st.push(new Pair(pv + 1, begin));
    }
  }
}
```

Secondly, Listing 4.11 displays an iterative version of the quicksort algorithm that uses a supporting array to sort the elements of the array passed as argument. The method consists in fifteen statements, in order: three VariableDeclaration, two

AssignExpr, WhileStmt, three AssignExpr, and two IfStmt with two AssignExpr in their body. There are two main differences in the structure between the two versions of the same algorithm: the total number of statements, twelve against fifteen, and the absence of MethodCall statements in the second version. An iterative version of the quicksort algorithm is stored in our template storage, and it uses an array to sort the elements like the method given as an example. The main differences between the template and the method are the following: the method has three VariableDeclarationExpr before the WhileStmt, whereas the template has only two; types and names of variables are different. The analysis has computed a similarity score of 0.86.

Listing 4.11: An iterative version of the quicksort algorithm using an array to handle the sort of the elements.

```
public static void quickSortArray(
    long arr[], long low, long high) {
  // Create an auxiliary stack
  long[] list = new long[high - low + 1];
  long max = -1;
  long pivot = 0;
  // push initial values to stack
  list[++max] = low;
  list[++max] = high;
  // Keep popping from stack while not empty
  while (max >= 0) {
    h = list[max--];
    l = list[max--];
    // Set pivot element at its correct
    // position in sorted array
    pivot = partition(arr, low, high);
    // If there are elements on left side
    // of pivot, push left side to stack
    if (pivot - 1 > low) {
        list[++max] = low;
        list[++max] = pivot - 1;
    }
    // If there are elements on right side
    // of pivot, push right side to stack
    if (pivot + 1 < high) {
        list[++max] = pivot + 1;
        list[++max] = high;
    }
  }
}
```

Our approach correctly identified both versions because the analysis uses templates for different versions of the same algorithm, making the recognition more accurate. Still, the storage containing all templates can be updated with more versions of algorithms to make the approach more sensitive to differences and up-to-date.

4.2.3 Refactoring for Energy Efficiency

Energy efficiency has emerged as a significant concern in modern software engineering, driven by the necessity to conserve battery life in applications [122]. Furthermore, the growing global awareness of environmental issues due to climate change and increased global warming has led to a heightened demand for reducing the power consumption of everyday devices, such as smartphones and laptops. While the primary focus has been on hardware efficiency, recent research has revealed that software can also contribute to energy inefficiencies [62].

The advantages of refactoring extend beyond enhancing code comprehensibility, encompassing aspects like extensibility, reusability, and testability. Surprisingly, one significant challenge that has received relatively less attention is energy consumption during software development [107].

Despite the increasing need for developers to optimise the energy efficiency of their software, they often lack the necessary knowledge to do so [100, 147]. This is unfortunate for several reasons: firstly, energy efficiency has become a primary concern for sustainability in computing systems; secondly, with the widespread use of mobile platforms, battery consumption is a critical factor for evaluating and adopting mobile applications; and thirdly, energy-efficient applications can result in reduced cooling costs and less environmental pollution [147].

Nevertheless, creating energy-efficient software is a challenging task. Understanding where energy is consumed in the code and how it can be restructured to reduce consumption remains a fundamental issue. Although there are tools for estimating energy consumption (e.g., [58, 73, 76]), they have limitations. They require an in-depth understanding of low-level implementation details, which is often impractical for time-pressed programmers, and they do not offer direct guidance on energy optimisation, bridging the gap between identifying energy consumption and code modification. Consequently, programmers often resort to searching for energy-saving best practices in online forums and blogs, which may lack empirical evidence and accuracy [106]

Recent research into energy optimisation reveals that incorrect choices made by programmers during software development can negatively impact application energy consumption [59]. They investigated the effects of using Java Collections, demonstrating that selecting the wrong data structure can reduce energy efficiency by up to 300%.

Other researchers, such as Bunse and colleagues [18, 19], focused on dynamically adapting systems to use the most energy-efficient sorting algorithms, while Manotas and colleagues [82] designed a tool to autotune Java applications by selecting energy-efficient implementations for Collections APIs. There have also been explorations into the energy impact of design patterns by various researchers [17, 75, 117].

Choosing the right algorithm for a particular task is a crucial consideration, as emphasised by Schmitt in their 2021 work [121]. An integral component of many programs is the sorting of data, a task that is well-established. Various studies have explored the idea of energy conservation within the context of sorting algorithms, as evidenced by research by Rashid [113] and Chandra [26]. These works mainly focus

on battery-powered mobile devices and they consider just sorting algorithms.

Our proposal is to employ the matching algorithm approach (referenced in subsection 4.2.1) to detect algorithms within the source code and suggest a refactoring strategy aimed at choosing the most energy-efficient algorithm. For instance, consider the bubblesort algorithm, which can be implemented in at least three different variations: iterative, iterative with a flag, and recursive. To evaluate their energy consumption, a dedicated benchmark would be conducted, ultimately selecting the most efficient version. To demonstrate the effectiveness of this refactoring, it would be applied to a set of well-known open-source applications for assessment.

4.2.4 Summary

This module presented an automatic approach to recognise algorithms using static analysis. By parsing the source code it is possible to identify all the statements composing a method, transform them according to a specified format, then compute Levenshtein distance for obtaining a similarity score between the method and several templates of known algorithms. The template having the highest score is suggested as the algorithm matching the analysed method. We performed an experiment on four methods to test our approach; the results obtained highlight a high accuracy when recognising the algorithm compared to a textual similarity. The versatility of the approach allows us to add more templates to widen the spectrum of recognisable algorithms and to increase the number of different versions of algorithms. This approach can be employed for several goals: program comprehension purposes on software development, supporting developers in understanding and implementing source code, by proposing alternative versions of the same algorithm; academic purposes to automatically assess students' assignments; energy efficiency

by proposing the most energy efficient version of the algorithm in order to reduce the power consumption of the entire application.

Chapter 5

Automatic Test Generation

Refactoring aims at improving non-functional attributes of a software system. To do so, each refactoring should check several preconditions in order to preserve the original behaviour of the system, indeed for object oriented languages such as Java, demonstrating that refactorings are semantically equivalent to the original program is a challenge [92]. Refactoring engines must be reliable, as a bug in a refactoring engine can silently introduce errors into the refactored program, making debugging difficult [30]. If a refactored code does not compile unlike the original one, the refactoring is clearly incorrect and can be easily undone. However, if the refactoring compiles but it changes the semantics of the application, it could lead to dire consequences. Even refactorings performed by Integrated Development Environments could be fault-prone [9].

Each refactoring may involve multiple and complex preconditions that are needed to guarantee behavioural preservation [130]. For example, the refactoring proposed in Section 3.2 requires a deep data dependence analysis to correctly determine where the synchronisation statement should be inserted; an erroneous identification could lead to data race errors.

In order to keep the refactoring reliable, developers can write tests that check

whether the refactored version of the code has kept the same behaviour of the original one. However, substantial effort is required when implementing test cases, due to the knowledge that developers have to acquire on the structure and behaviour of the system under test, and for the time needed to write many test cases to cover most of the execution paths.

This chapter presents an automated testing framework that provides two different modules for test generation. The first module is the *Automatic Generation of Effective Unit Tests based on Code Behaviour* (Section 5.1),This module automatically generates tests for untested classes by analysing the classes of the application to find similarities between them and adapting existing tests for classes without tests. The second module, *Automatic Generation of Accurate Test Templates based on JUnit Asserts* (Section 5.2), automatically generates templates of test cases for untested methods by analysing the type returned by the method to select the best set of assertions to effectively test the method.

5.1 Automatic Generation of Effective Unit Tests based on Code Behaviour

Producing test cases is an effective way to check the correctness of a software system, and to check that evolutionary changes aiming at improving functionalities are not introducing defects to previously correct code [61, 151]. However, developing tests is a time consuming activity. When designing tests, a developer has to take into account the behaviour of the component under test to determine the set of inputs and expected output, which are essential to implement a test case. Moreover, during

implementation some boilerplate code needs to be added to give the needed context to the test case.

The existing literature on tests suggests several approaches for assisting the work of developers. Since one of the tasks of the developers is the selection of input values to be given to a method, combinatorial approaches for input values can be very effective and many tools have been implemented to find values among given validity ranges, as well as outside validity ranges [150, 22]. Another task developers have to perform is implementing methods calls that check the behaviour of a class, assistance for it has been given by tools that e.g. randomly generate a sequence of calls [46, 99, 136]. Moreover, for the task of finding expected output values to check against the resulting execution behaviour, often the solution is building a model of the system [23]. Other approaches for producing tests and making them robust include the generation of code variations to make sure that tests can find the erroneous behaviour [65]. Moreover, some approaches have been proposed to automatically generate code that passes all tests [60].

Most of the approaches aim at having and executing as many test cases as possible, which is worthy for checking a large amount of execution scenarios. However, given the large number of possible input data and execution paths inside the software system under test, execution could take an amount of time larger than the time frame available to have timely feedback. This is mainly relevant for agile practices, which prescribe both as many tests as possible, as well as developing components, integrating and testing the overall system several times a day, to ensure minimal design and correct execution [10, 47]. For this, during development, test execution time is curbed, and test cases to be executed have to be selected among available ones [72, 93, 116].

This study aims at automatically generating test cases tailored to the behaviour of the class under test. Our approach provides a class with a test by using static analysis to determine its behaviour, then such a behaviour is checked against the behaviour gathered for other classes, finally tests are generated starting from previously known tests for classes having a similar behaviour. Since generated tests are tailored to the code to be tested, they are effective for code coverage and for finding bugs. We have used our approach to generate tests for several software systems, whose source code is available. According to our experiments, the tests we have generated manage to extend the amount of code coverage significantly, both by executing new paths within classes that had some test cases, and by generating tests for classes that had not been previously tested.

The rest of the module is organised as follows. Section 5.1.1 describes how we perform the analysis of classes to gather their behaviour. Section 5.1.2 reports our approach for generating tests according to the knowledge on the behaviour of classes and other tests used as samples. Section 5.1.3 draws the conclusions.

5.1.1 Analysis of Software Systems

Generally, the developer creates a test case once he has gathered some knowledge on the expected behaviour of the class to be tested. Then, for each method of the class, he determines a set of inputs, among valid or invalid ranges for the needed parameters, and an expected output to be compared with the output provided by the method execution.

The proposed automatic analysis tool aims at revealing the behaviour of a class by extracting some characteristics of the static code. Moreover, existing test cases, implemented by developers, are analysed in order to be later used as templates

of test cases for some other classes. Figure 5.1 shows the essential components implementing the stages of the proposed approach: (i) the analysis of classes and the extraction of their main characteristics; (ii) the computation of similarity scores through comparison; (iii) the analysis of test cases aiming at finding useful templates, mainly for untested classes; (iv) the generation of the code implementing new test cases for selected classes.

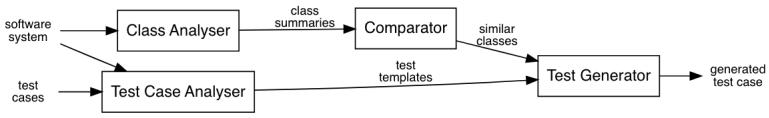


Figure 5.1: Main components realised the proposed approach for generating test cases: analysing classes, analysing test cases, comparing classes, and generating test cases.

For a software system, all the classes are analysed to find their behaviour, this is summarised starting from all the used APIs (component *Class Analyser* in Figure 5.1). APIs are classes and methods provided to the application by the underlying platform. E.g. the standard library APIs of the Java platform provides applications with useful classes and methods giving many functionalities such as generating random numbers, accessing the file system, etc. The methods of a class under analysis will then be marked according to the APIs used. This is a way to recognise that classes and methods of the APIs are an indication of the intent of the calling code [40, 39, 138]. E.g. a method using a variable having type *List* and using methods *isEmpty()* and *add()* on such a variable exhibits, at runtime, a behaviour comprising the update of the elements on such a list (Java APIs include interface *List*, which declares methods *isEmpty()* and *add()*).

Therefore, for each analysed class, all declared variables, and method calls are searched, then a list is created that holds all the APIs that have been found. Once classes have been characterised by the list of used APIs, they will be checked to

find out whether they have a comparable behaviour (component *Comparator* in Figure 5.1). Listing 5.1 shows the listing of two classes found in the RepoDriller software system [1]. Both classes use Java APIs *List* and *String*, hence both have been characterised by the list of APIs List, String.

[1] https://github.com/mauricioaniche/repodriller

Listing 5.1: Classes *OnlyDiffsWithFileTypes* and *OnlyDiffsWithoutFileTypes* implemented in RepoDriller software system, using the same Java APIs, and having the same signature for their method.

```
package org.repodriller.filter.diff;

import java.util.List;
import org.repodriller.util.RDFileUtils;

/**
* Only process diffs on files with certain file extensions.
*
* @author Ayaan Kazerouni
*/
public class OnlyDiffsWithFileTypes implements DiffFilter {
    private List<String> fileExtensions;

    public OnlyDiffsWithFileTypes(List<String> fileExtensions){
        this.fileExtensions = fileExtensions;
    }

    @Override
    public boolean accept(String diffEntryPath) {
        return RDFileUtils.fileNameHasIsOfType(diffEntryPath, this.fileExtensions);
    }
}

package org.repodriller.filter.diff;

import java.util.List;
import org.repodriller.util.RDFileUtils;

/**
* Only process diffs on files without certain file extensions.
*
* @author Ayaan Kazerouni
*/
public class OnlyDiffsWithoutFileTypes implements DiffFilter {
    private List<String> fileExtensions;

    public OnlyDiffsWithoutFileTypes(List<String> fileExtensions){
        this.fileExtensions = fileExtensions;
    }
```

In order to reveal whether a pair of classes have a comparable behaviour, we measure similarity according to Jaccard similarity coefficient. The latter, when measuring the similarity of two sets, is the ratio between the cardinality of the intersection and the cardinality of the union of the sets to be compared. For a pair of classes A and B, we measure the cardinality of the intersection of the two sets of APIs found in their code, hence used inside the respective classes (given as A_{API} and B_{API} , respectively), and the cardinality of the union of the two API sets. Then, the Jaccard similarity coefficient for the pair of classes is the ratio of the latter cardinalities.

$$J(A, B) = \frac{|A_{\text{API}} \bigcap B_{\text{API}}|}{|A_{\text{API}} \bigcup B_{\text{API}}|}$$

As such, our similarity measure finds whether a class pair has many or few APIs in common, where 1 is the upper bound and 0 is the lower bound for the measure. Once a pair of classes has been found to have a high degree of similarity (i.e. Jaccard similarity coefficient near to 1), a test that could be available for one class will be used as a template for the other one. For the pair of classes shown in Figure 5.1, the similarity measured is 1, as they both use the same Java APIs List and String.

The code of test cases is analysed to find the class that is the target of each test (component Test Case Analyser in Figure 5.1). Moreover, for each tested class, we reveal the list of the methods that are invoked by the test. Therefore, for a test case T we will have the list of the tested classes, e.g. *A, B, C* and the methods invoked on each class. Then, when a test case uses several classes, in order to determine which class is under test, we distinguish classes whose instances are used as parameters of method calls, from classes whose methods are invoked. We associate the test with the class whose methods are called. Therefore, a test for class A is found when in

the test code there is at least one method called on an instance of class A. As a result, the same test case could be associated with more than one class, when its code calls methods on instances of different classes.

The JavaParser library has been used to perform code inspection of classes and test cases, for generating tests [129]. JavaParser lets us navigate the source code as an Abstract Syntax Tree (AST) having a root representing the entire file, to which all code elements are connected, in particular each class declaration. From this, in turn, multiple nodes can be reached, which represent the fields or methods of the class.

Code inspection has been performed by using JavaParser's class *VoidVisitorAdapter*, which lets us define a *Visitor* class to search for a specific property. In the *Visitor* class, a visit() method is implemented, which takes as parameters the type of object being searched (class declaration, method declaration, etc.) and the container where data are saved; the method body will implement the operations to be carried out each time the element specified in the parameter is found. The Visitor class works within a CompilationUnit (representing a Java file), therefore it is necessary to associate the Visitor with a *CompilationUnit*. This association was carried out for all the files of the source code.

Additionally, code inspection was performed for each *CompilationUnit* using method *findAll(Class¡T¿ NodeType)*, which searches within a *CompilationUnit* all the elements that match the *NodeType* passed as parameter. We searched e.g. *MethodCallExpr* that represents method calls. For each expression found, some operations were performed in order to resolve the type found so as to trace its origin. This was useful to find all the APIs used within each class.

5.1.2 Test Generation

This section describes the component *Test Generator* shown in Figure 5.1.

Suppose that a pair of classes A and B has a high similarity: the idea is to generate tests for class B by replacing all occurrences of class A in the tests found for class A, with occurrences of class B. Of course, a test for class A would call methods implemented in class A. Hence, when using the test for class A as a template for class B, all the methods called have to be checked, and possibly changed.

Four possible cases have been identified to guide method replacement on a template test. Firstly, classes A and B have the same method signatures, this is the case when e.g. they implement the same interface. Secondly, classes A and B have methods having the same name but a different signature, i.e. the type and number of parameters differ. Thirdly, classes A and B have methods having different names with some resemblance to each other. Method names exhibiting some resemblance are, e.g., like the following: *getNumberOfValues()* and *getNumberValues()*; though these names differ, a human reader can easily recognise that they could be used for the same goal. Fourthly, methods having the same (or almost the same) input parameters, as their expected types, have some resemblance to each other.

To begin with, classes are automatically inspected to find the list of all their methods, together with each associated signature.

As for the first case, when the methods of the classes in a pair of similar classes have the same signature, tests are generated by simply replacing the name of class A with the name of class B. This is also a necessary step for test generation when the following second case occurs.

For the second case, the methods in the pair of classes have the same name, however different parameters. Then, parameters will be replaced as follows. When

the parameters used by the method call on the template test are a superset of the needed parameters for generating the new test, then the method call will be rewritten by reducing the number of parameters and selecting those needed, in the proper position. When new parameters are needed for the method to be called, i.e. they are not contained within the parameters of the original method call, then a method call is written that generates new parameters with default values, in order for the test code to be syntactically correct.

The third case involves a different analysis, it consists in having to call a method on the generated test that is not in the original template test. In order to find the method that will replace that occurring in the original method call, a method with a similar name will be selected. As a general rule, each method name will be decomposed into its constituting words, according to conventions on the use of upper cases, to form a *bag of words*. Hence, e.g. a method *getNumberOfValues()* will produce a bag of words consisting of *get, number, of, values*. Then, set comparisons will be made to find the possible substitution method, selected among the most similar ones, on the basis of the Jaccard similarity index for the bag of words of each method. When computing similarity between the above method and method *getNumberValues()*, the intersection set will have cardinality 3 and the union set will have cardinality 4, hence similarity among methods will be determined as 3/4=0.75. In this example, the two methods are highly similar, and can be interchanged for the sake of test generation.

The fourth case consists in finding which pair of methods from two different classes have a similar set of input parameters according to their type. Here, we list for each method of the pair the types of its parameters and find the Jaccard similarity index among the two sets of types.

For the purpose of generating a new test starting from a template test, the replacements of class and method names were performed by means of the JavaParser *LexicalPreservingPrinter* class, which yields Java code that is formatted correctly and in a standard style.

When writing code, some elements not meaningful for the compiler are however important to humans, such as. e.g., indentation and comments, and should be preserved. The above class provides a method *setup(Node node)*, which prepares the node to be printed. The node that is passed represents the *CompilationUnit*, it is therefore possible to make changes to the nodes (e.g. change the name of the class, change the name of the methods, insert comments, etc.), hence transforming the original code. Method *setup()* indicates the point from which all the changes will be made, which will then result in an output using method *print(Node node)*. Printing generates a new Java file containing the transformed code. Code writing was used for the generation of tests according to the substitutions described above (see Section 5.1.1).

5.1.3 Summary

This study has proposed a novel and effective approach capable of generating test cases which takes advantage of the knowledge gathered from the static code of classes to be tested and from the static code of existing test cases. As such, computation time for generating test cases is limited (depending on the amount of code to be analysed, and not on its execution time).

We have shown that the devised strategies for comparing classes and finding the pairs with similar behaviour, in order to reveal which test templates are applicable, are effective, since many analysed software systems have exhibited numerous

enough pairs of classes with similarities higher than 0.5 (the mid value was chosen as a threshold). Moreover, generated test cases were a significant number for the analysed software systems, showing that the proposed substitutions of class and method names performed on existing test cases (taken as templates) are a viable solution. Test cases generation managed to include classes that had not been previously tested, hence increasing code coverage and enhancing the effectiveness of tests.

This approach generates tests for classes that have not been previously tested. In order to generate more comprehensive and effective tests, we have designed an approach to generate templates of test cases for untested methods. These two approaches can be used together to provide developers with a complete test generation framework for Java application. The following Section describes this last module of the testing framework, highlighting the approach and the results of the experiments.

5.2 Automatic Generation of Accurate Test Templates based on JUnit Asserts

An effective test suite can validate the correctness of a software system and improve its maintainability, reusability and robustness over time [118, 61, 124]. Moreover, a test suite can help developers check that enhancements and refactoring activities do not introduce defects [103, 13]. However, manually implementing a test suite could be challenging and time consuming, since the developer should produce at least a test case executing each method and feature in different scenarios. Conversely, poor test coverage decreases the effectiveness of a test suite, leading to the misidentification of defects [91]. The state of the art provides several approaches to assist developers

during software testing. The selection of proper input values can be a critical task to fulfil when implementing test cases, indeed a broad and accurate spectrum of values can improve the quality of the test suite. E.g., a combinatorial approach for the selection of input values can be very effective [14, 22]; moreover, a solution that automates the selection of expected output values to check the desired behaviour could make use of a proper model that represents the desired system [23].

For generating the code of a test suite, many tools have been proposed. Some tools focus on the random generation of a sequence of calls and values that constitute a test case [15, 36, 99, 119]. Generally, random testing tools generate many test cases, and consequently their execution time increases significantly, then developers need some criteria and tools to select among these a smaller set of test cases [66, 4]. Moreover, generated test cases can often carry several test smells, lowering the effectiveness and quality of the test suite [102]. Test cases that have been generated by a random approach can be improved by removing duplicate and redundant tests while keeping the same code coverage [114, 94, 77]. One of the most effective test generation tools is Evosuite [46, 145], which is search-based and uses evolutionary search to automatically generate test suites aiming at maximising code coverage [101]. Although Evosuite has greater coverage and accuracy than other generation tools, test generation takes considerable time, i.e. on two time budgets of 30 and 120 seconds, tests achieve 65.7% and 77.1% of mean line coverage respectively [145]. Moreover, a large amount of computational resources and time are needed to properly run it, otherwise several OutOfMemoryException crashes could occur, or the generated test suite exhibits low coverage [145]. The need to increase usability and readability of generated tests has been pointed out, as generated tests are less readable than manually implemented ones [115, 51].

This study aims at automatically generating a set of test case templates for methods of an application that have not been previously tested. The template contains the method to be called, a JUnit assert, and as many input parameters as possible. Our proposed approach uses statistics on the most used JUnit asserts for each return type of a method, which have been gained by means of a static analysis on software repositories including test suites. The statistics on return types of methods have been collected for most of the standard Java library methods. Then, for an application that needs to have (more) test cases, our approach (and corresponding tool) generates a customised test case template for each method, selecting the assert statistically more relevant for the return type of the method to be called.

We have used our approach to generate test templates for an open source github repository. According to our experiments, the templates generated have extended the amount of code coverage significantly, providing tests for methods not previously tested, and increasing the number of tested classes, methods and paths. Furthermore, the generated templates are easy to understand and versatile, since the generated assert directly tests a method, unaffected by boilerplate code, and gives developers the possibility to easily customise it. As a result, the developer is freed from the implementation of most portions of test code, as she only needs, in some cases, inserting input parameters and an expected value, when the provided tool cannot determine these automatically. In the scenario where our approach is used to support refactoring activities, as proper tests are needed to check whether code changes have kept the behaviour unmodified, our corresponding tool generates a complete test case for every refactored method, when applying the refactoring technique Extract Method. In such a case, the generated test case will use as expected

value for the new method the returned value of the original method.

The rest of the paper is organised as follows. Section 5.2.1 describes the analysis performed to gather the data needed for generating test case templates. Section 5.2.2 reports how the template generation is carried out. Section 5.2.3 wraps up the presented module.

5.2.1 Analysis of Software Systems

Implementing test cases is an activity that takes some time and some effort from developers. The more automated test development the better. The proposed approach assists in test cases implementation, hence obtains automation. Statistics are gathered from classes and methods found from repositories, and from their test cases. When parsing all Java files, separating application source code from test one is possible thanks to the typical project structure (usually the source code is organised in two directories: *main* and *test*). The analysis has been arranged in three main steps: (i) software systems, selected from repositories, are parsed to determine how JUnit asserts were used and how they relate to each method called and to the return types of the method called; for this, test code was separated from application code by relying on the structure of the software system; and within this analysis, then (ii) return values for some Java APIs such as Collections and String were further characterised as they need a specific check in the used assertion than that of other types; then, (iii) applications for which tests are needed are parsed to collect their list of methods (with their signature) and tests are generated for them.

Figure 5.2 shows the steps followed by the proposed approach. Steps 1 and 1.1 are performed once and data gathered serve for subsequent test generations. However, when more software repositories become available that are representative of the use

Figure 5.2: Flowchart representing the main steps followed by the approach generating test cases.

of asserts the said two steps could be executed again. The other steps (2, 2.1 and 3) are performed each time new test templates need to be generated for an application. The following subsection details steps 1 and 1.1; which present similarities with steps 2 and 2.1. Parsing activities detailed below are based on the Javaparser library: an automatic parser that takes as input one or more .java files and generates an abstract syntax tree (AST) for each, providing means to perform operations on ASTs, such as reading, inserting, deleting and updating [129]. Two customised visitors have been implemented, the first to visit test directories and collect statistics on asserts (step 1 in Figure 5.2); the second to visit main directories for gaining method signatures for which tests will be generated (step 2 in Figure 5.2).

One of the most used and reliable approaches for unit testing in several programming languages is the use of assertions. An assertion represents the point in the code where the expected value and the returned value of a method call are compared. A high assertion coverage, i.e. the percentage of statements directly covered by assertions, has a strong correlation with the effectiveness of a test-suite [157]. An open source testing framework for Java is JUnit, which provides several APIs for simplifying test case implementation, and includes several types of assertions. Most of the assertions take two input parameters, the first one is the expected value and the second one is the value returned at runtime by the called method. There are other assertions, e.g. the assertTrue(actualValue), which take as input just the returned value, which in the assertTrue() must be a boolean and the expected value is true.

We have collected the frequency of each assert (step 1.1 in Figure 5.2) used by test cases for the types returned by called methods for each assert, e.g. the frequency of return type String tested with assertEquals(), etc., such frequencies are later used to guide test generation (step 3 in Figure 5.2). We have found 8058 assertions in six real-world Java projects having different sizes (see Section 6.6).

Listing 5.2 shows four examples of asserts containing method call expressions or a variable (they have been found in logback github repository, The first shows an assertTrue() that checks whether the value returned by the method call checkError() is true, hence we can easily identify the checkError() method as the one tested by the test case. The second describes an assertEquals() of a variable; then to understand whether such a variable is the result of a method call, or the result of a chain of method calls, we look for the variable declaration and its assignment. The third and fourth examples show a more complex situation, where the actual value tested is a compounded expression; hence to correctly extract the return type tested, we have performed some additional activities.

Listing 5.2: Four examples of asserts with different arguments from the logback repository.

```
// The argument is a MethodCallExpression
1. Assertions.assertTrue(checkError())
// The argument is a variable
2. Assertions.assertEquals("c", result)
// The argument is a chain of MethodCallExpression or FieldAccessExpr
3. assertTrue(listAppender.strList.get(0).startsWith("testMethod"))
4. assertEquals(1, listAppender.strList.size())
```

The last two examples in Listing 5.2 show two asserts whose actual value is a method call chain. To properly extract the type tested by the assert, we consider the last two elements of the chain: the method call and its scope. If we considered just the last element, we would have had types like *Integer*, *Float* etc., therefore

missing some occurrences of types such as *String* or *Collections*. The third example in Listing 5.2 is a chain of *FieldAccessExpr* and *MethodCallExpr*; we select the startsWith("testMethod") method call, and we check the type of its scope, the method call *get(0)*, which returns a *String*; we can then store the triple (*assertTrue*, *String*, *startsWith*()), where starting from left we have the name of the assert, the type tested and the chain used. The fourth example is like the third, the only difference is that the scope of the method call is a variable. According to the above, we have designed a visitor that automatically parses the code of all test classes inside the application and looks for test cases. Then, for each test case found, asserts are collected and stored as a pair composed by the type of the assert (e.g., *assertTrue()*, *assertEquals()*, etc.) and the type tested, i.e. the type returned by the method call or the variable assigned within the assert.

This approach allows us to better generalise our analysis, gathering more data for our template generation. We can create more complex templates and guarantee the generation even for complex types like *Collections*. We decided to include this category to our analysis because it is widely used by many projects, unlike other libraries that can differ according to the project needs [28].

5.2.2 Test Template Generation

This section describes the generation of a test suite (from scratch) for applications, to increase the number of tests available. Our tool generates a set of test templates for classes and methods that have not already been tested (step 3 in Figure 5.2). We combine the analysis of code, described in Section 5.2.1, with the results given by JaCoCo (www.eclemma.org/jacoco), an open source library for Java that automatically performs code coverage analysis. The output of JaCoCo is a set of html pages

showing the code coverage of each method's instructions and branches. We filter the methods that our code analysis as found as not-tested (i.e. they do not appear in any test case), with the output given by JaCoCo, to extract just the methods whose lines of code are not fully covered by any existing test, and generate a template test for each of them.

To generate the test template, we first determine which assert can be the most accurate to test that specific method. We base our evaluation on the statistics gained from the test case analysis on several software repositories (described in Section 5.2.1). We select the assert with the highest frequency for the type returned by the method and use it for the generation of the template. If there are asserts having a close frequency, we will generate as many templates as many close asserts for the same method.

Then, to make the generated test case complete, the proper input values and expected values are needed. To help the developer, we have three solutions for the selection of values. Firstly, our tool can be given a .CSV file where each line is the triple: method's signature, input arguments (if any), and the expected value in the assert; then our tool will automatically generate the template filled with data. Secondly, when the system is being refactored, we use the original method (before refactoring) and its test code to gather information regarding the parameters and values used. As for refactoring the behaviour should be unchanged, the input and the output values remain effective and can be included in our generation. Thirdly, we integrated our approach with an automatic value generation tool. We have selected Java Faker (github.com/DiUS/java-faker), which automatically returns a value for a variable based on its name and type, e.g. *String streetAddress = faker.address().streetAddress(); // 60018 Sawayn Brooks Suite 449*, where

the variable streetAddress is associated with the value "*60018 Sawayn Brooks Suite 449*". Therefore, we try to guess proper values needed for test templates according to names of parameters and methods.

5.2.3 Summary

This module and the one presented in 5.1 form the testing framework. The developers can use these modules to generate an effective test suite for their application, significantly reducing the time needed for writing all tests. Moreover, this framework can be used together with the refactoring framework to generate an improved version of their application.

This module and the one presented in Section 5.1 form a comprehensive testing framework that developers can use to generate an effective test suite for their Java applications. This can significantly reduce the time needed for writing tests manually.

In addition, the testing framework can be used in conjunction with the refactoring framework presented in Sections 3, 4 to generate an improved version of the application. For example, the refactoring framework can be used to identify and refactor methods to a parallel version, which can improve the performance of the application. The testing framework can then be used to generate tests for the refactored code, ensuring that it still works as expected.

Overall, the testing framework provides a valuable tool for developers to improve the quality of their Java applications.

Chapter 6

Experiments and Results

6.1 Data Dependence API

We conducted experiments using our custom library on seven open source projects, selected from among the most popular repositories available on the Maven repository (mvnrepository.com/popular). The source code for each of these repositories is accessible on GitHub. The gathered metrics for the seven analysed projects are presented in Table 6.1.

The first column of the table displays the names of each software system. Moving from the second column onward, the metrics are as follows: "Total" represents the total number of methods analysed; "Stateless" indicates the number of methods categorised as 'STATELESS' at the end of the analysis; "Read" shows the number of methods categorised as 'READ'; "Write" corresponds to the number of methods categorised as 'WRITE'. The subsequent six columns display metrics for both 'READ' and 'WRITE' methods. In the case of 'READ' methods, the "Both" column represents the number of methods categorised based on both factors: 'READ' method calls and 'READ' operations on external variables. On the other hand, the "Method" and "Variable" columns represent the number of methods categorised

based solely on method calls or used variables, respectively. Similar columns are provided for 'WRITE' methods.

The metrics reveal that 'READ' and 'WRITE' methods account for over 80% of the total number of methods across most projects. An exception is observed in the JavaHamcrest project, where 'READ' methods make up 80% of the total. In many analysed projects, 'STATELESS' methods constitute approximately 10% of the total number of methods, indicating that they can be readily executed in parallel. For other methods, a synchronisation mechanism is required before they can be run in parallel due to their method calls or access to shared data.

Methods categorised exclusively based on method calls and external variables underscore that data dependence is not solely determined by the use of non-local variables. Instead, method calls also carry significant weight when considering parallel execution. The size of the output set may provide an indication of the effort required to implement synchronisation mechanisms for parallel execution.

Table 6.2 displays the number of methods that write varying numbers of variables, as indicated at the top of the respective columns. The columns range from "1" (representing the number of methods with just one external variable written) to "7+" (representing the number of methods with 7 or more external variables written). The analysis indicates that nearly 80% of methods write just one external variable, with the exception of the Clojure project, where this figure is 64%.

Now, we analyse three methods extracted from the Jackson-Databind project (available at github.com/FasterXML/Jacksondatabind). In Figure 6.1, you can see the 'serialize()' method, categorised as 'WRITE' due to its inclusion of both a write operation on the 'ser' variable and a write method call to `'_findAndAddDynamic()'` (the left side displays the source code, while the right side exhibits the report

Table 6.1: Metrics for each category of methods in the analysed software systems

		category			read methods			write methods		
analysed systems	total	stateless	read	write	both	meth	var	both	meth	var
jackson-databind	10726	1214	6770	2742	1622	1327	3821	252	1726	764
joda-time	8242	985	5103	2154	2301	1728	1074	268	1760	126
logback	5420	526	2372	2522	568	366	1438	323	1616	583
junit4	3708	937	1868	903	786	524	558	46	494	363
clojure	2554	330	1511	713	658	140	713	164	418	131
JavaHamcrest	978	102	817	59	338	196	283	1	56	2
repodriller	412	30	232	150	55	15	162	12	81	57

Table 6.2: Number of methods performing write operations to one or more external variables

systems	total	**1**	**2**	**3**	**4**	**5**	**6**	**7+**
jackson-databind	1016	863	98	32	17	2	1	3
logback	906	746	97	32	16	10	2	3
junit4	409	382	23	0	0	2	2	0
joda-time	394	223	60	79	11	8	1	12
clojure	295	193	57	20	8	5	1	11
repodriller	69	55	6	2	4	0	0	2
JavaHamcrest	3	2	1	0	0	0	0	0

generated by our analysis).

Figure 6.2 presents a method whose category is determined exclusively by a write operation. Despite containing a call to a method labelled as 'READ' (i.e., 'getParent()'), its category is solely influenced by the write operations on the listed external variables, as explained in Section 2.

Finally, Figure 6.3 showcases a method whose category is determined by the method calls 'createUsingDefault()' and 'deserialize()', both categorised as 'WRITE' methods. The report also provides information about the "Callers," which includes all methods calling the current method. This detail offers valuable context for understanding how the method will be executed.

Listing 6.1: On the left, the method *serialize()* from the Jackson Databind project, on the right, all the information extracted by our library. Method's category is given by a write operation on the external variable *ser* and a write method call *_findAndAddDynamic()*

```
public void serialize(Object value, JsonGenerator g,
    SerializerProvider provider) throws IOException {
    Class<?> cls = value.getClass();
    PropertySerializerMap m = _dynamicSerializers;
    JsonSerializer<Object> ser = m.serializerFor(cls);
    if (ser == null) {
        ser = _findAndAddDynamic(m, cls, provider);
    }
    ser.serialize(value, g, provider);
}

Qualified name:
com.fasterxml.jackson.databind.ser.std.StdKeySerializers.*Dynamic.serialize
Category: WRITE
External read variables: [provider, g, cls, _dynamicSerializers, value]
External write variables: [ser]
Write Method Calls:
com.fasterxml.jackson.databind.ser.std.StdKeySerializers.Dynamic.
    _findAndAddDynamic
External read methods:
java.lang.Object.getClass
Callers:
com.fasterxml.jackson.databind.deser.builder.BuilderWithUnwrappedTest.
    testWithUnwrappedAndCreatorSingleParameterAtEnd
```

Listing 6.2: On the left, the method *nextToken()* from the Jackson Databind project, on the right, all the information extracted by our tools. Method's category is given by write operations on three external variables: *_nodeCursor, _currToken, _closed*

```
public JsonToken nextToken() throws IOException {
    _currToken = _nodeCursor.nextToken();
    if (_currToken == null) {
        _closed = true; // if not already set
        return null;
    }
    switch (_currToken) {
    case START_OBJECT:
        _nodeCursor = _nodeCursor.startObject();
        break;
    case START_ARRAY:
        _nodeCursor = _nodeCursor.startArray();
        break;
    case END_OBJECT:
    case END_ARRAY:
        _nodeCursor = _nodeCursor.getParent();
    default:
    }
    return _currToken;
}
```

```
Qualified name:
com.fasterxml.jackson.databind.node.TreeTraversingParser.nextToken
Category: WRITE
External read variables: [START_OBJECT, END_ARRAY, END_OBJECT, START_ARRAY]
External write variables: [_nodeCursor, _currToken, _closed]
Read Method Calls:
com.fasterxml.jackson.databind.node.NodeCursor.getParent
Callers:
com.fasterxml.jackson.databind.DeserializationContext._treeAsTokens
com.fasterxml.jackson.databind.cfg.MutableCoercionConfig.
    setAcceptBlankAsEmpty
```

Listing 6.3: On the left, the method *deserialize()* from the Jackson Databind project, on the right, all the information extracted by our tools. Method's category is given by write method calls, *createUsingDefault()* and *deserialize()*

```
public T deserialize(JsonParser p, DeserializationContext ctxt) throws IOException {
    if (_valueInstantiator != null) {
        @SuppressWarnings("unchecked")
        T value = (T) _valueInstantiator.createUsingDefault(ctxt);
        return deserialize(p, ctxt, value);
    }
    Object contents = (_valueTypeDeserializer == null)?
    _valueDeserializer.deserialize(p, ctxt):
        _valueDeserializer.deserializeWithType(p, ctxt, _valueTypeDeserializer);
    return referenceValue(contents);
}

Qualified name:
com.fasterxml.jackson.databind.deser.std.ReferenceTypeDeserializer.
    deserialize
Category: WRITE
External read variables: [p, ctxt, _valueTypeDeserializer, _valueInstantiator,
    _valueDeserializer]
Read Method Calls:
com.fasterxml.jackson.databind.JsonDeserializer.deserializeWithType
com.fasterxml.jackson.databindJsonDeserializer.deserialize
Write Method Calls:
com.fasterxml.jackson.databind.deser.ValueInstantiator.createUsingDefault
com.fasterxml.jackson.databind.deser.std.ReferenceTypeDeserializer.
    deserialize
Callers:
com.fasterxml.jackson.databind.ser.TestSimpleTypes.testShortArray
```

6.1.1 Discussion

In table 6.1, the number of stateless methods represents just 10% of the methods. This data highlights how most of the methods in a project have some dependencies, indeed they share reading or even writing operations with other methods. Therefore it is necessary to properly check the dependencies of a code fragment before refactoring it, e.g. executing the statement in parallel. The proposed API can effectively help developers in these tasks, which could be complicated and sensitive if done manually.

Read methods represent the majority in the analysed methods (except for *logback* which is the only one where write methods are higher than read methods). The analysis highlights how the dependencies between methods are not always strict, since if two methods read the same variable without writing it, they can be easily run in parallel without further synchronisation despite having at least one variable in common. Correctly identifying this kind of dependencies is crucial when developing parallel architectures, indeed, an error in identifying a read variable in a write variable would lead to the insertion of a synchronisation process which would not be necessary, it could burden the code and even worse the performance of the application.

Despite the analysis taking into account both method and variables to detect all the dependencies, methods defined in external libraries are not analysed since the source code is not available for the static analysis. We have mitigated this by creating a .txt file containing all the qualified signatures of the encountered methods, by manually labelling them as stateless, read or write. Methods that are not inside the list are reported to the developer in order that he can update the file, broadening the number of handled methods.

The approach uses static analysis to gather all the data required for the dependency analysis. Accordingly, all the types referring to interfaces cannot be inferred since the actual type will be instantiated during the execution. In order to keep these cases in our analysis, we select all the methods defined in the subclasses of the interface, and we assign to the type the worst label found according to the state machine (figure 3.1. E.g., the approach is analysing a method to understand its dependencies: a method call defined in an interface is found, we do not know which class will be instantiated during the execution, therefore we select all the methods with the same signature defined in the subclasses and select the one with the highest label.

The determination of the quantity of writing variables within a method holds significant importance in assessing the effort needed for parallelising that method. Specifically, a greater number of writing variables within a method necessitates a higher count of synchronisation statements to enable the method's effective parallel execution. Furthermore, it is advisable for developers to steer clear of creating methods with such dependencies, as this results in increased code coupling, thereby diminishing its reusability. Table 6.2 illustrates that the majority of methods typically involve just one external write variable. However, there exist a notable subset of methods that encompass more than three variables, which may indicate suboptimal programming practices and a limited inclination towards parallelisation.

It is essential to comprehend the labelling of a method to effectively characterise its behaviour and dependencies (Figures 6.1,6.2,6.3). Methods categorised by a variable indicate a data dependency, signifying a connection between two statements that access or modify the same resource. In contrast, methods categorised

by another method represent a control dependency, wherein a program instruction is contingent on the preceding instruction's evaluation outcome for execution. Distinguishing between these two types of dependencies is crucial for a comprehensive understanding of the method under examination. This distinction aids in the accurate assessment of synchronisation requirements when considering potential parallelisation.

6.2 From Sequential to Parallel

Our experiments were designed to automatically analyse and modify an application while assessing the correctness of the results and measuring performance improvements. We conducted these experiments on a sample application that extracts data from the Amazon Books Reviews dataset [31] to derive insights on books, authors, and reviews. The source code of the analysed application is publicly available in a repository (https://github.com/AleMidolo/BookReviews, accessed on 21 July 2023). This dataset comprises approximately 3 million book reviews, covering 212,404 unique books, with multiple users providing reviews for these books. To ensure reasonable execution times for testing the correctness of the transformations and evaluating performance, we selected a subset of 166,667 reviews for analysis.

Among the 44 methods found within the six classes of the application under examination, our tool automatically identified and refactored five methods into parallel versions. One of these methods, getUserForAuthor(HashMap¡String, Book¿ books, List¡Review¿ reviews), is illustrated in Listing 6.4. The automatic analysis of this method proceeded as follows: the method call getAuthors() at line 2 (Listing 6.4) was initially selected for analysis. The context of this method call was evaluated,

and further assessment was conducted as the context matched one of the feasible cases outlined in Section 3.2.2; data dependence analysis identified the instruction at line 4 as data-dependent. This determination was made because the authors list was populated by the method call at line 2 and subsequently used at line 4; the Control Flow Graph (CFG), which consisted of statements within the called methods, was constructed. Conditional branches and loops were analysed to determine whether any adjustments were needed (see Section 3.2.4). However, in this case, the code did not contain any if/else statements, resulting in a CFG with no alternative paths. Additionally, there were no loops present in the CFG; potential parallel execution paths were identified; finally, the paths, including line 2 (along with the CFG of the called method) and line 3 (also including the CFG of the called method), were automatically assessed to count the number of instructions comprising each path. This evaluation helped determine whether executing these paths in parallel would yield performance improvements.

Listing 6.4: This code extracts the authors from the books and the users from the reviews, then it assigns to each author all the users that have provided at least one review for the author's books.

```
1       ExtractData extractor = new ExtractData(books, reviews);
2       HashMap<String, Author>  authors = extractor.getAuthors();
3       HashMap<String, User> users = extractor.getUserForAuthor();
4       authors.values().forEach(author -> {
5               List<String> usersId = author.getBooks().stream().
6                       flatMap(b -> b.getReviews().stream().
                                map(r -> r.getUserID())).
7                       collect(Collectors.toList());
8               usersId.forEach(u -> author.addUser(users.get(u)));
9       });
10      return authors;
    }

// parallel version of some instructions above
2    CompletableFuture<HashMap<String, Author>>  f =
         CompletableFuture.supplyAsync(() -> extractor.getAuthors());
3    HashMap<String, User> users = extractor.getUserForAuthor();
4    HashMap<String, Author>  authors = f.get();
```

Figure 6.1 shows a portion of the CFG for the code in Listing 6.4, from line 1 to line 5. The paths that should be executed in parallel are (i) the instruction at line 2; (ii) the instruction at line 3. Line 4 presents a statement that depends on the output of line 2. The first path has nine instructions, eight are the instructions contained in the CFG_2, of which, six are inside a loop. The second path has 10 instructions, 11 are the instructions contained in the CFG_3, of which, 5 are inside a loop. The method is suitable for parallel refactoring because the instruction numbers are sufficiently large and both CFGs found contain a loop. Therefore, a new version of the method has been generated that contains the constructs for parallel execution.

Such a version can be seen in the bottom part of Listing 6.4; the method call was given as a lambda expression in the CompletableFuture call supplyAsync(); just after line 3, the get() call was introduced to synchronise the execution and create the authors HashMap.

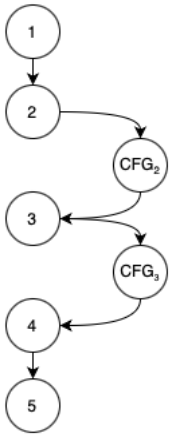


Figure 6.1: CFG extracted from the method's graph *getUserForAuthor()* shown in Listing 6.4.

We assessed the performance of the generated refactored parallel version by using Java Microbenchmark Harness (JMH), a standard performance test harness that provides APIs to write formal performance tests. We configured the library to have warm-up cycles for performance measurements; each run had five warm-up iterations and five normal iterations. JMH tests are the best indicators of performance improvements since they isolate the subject that is evaluated, avoiding influences by other subjects. Table 6.3 shows the resulting benchmarks of the methods that were automatically refactored by our approach. Method column presents the name of the refactored method; Time (ms) column displays the execution time in milliseconds for the method; Speed-Up column is the speed-up obtained by the parallel version (computed as runtimeold/runtimenew); finally, the sub-columns Sequential

and Parallel show the respective measured execution times for the sequential (runtimeold) and parallel (runtimenew) versions. The average overhead measured for each CompletableFuture call was 7.77 ms, calculated on 10 different runs and after 10 warm-up runs. The overhead represents the time needed for the JVM to create and execute the new thread that will handle the parallel path; it was computed as the difference between times for the execution of the parallel and sequential versions. The measured performance will be further discussed in Section 6.2.1.

Table 6.3: Execution times for all five methods refactored for the sequential and parallel executions.

Method	**time (ms)**		**speed-up**
	sequential	**parallel**	
extractFromDataset	36K	17K	2.11
extractMostReviewedAuthor	485	330	1.46
extractLeastReviewedAuthor	522	318	1.64
extractAverageReviewedAuthor	533	317	1.68
getUserForAuthor	700	486	1.44

To validate the accuracy of our analysis and transformation in the case study, we created a test suite to ensure that the generated parallel version maintained the same behaviour as the sequential one. We utilised the JUnit framework to write and execute all the tests. Table 6.4 presents the tests implemented for the five refactored methods. The Description column provides a concise description of what each test verifies, while the Result column displays the values returned by both sequential and parallel executions.

Each test includes an assertion in which the expected value is derived from the output of the sequential execution of the method, while the actual value comes from the parallel execution. In all executed tests, the results produced by the parallel version matched those of the sequential version. Specifically, for the *getUserForAuthor()*

method, the only one returning a HashMap, all values within the HashMap were identical in both versions. Similarly, the same consistency holds for the books and reviews extracted by the *extractFromDataset()* method.

Table 6.4: Tests executed to validate the correctness of the refactored parallel code.

Description	**Called method**	**Result**
number of books	*extractFromDataset*	212,404
number of reviews	*extractFromDataset*	333,335
all books are equals	*extractFromDataset*	true
all reviews are equals	*extractFromDataset*	true
most reviewed author	*extractMostReviewedAuthor*	Lois Lowry
number of reviews	*extractMostReviewedAuthor*	3822
number of books	*extractMostReviewedAuthor*	8
least reviewed author	*extractLeastReviewedAuthor*	John Carver
number of reviews	*extractLeastReviewedAuthor*	0
number of books	*extractLeastReviewedAuthor*	1
average reviewed author	*extractAvgReviewedAuthor*	Mark Bando
number of reviews	*extractAvgReviewedAuthor*	14
number of books	*extractAvgReviewedAuthor*	2
number of authors	*getUserForAuthor*	127,279
all authors are equals	*getUserForAuthor*	true
all users are equals	*getUserForAuthor*	true

6.2.1 Discussion

To comprehensively assess the effectiveness of our approach, we conducted a performance benchmark to measure the actual improvement in execution time, along with a suite of tests to verify the correctness of the applied transformations.

The average speed-up across all five benchmarked methods was 1.66. This ranged from the highest speed-up in *extractFromDataset()* at 2.11 to the lowest in *getUserForAuthor()* at 1.44. It's important to note that the performance of the parallel version, and consequently the achieved speed-up, is significantly influenced by the workload distribution between the two parallel paths. When the execution times of

the two paths are closely aligned, maximum gains are realised as the paths can run concurrently without waiting for each other.

As elaborated in Section 3.2.4, our analysis constructed a Control Flow Graph (CFG) to estimate the number of operations for each path. We precisely determined the number of operations and identified the presence of loops but did not estimate execution times. Consequently, some disparities in execution times between the two paths might occur, potentially impacting performance gain when the path with less work has to wait for the other to complete.

The five benchmarks conducted for the application yielded varying speed-ups (refer to Table 6.3). Notably, for the extractFromDataset() method, the two parallel threads exhibited closely aligned execution times, resulting in a substantial performance gain. Conversely, methods with speed-ups of 1.44 and 1.46 experienced more significant differences in the execution times of the two parallel paths. Despite these variations, our transformations were proven effective, as they consistently delivered substantial performance improvements.

The *extractFromDataset()* benchmark required a longer execution time compared to the others due to multiple I/O operations to read data from .csv files. As detailed in Section 6.2, these files contained over 200,000 records for books and 300,000 records for reviews.

Table 6.4 provides insights into the tests conducted to validate the correctness of our applied refactoring. The Result column presents values returned by both sequential and parallel versions, demonstrating that, in each execution, the results remained consistent. This affirms that the control flow and data dependence analyses (Section 3.2.3) accurately identified dependencies and that synchronisation statements were correctly inserted. A misidentification could have resulted in desynchronisation and

test failures.

We assessed the code coverage of our test suite using JaCoCo (https://www.eclemma.org/jacoco/, accessed on July 21, 2023), an open-source Java library for automated code coverage analysis. Our test suite achieved a total code coverage of 94%, with only catch branches for try statements remaining unexecuted. All branches executed in both sequential and parallel modes were covered, confirming the correctness of the transformation. The code coverage analysis output for the analysed application is available in our GitHub repository.

The measured overhead represented approximately 0.05% of the parallel execution time for the *extractFromDataset()* method and between 1.5% and 2.5% for the other methods, minimally impacting parallel executions.

Our approach focuses on ensuring both the correctness of transformations and performance enhancements while minimising potential overhead from parallel threads. However, since our approach relies on static code analysis, there are specific scenarios that cannot be fully evaluated.

Firstly, if we identify subsequent instructions or blocks of instructions with low estimated computational efforts and data dependence, we refrain from transforming the code into a parallel structure. While this prevents unnecessary clutter and synchronisation in situations where performance gains are uncertain, it's essential to note that our analysis relies on estimated computational efforts. These estimates may be inaccurate in cases involving cycles, unknown repetition numbers, or calls to methods from external libraries, potentially causing us to miss some performance improvement opportunities. To address this, we could explore ways to enhance execution time estimation and consider developer input.

Secondly, static analysis alone cannot predict which method will be executed

at runtime in cases involving polymorphism. In such instances, we err on the side of caution, assuming the least favourable data dependence and performance gains. Consequently, there might be instances where potential opportunities for parallel execution are missed.

Thirdly, when a code fragment involves multiple references to objects of the same type, distinguishing between different instances can be challenging through static analysis, especially when a variable is conditionally assigned to one of several references. This can result in missed opportunities for parallel execution of distinct objects, potentially limiting performance gains.

Lastly, our approach may overlook scenarios where instruction reordering is possible, allowing dependent statements to be moved farther apart while maintaining correctness. These situations represent another area where performance gains could be missed.

It's worth emphasising that we have incorporated these limitations into our analysis and tool to prioritise correctness when executing the transformed code. This approach may result in a few cases where performance is not maximised but ensures that the behaviour remains correct.

6.3 Java Loops to Stream

The analysis was performed on a sample of eight software systems, which were selected from Maven Repository among the most popular ones. Table 6.5 shows, from the second column: count of for loops that iterate a collection; count of loops having references to local variables defined outside the loop; count of loops having more than 5 lines of code (LOC) in their body; count of loops having at most 5 LOC

in their body, which will be further analysed; then the five rightmost columns give the number of loops matching the five proposed templates: column *fret* represents for loops having a conditional statement and a return statement; *mret* shows for loops having an assignment, a conditional and a return statement; *temp* represents for loops having a temp and a condition on the current value; *add* shows for loops having a temp and producing a collection; finally, column *if&else* represents for loops having a if-then-else-if statement.

For the experiment we have considered only for loops having at most 5 LOC, because more lines means a greater difference from the above templates. Such loops represent about 79% of the existing loops, while the loops having more than 5 LOC represent about 4%; the remaining 17% is in the local column. The most common matching template found was the one creating a new collection (column collect) being found on average in 18.7% of loops having at most 5 LOC (we can transform them using the template described in subsection 2.4). The least common loop template was the one having a if-then-else statement (column if-else, template in subsection 2.5) found on average in the 2% of loops.

All the matching templates amount to 42.87% of the loops having at most 5 LOC. This means that nearly half of the loops have been successfully refactored. The remaining 57.13% are loops that: (i) could be handled by implementing an approach similar to that in [45], however we avoided that as our templates are all novel contributions, or (ii) cannot be transformed into stream-based code as they have a complex structure, e.g. loops with multiple statements in their body, which are far from the functional programming style.

systems	tot	lcl	>5	≤5	fret	mret	temp	add	if&else
findbugs	560	98	44	418	41	12	39	56	12
fitnesse	176	18	6	152	17	3	15	46	2
jackson-databind	93	16	9	68	6	4	14	12	1
jedit	112	19	3	90	9	5	9	9	3
junit4	85	9	0	76	10	1	8	18	1
lombok	88	21	4	63	6	2	2	14	1
mockito	81	17	0	64	13	2	2	12	1
tomcat	245	41	13	191	27	1	13	26	6

Table 6.5: Metrics on the analysed software systems

6.3.1 Discussion

Our approach proposed several innovative templates compared to the state of the art. Figure 6.5 shows an example proposed by the authors in [56] to refactor a for loop into a stream pipeline. The first fragment is the original code with the for loop. The loop cycle on an entrySet of a map, and after two conditional statements, add the value of the entry into an *ArrayList*. The second fragment shows the refactoring performed in [56]: they have replaced the for loop with a stream call on the entry set, then a filter is inserted to replace the first conditional statement, then a *forEach* is inserted within which the remaining statements were inserted. They are proposing a partial refactoring, since the statements from line 5 to line 7 are just copied within the *forEach* statement, which represents nothing more than a simple for loop. The only effective stream operation used by their approach is the filter method to replace the if condition *(isValid(enty))*. Conversely, the last fragment shows the refactoring proposed by our approach: we have inserted another filter operation to replace the second if statement. The stream APIs do not allow us to declare a variable, indeed we have removed the statement at line 5 since it's just a temp variable, indeed the *cl* variable can be replaced with its value, *entry.getKey()*. At this point, in order to replace the *result.add(entry.getValue()* operation in the for loop, we have inserted a *map()* operation to convert all the elements in the stream to the *entry.getValue()*

value, and through the *collect()* operations we are able to return a list that will be assigned to the *List< String > result* variable. The resulting stream pipeline fully embraces the functional style keeping the same behaviour of the imperative one. Table 6.5 highlights how the most common template found in the repositories is the one that creates a new collection using the collect operation, highlighting that many for loops are used to fill or generate collections.

Our proposed refactoring stands out for its conciseness and clarity, effectively minimising the code's line count when compared to alternative refactoring approaches. Additionally, it imparts a more coherent structure to the code, enhancing its intuitiveness and comprehensibility for developers. The amalgamation of functional and imperative styles, as exemplified in the second code fragment, introduces a potential risk of complicating the code structure, consequently diminishing its readability and maintainability.

However, it's important to acknowledge that a complete refactoring of a for loop into a stream pipeline may not always be feasible. In certain scenarios, a hybrid solution may be a viable choice. In such instances, the decision to retain the for loop or employ the hybrid approach lies with the developer. It is worth noting that the hybrid solution may offer opportunities for parallelisation using the *parallelStream()* API, which, when used appropriately, can lead to performance enhancements in the code.

The applicability of this approach is strictly related to the presence of for loops within the source code and on their structure. As stated in section 4.1, a for loop should pass all the preconditions and match one of the template to be refactored. This highly depends on the statements that comprise the cycle and its dependencies with external variables. The differences between functional and imperative style

fall heavily on the fact that many tasks are most naturally attacked by imperative means and cannot be represented as readily in a functional manner [44, 48].

Listing 6.5: This code is taken from the example 3 in figure 4 in [56], where they show their refactoring from for loop to stream. The last listing represents the refactoring proposed by our approach

```
// The original code of the example proposed by Lambdaficator
List<String> findReloadedContextMemoryLeaks() {
    List<String> result = new ArrayList<String >();
    for (Map.Entry<ClassLoader,String> entry : childClassLoaders.entrySet())
        if (isValid(entry)) {
            ClassLoader cl = entry.getKey();
            if (!((WebappClassLoader)cl).isStart())
                result.add(entry.getValue());
        }
}

// Their proposed refactoring
List<String> findReloadedContextMemoryLeaks () {
    List<String> result = new ArrayList<String >();
    childClassLoaders.entrySet().stream()
        .filter(entry -> isValid(entry))
        .forEach(entry -> {
            ClassLoader cl = entry.getKey();
            if(!((WebappClassLoader) cl).isStart())
                result.add(entry.getValue());
        });
}

// Our proposed refactoring
List<String> findReloadedContextMemoryLeaks () {
    List<String> result = childClassLoaders.entrySet().stream()
        .filter(entry -> isValid(entry))
        .filter(entry -> !((WebappClassLoader) entry.getkey()).isStart())
        .map(entry -> entry.getValue())
        .collect(Collectors.toList());
```

6.4 Matching Algorithms

We have compared our approach to a text-based search approach between methods and the templates of the algorithms. Table 6.6 shows the metrics obtained for the four methods previously described: column method displays the method considered for the analysis, respectively bubblesortV1 (listing 4.7), factorialV1 (listing 4.9), quicksortStack (listing 4.10) and quicksortArray (listing 4.11). The other five main columns are the templates used by our approach to match the algorithms: ItBubblesort is the bubblesort iterative version (the one displayed in Listing 4.8); RecFactorial is the factorial recursive version; ItQuicksortST is the quicksort iterative version using a stack to sort the elements; ItQuicksortAr is the quicksort iterative version using an array to sort the elements; ItMergesort is the mergesort iterative version. We have also considered the mergesort to show how the analysis is able to distinguish different algorithms. Each of these columns have two subcolumns: sim and text are respectively the similarity score of our approach and the similarity score of the text comparison approach. We have highlighted in bold text the highest score corresponding to the correct identification of the algorithm in subcolumn sim; whereas in subcolumn text the maximum score has been highlighted for the text match. We can see that the score assigned by our approach is much higher in each case, indicating a higher precision in the identification.

6.4.1 Discussion

For all the five methods shown in Table 6.6, our approach shows a higher identification score compared to the text similarity approach. Indeed, we have a higher similarity score that is more than double for the quickSortStack (0.84 compared to

Table 6.6: Similarities between the examples shown before and five different known algorithms used by our approach as templates. The first column, *method*, shows the name of method analysed, while the other columns displays for each template the similarity score given by our approach, column *sim*, and by a text comparison approach, column *text*.

	ItBubblesort		RecFactorial		ItQuicksortSt		ItQuicksortAr		ItMergesort	
method	sim	text	sim	text	sim	text	sim	text	sim	text
bubblesortV1	**0.85**	**0.52**	0.16	0.05	0.15	0.28	0.21	0.30	0.16	0.21
factorialV1	0.14	0.07	**1.0**	**0.7**	0.07	0.09	0.07	0.05	0.06	0.04
quicksortStack	0.25	0.21	0.08	0.06	**0.84**	**0.37**	0.42	0.29	0.23	0.26
quicksortArray	0.26	0.21	0.06	0.02	0.46	0.28	**0.86**	**0.34**	0.30	0.33

0.37) and quickSortArray (0.86 and 0.34) methods, and values about 40% greater for bubblesortV1 (0.85 and 0.52) and factorialV1 (1.0 and 0.7) methods. Our approach performs better because it can generalise the matching, without considering names of variables, comments and names of types.

Moreover, the matching scores given by our analysis are clearly greater than the score of other algorithms, whereas, with a text similarity approach, we can see that the quicksortArray method has a similarity score of 0.34 for ItQuickSortAr, 0.33 for ItMergesort and 0.28 for ItQuicksortSt, hence the closeness of such scores can bring ambiguity in the correct identification of the algorithm.

Finally, our approach can distinctly recognise two different versions of the same algorithm: the quicksortStack has 0.84 score as ItQuicksortSt and 0.42 as ItQuickSortAr, while the quicksortArray method has respectively 0.46 and 0.86. The accurate identification of the version used is crucial when suggesting improvements or proposing different versions.

The ability to recognise an algorithm is related to the set of templates, which is not easy to maintain and grow. An algorithm can be matched if a similar template of the same algorithm is already part of the templates. To handle this, the database can be populated with the most popular algorithms, and, if an algorithm is not included, it could be added by a developer using our approach and tool.

The approach extracts the statement's type to evaluate the similarity between algorithms. On the one hand, two algorithms can have similar statements despite having a different behaviour, thus showing low accuracy in identifying code's behaviour. On the other hand, the approach can give a degree of generalisation, since it is not dependent on names and types encountered, therefore it is not based on the comprehension of how the algorithm was implemented, but on its structure. This property is the main difference between our approach and type-3 clone approaches.

6.5 Unit Tests based on Code Behaviour

The proposed approach has been employed for analysing several Java software systems found on repositories. Since we exploit existing tests as templates for the generation of new tests, we have taken software systems from Maven Repository[1], which lets us quickly check the existence of tests.

Table 6.7 shows a summary of the metrics related to our test generation solution and produced for the software systems under analysis. For each analysed system, column *classes* shows the number of classes; column *tests* gives the number of existing tests; column *tested* gives the number of classes for which at least a test has been found in the repository; column *single* gives the number of tests found in the repository that execute a single class; column *gen* gives the number of newly generated tests, thanks to our approach; column *incr* gives the percentage increment of available tests accruing from our test generation; column *cover* gives the number of classes for which a test has been generated that were not previously tested, hence increasing code coverage. Moreover, columns *max* and *min* give the maximum and

[1] https://mvnrepository.com

minimum values of similarity found among a pair of classes on the project, respectively. Finally, column $t > 0.5$ gives the number of class pairs found to have a similarity greater than the threshold set as 0.5.

For the experiments, the minimum similarity threshold among classes has been set at 0.5, in order to select pairs of classes comparable enough to each other, and make the test generation effective. Among all software systems, similarity between pairs of classes was between 0.03 and 1. By setting the similarity threshold to 0.5, we have found a relatively low number of pairs (typically a bit less than 1%, 0, 9% for RepoDriller) compared to the total number of possible pairs. Still, for the analysed systems many classes have a comparable behaviour, i.e. between 2 and 495 pairs for the smallest and largest software system, respectively (see column $t > 0.5$ in Table 6.7).

For each pair of classes whose similarity is greater than the predetermined threshold, test generation was performed according to the four previously defined cases. The number of test cases generated ranged from 2 to 402 (see column *gen* in Table 6.7). The percentage increment for tests was between 13% and 72%. Moreover, column cover shows that between 6 and 63 additional classes were tested. In our approach, classes that have no test cases in the repository are selected as candidates for the following analysis finding class similarity and applicable test cases to be used as templates. Hence, when it is possible to generate tests, code coverage is also improved.

For test generation, the number of class pairs that matched our third method substitutions strategy (i.e. method name similarity) were greater than the other cases (up to 245 for the largest software system). However, sometimes it was not possible to employ the third method substitution strategy or the fourth (i.e. input

Table 6.7: Similarities between the examples shown before and five different known algorithms used by our approach as templates. The first column, *method*, shows the name of method analysed, while the other columns displays for each template the similarity score given by our approach, column *sim*, and by a text comparison approach, column *text*.

system	classes	test	tested	single	gen	incr	cover	max	min	t > 0.5
argparse4j	60	29	31	3	21	72%	7	1	0.05	23
jrn-unixsocket	17	15	9	0	2	13%	0	0.8	0.06	2
junit4	180	228	94	9	61	27%	19	1	0.03	106
mybatis3	279	572	129	9	402	70%	63	1	0.03	495
plexus-io	50	14	18	2	6	43%	6	1	0.06	20
repodriller	57	33	32	3	26	79%	6	1	0.05	29
vertx-mail-client	38	54	18	8	10	19%	0	1	0.05	11

parameter similarity) either. Several methods (up to 80) were substituted by means of the first and the second cases. For such matchings a number of new tests were generated.

6.5.1 Discussion

For the analysed software system RepoDriller, and for the classes shown in Figure 5.1, unit test *GenOnlyDiffsWithFileTypesTest* was generated by taking as a template the existing unit test *OnlyDiffsWithFileTypesTest*. Figure 6.6 shows the code of existing and generated unit tests. Following the above considerations on the similarity of classes and methods, generated test case *GenOnlyDiffsWithFileTypesTest* has been produced by substituting occurrences of class *OnlyDiffsWithFileTypes* into occurrences of class *OnlyDiffsWithoutFileTypes*. The two tests look equal since the classes used to generate the tests have the same structure (methods, signatures etc.). Beyond the test structure, the approach replicates the input values used in the test, which may not always be appropriate for the generated class.

The generated test cases provide a test for classes that were not previously tested, improving the code coverage of the test suite. Moreover, given the similarity between

the classes, frees the developer from having to write similar tests between different classes.

Listing 6.6: Test case *OnlyDiffsWithFileTypesTest* for class *OnlyDiffsWithFileTypes* and generated test case *GenOnlyDiffsWithFileTypesTest* for class *OnlyDiffsWithoutFileTypes*, for RepoDriller software system.

```
package org.repodriller.filter.diff;

import java.util.Arrays;
import org.junit.Assert;
import org.junit.Test;

public class OnlyDiffsWithFileTypesTest {

    @Test
    public void shouldAcceptIfFileHasExtensionWithDot() {
        Assert.assertTrue(new OnlyDiffsWithFileTypes( Arrays.asList("cpp",
            ".java")).accept("/dir/File.java"));
    }

    @Test
    public void shouldAcceptIfFileHasExtensionWithoutDot() {
        Assert.assertTrue(new OnlyDiffsWithFileTypes( Arrays.asList(".cpp",
            "java")).accept("/dir/File.java"));
    }

    @Test
    public void shouldRejectIfFileDoesNotMatchExtensions() {
        Assert.assertFalse(new OnlyDiffsWithFileTypes( Arrays.asList("cpp",
            ".java")).accept("/dir/File.css"));
    }
}

package org.repodriller.filter.diff;

import java.util.Arrays;
import org.junit.Assert;
import org.junit.Test;

public class GenOnlyDiffsWithFileTypesTest {

    @Test
    public void shouldAcceptIfFileHasExtensionWithDot() {
        Assert.assertTrue(new OnlyDiffsWithFileTypes( Arrays.asList("cpp",
            ".java")).accept("/dir/File.java"));
    }
```

The tests that can be generated for a software system depend both on the amount of classes that are found to have a similar behaviour and the existing tests that can be taken as a template. However, since the approach takes into account the APIs used by the class, it may be interesting to try the generation for classes belonging to different projects.

6.6 Test Template Generation

The tool developed according to the proposed approach has been employed for analysing several Java software systems. Firstly, we have extracted all test cases and collected asserts and types, as discussed in the previous analysis (Section 5.2.1).

The analysed repositories have been selected according to the most popular repositories on Maven Repository, and they are the following: *repodriller, commons-codec, threetenbp, slf4j, logback, and junit4*. Since we use existing tests for our analysis, we have taken software systems from Maven Repository, which lets us quickly check the existence of tests. Table 6.8 shows some relevant metrics of analysed software systems. Column *Systems* shows the name of the project; *Classes* shows the number of classes; *Methods* displays the total number of methods; *Tests* gives the number of test classes; *TestCases* gives the number of test cases; finally, column *Asserts* gives the number of asserts. The projects were selected according to the high number of tests and test cases, and the increasing number of tests for each.

Table 6.9 displays the metrics extracted from the six repository analysed: first column, *AssertType*, displays the most common assert type we have encountered, the following eight columns, *Byte*, *Boolean*, *Char*, *Double*, *Float*, *Integer*, *Long* and *String* represent, respectively, the number of asserts found for each class type; since

Table 6.8: Metrics of the analysed software systems used for gathering statistics.

system	classes	methods	tests	testCases	asserts
repodriller	59	299	34	100	378
commons-codec	90	768	81	817	2202
threetenbp	121	2126	96	2532	3516
slf4j	146	1240	106	179	373
logback	295	1637	509	1097	1720
junit4	243	1344	1017	1341	1235

Table 6.9: Metrics extracted from six systems: each row shows the occurrences of an assert, and each column displays the occurrences of a type. Each value is the number of times that a given assert has been used to test a given type.

AssertType	Byte	Boolean	Char	Double	Float	Integer	Long	String	Collect	Ref	SUMa
assertEquals	5	543	37	14	1	1635	364	1788	68	1893	6348
assertNotEquals	0	0	0	4	4	14	4	14	0	10	50
assertTrue	0	873	0	0	0	40	4	2	0	8	927
assertFalse	0	295	0	0	0	2	0	0	0	2	299
assertSame	0	0	0	1	1	0	0	2	1	43	48
assertNotSame	0	0	0	0	0	0	0	2	0	6	8
assertNull	0	0	0	0	0	0	0	97	3	82	182
assertNotNull	0	0	0	0	0	0	0	15	11	170	196
SUMt	5	1711	37	19	6	1691	372	1920	83	2214	8058

Java has classes to express primitive values, we have merged them, hence e.g. the column Integer will contains both the type int and the type java.lang.Integer; the next two columns, *Collect* and *Ref*, show, respectively, the merged interfaces *Collection* and *Map* with their all subclasses (e.g. *ArrayList*, *HashMap*, *HashSet* etc.), and all the other types of external libraries, other Java libraries, and classes of the system analysed; the column *SUMa* represents the sum of the occurrences for the specified assert, whereas row *SUMt* displays the total count of occurrences for each type.

The most used assert is the *assertEquals*, representing the 78% of the total occurrences of assert identified by our analysis; conversely, *assertNotSame* is the lowest one with less than 0,1%. This outlines how primitive types are often tested by a comparison of values, except for the *boolean* type, indeed the *assertEquals* represents

Table 6.10: Metrics extracted for asserts that present a chain to test a specific type. On the left the assert type followed by the method called on it, while on the right the number of occurrences found.

Assert	Chain	Count
assertEquals	java.lang.String.contains	23
assertTrue	java.lang.String.contains	66
assertTrue	java.lang.String.startsWith	24
assertTrue	java.lang.String.matches	23
assertTrue	java.util.List.contains	138
assertEquals	java.util.List.get	62
assertEquals	java.util.List.size	195

the 31% of the total occurrences found, whereas the assertTrue and *assertFalse* are respectively the 51% and 18%.

Table 6.10 shows the most significant chains for types *String* and *Collection* that are identified by our analysis. Despite the *String* type is widely directly tested with 1920 occurrences (Listing 5.2), it is also tested with different calls, such as *contains*, *startsWith* and *matches*; in particular, the contains method is tested both with *assertTrue* and *assertEquals*. For the *Collection* type, we found that the most used collection is List; the most two frequent calls are *contains* and *size*, however the list is also directly accessed with the *get* call.

6.6.1 Discussion

Our approach automatically generates the template based on the said analysis and chooses assert types according to the metrics extracted. E.g., a *String* is mostly tested with the *assertEquals*, however if the developer wants to check a specific character or substring of it, he can follow our tool's suggestion to use *contains* or *startsWith*, depending on the purpose of the test. We used our tool for analysing and generating tests for the repodriller toolkit. Table 6.11 shows the metrics of

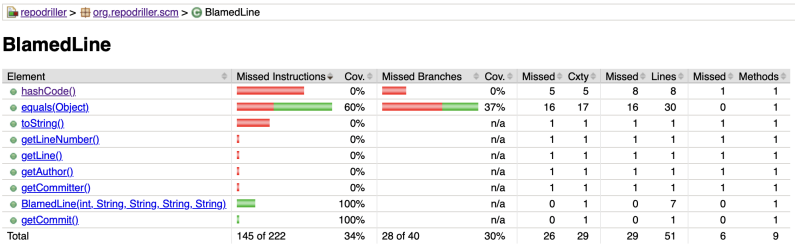
Table 6.11: The metrics obtained by the analysis of the *repodriller* repository.

system	classes	tests	tc	totM	testM	!testM	genC	genT
repodriller	56	35	100	371	263	108	17	43

repodriller code: column *Classes* represents the number of classes; *Tests* gives the number of existing tests; *Tc* shows the number of test cases; *TotM* gives the number of methods; *TestM* displays the number of methods covered by a test according to Jacoco; !*TestM* shows the number of method not covered; *GenC* gives the number of tests generated; and, *GenT* gives the number of template test cases generated. Almost 30% of methods are not covered, therefore it is worth expanding the test suite. We successfully generated 43 templates of test cases, and the additional test cases cover almost 40% of methods not tested, whereas the remaining 60% are methods returning a type that was not suitable, i.e. a return type not in the standard Java APIs or without statistics. We increased the number of test cases by 43%, giving a greater robustness and coverage to the test suite. We have generated 17 new test cases, and the developer can decide whether to merge these tests with existing ones or include them inside the test suite.

Figure 6.2 shows an example of a class not entirely covered by the suite; it is the output of JaCoCo for the BlamedLine class in repodriller. The class presents six methods that are not covered by any test, i.e. hashCode(), toString(), getLineNumber(), getLine(), getAuthor() and getCommitter(). Our approach has correctly identified such methods and has generated a test case for each of them. Listing 6.7 shows an example of a template generated for the method hashCode(), which returns a String. Our tool has selected assertEquals to properly test it, i.e. the most frequent assert used for that type. We have manually inserted the expected value based on the other test cases of the application. The test case was correctly

compiled and executed.


repodriller > org.repodriller.scm > BlamedLine

BlamedLine

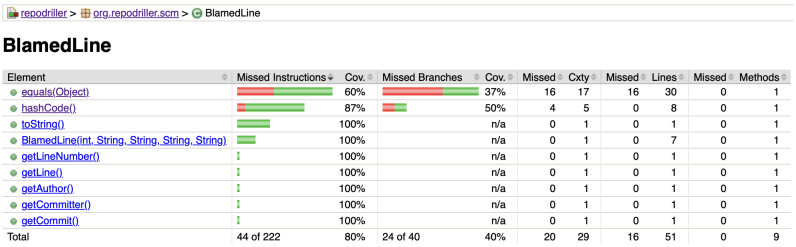
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
hashCode()		0%		0%	5	5	8	8	1	1
equals(Object)		60%		37%	16	17	16	30	0	1
toString()		0%		n/a	1	1	1	1	1	1
getLineNumber()		0%		n/a	1	1	1	1	1	1
getLine()		0%		n/a	1	1	1	1	1	1
getAuthor()		0%		n/a	1	1	1	1	1	1
getCommitter()		0%		n/a	1	1	1	1	1	1
BlamedLine(int, String, String, String, String)		100%		n/a	0	1	0	7	0	1
getCommit()		100%		n/a	0	1	0	1	0	1
Total	145 of 222	34%	28 of 40	30%	26	29	29	51	6	9

Figure 6.2: Jacoco's code coverage output **before** our templates generation of the class BlamedLine from repodriller repository.

Listing 6.7: An example of a generated template for the class *BlamedLine* and the method *hashCode()*.

```
@Test
public void hashCodeTest() {
    Assert.assertEquals(
            new BlamedLine(5, "  }", "Mauricio Aniche", "Mauricio Aniche",
            "a4ece0762e797d2e2dcbd471115108dd6e05ff58").hashCode(),
            blame.get(5).hashCode());
}
```

Figure 6.3 displays the output of JaCoCo after the insertion of our templates inside the test suite: the coverage of the instructions has reached 100% for almost all not previously tested methods, except for the hashCode() method because it contains four different ternary operators that are not executed for both conditions. The total coverage of the instructions has increased by 46% thanks to our test generation. Summing up, the number of tests generated depends on the number of methods that have not been previously tested, indeed their return type must be suitable for our analysis to ensure the generation. Furthermore, according to the metrics shown in Table 6.9, we generated a template test suite even for applications that have not been previously tested.


repodriller > org.repodriller.scm > BlamedLine

BlamedLine

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
equals(Object)		60%		37%	16	17	16	30	0	1
hashCode()		87%		50%	4	5	0	8	0	1
toString()		100%		n/a	0	1	0	1	0	1
BlamedLine(int, String, String, String, String)		100%		n/a	0	1	0	7	0	1
getLineNumber()		100%		n/a	0	1	0	1	0	1
getLine()		100%		n/a	0	1	0	1	0	1
getAuthor()		100%		n/a	0	1	0	1	0	1
getCommitter()		100%		n/a	0	1	0	1	0	1
getCommit()		100%		n/a	0	1	0	1	0	1
Total	44 of 222	80%	24 of 40	40%	20	29	16	51	0	9

Figure 6.3: Jacoco's code coverage output **after** our templates generation of the class BlamedLine from repodriller repository.

One of the main criteria to evaluate the effectiveness of a test suite is code coverage. The larger the portions of code covered, the greater the possibility of finding defects. Of course, choosing the proper input values can make the test case very effective. However, writing tests for all the classes and methods is time-consuming, and developers tend to focus on testing just the most used and complex functionalities. This study has proposed an innovative approach to generate test templates for methods that have not been previously covered by any test case. This frees developers from the effort of writing test code, while at the same time supporting them as the clean generated code can be further customised. The approach firstly gathers knowledge from the code of test cases in other software systems, to learn how to properly generate new ones. The execution time to gather statistics is indeed limited as it depends on the amount of code analysed, executing code of applications is not needed. The collected statistics are mainly used to generate proper templates for other systems having parts that have not been tested. It has been found that the test generation is effective and accurate, both because there is a significant number of test templates generated, and because the code coverage has been increased by generated tests. The greater the number of tests calling different methods, the greater the coverage and the quality of the test suite.

Chapter 7

Conclusion

The relentless progression of both hardware, as encapsulated by Moore's Law, and software, governed by Lehman's Laws, necessitates an ongoing commitment from developers to maintain and update their applications. This undertaking involves substantial management costs, extended working hours, and often entails complex tasks. The integration of automated refactoring and testing tools emerges as a pivotal solution, significantly alleviating the burden associated with maintenance and aiding developers in navigating arduous and intricate activities.

This dissertation introduces a set of innovative approaches designed to furnish a fully automated suite of tools for refactoring and testing software applications. The refactoring unit encompasses tools for seamlessly transforming sequential code into parallel structures, refactoring for loops into Java streams, and identifying as well as refactoring algorithms. Simultaneously, the testing unit provides two tools for generating tests for classes and methods that lack prior testing. These approaches are collected under a unified comprehensive framework, denoted as *ReFrame*. The modularity of the framework enables the independent or collaborative utilisation of each feature.

The framework employs static analysis to compile all necessary information for

subsequent analyses, thereby necessitating only the source code of the application to execute all optimisations. The generated code aligns consistently and semantically with the existing code in the application, with developers being informed about alterations made to their source code to facilitate an understanding of the decision-making process behind the transformations.

Each module of the framework undergoes rigorous testing on real-world scenarios, affirming their correctness and validity. The outcomes from the refactoring modules underscore the tools' adeptness in accurately identifying, executing, and producing refactored code while preserving the application's behaviour and effecting notable improvements in performance, readability, and quality. The testing tools exhibit substantial increases in test coverage for analysed applications, encompassing a greater number of methods and classes. This mitigates the likelihood of encountering bugs during release phases and ensures that refactored code aligns with the original intent of the application.

The adoption of a unified tool for both refactoring and testing substantially diminishes the time and resources required to configure and operate distinct tools. Notably, the hindrance often lies in the time spent installing, configuring, and mastering a refactoring or testing tool, surpassing the actual benefits it brings to the application. This realisation inspired the conception of a singular framework that integrates various types of refactoring alongside relevant test generation. The tool's execution time is limited, contingent on the selected modules for refactoring and testing, and the volume of analysed code; actual execution of application code is unnecessary. The tool's user-friendly design mandates solely the application's source code, allowing developers to selectively engage with modules of interest.

7.1 Future Works

On November 30th, the official release of ChatGPT marked a significant milestone, showcasing the vast capabilities of Artificial Intelligence (AI) in diverse domains, including text recognition and production, image generation, audio synthesis, and notably, programming. ChatGPT falls within the broader category of Large Language Models (LLMs), distinguished by its proficiency in general-purpose language comprehension and generation. A language model, within this context, functions as a probabilistic representation of a natural language, generating probabilities for sequences of words based on the textual corpora on which it was trained, encompassing one or multiple languages.

Over the past year, several LLMs have been introduced, accessible either for free or through subscription models. Prominent among these models are GPT-4(https://openai.com/gpt-4), Gemini(https://gemini.google.com/app), LLama(https://ai.meta.com/llama/), and Bloom(https://bigscience.huggingface.co/blog/bloom).

Current research in the field is focused on investigating the functionality of these LLMs with programming languages, particularly in the realms of code generation, code summarisation, and code search. A promising avenue for future exploration involves examining the potential of these tools in automatic software refactoring and testing. For instance, one potential application could involve employing an LLM for concurrent refactoring, assessing its performance, and utilising the transformer from sequential to parallel, as proposed in Chapter 3. This approach could contribute to the creation of a labelled dataset wherein pairs of values are represented by sequential and parallel code. Fine-tuning the LLM with this dataset aims to evaluate its ability to autonomously propose accurate and effective concurrent refactorings.

Bibliography

[1] Syed Ahmed and Mehdi Bagherzadeh. "What Do Concurrency Developers Ask about? A Large-Scale Study Using Stack Overflow". In: *12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. Oulu, Finland, 2018. ISBN: 9781450358231. DOI: 10.1145/3239235.3239524.

[2] Qurat Ul Ain et al. "A Systematic Review on Code Clone Detection". In: *IEEE Access* 7 (2019), pp. 86121–86144. DOI: 10.1109/ACCESS.2019.2918202.

[3] Frances E Allen. "Control flow analysis". In: *ACM Sigplan Notices* 5.7 (1970), pp. 1–19.

[4] M. Moein Almasi et al. "An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application". In: *Proocedings of the IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 2017, pp. 263–272. DOI: 10.1109/ICSE-SEIP.2017.27.

[5] Sanjay B. Ankali and Latha Parthiban. "Development of Porting Analyzer to Search Cross-Language Code Clones Using Levenshtein Distance". In:

Proocedings of Fourth International Conference on Smart Computing and Informatics (SCI). Springer, 2021, pp. 623–632. ISBN: 978-981-16-0878-0. DOI: 10.1007/978-981-16-0878-0_60.

[6] Ellen Arteca, Frank Tip, and Max Schäfer. "Enabling Additional Parallelism in Asynchronous JavaScript Applications". In: *35th European Conference on Object-Oriented Programming (ECOOP)*. 2021. DOI: 10.4230/LIPIcs.ECOOP.2021.7.

[7] Soe Thandar Aung et al. "A proposal of grammar-concept understanding problem in Java programming learning assistant system". In: *J. Adv. Inform. Tech.(JAIT)* 12.4 (2021), pp. 342–350. DOI: 10.12720/jait.12.4.342-350.

[8] Aditi Barua and Yoonsik Cheon. "A catalog of while loop specification patterns". In: (2014).

[9] Gabriele Bavota et al. "When Does a Refactoring Induce Bugs? An Empirical Study". In: *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. 2012, pp. 104–113. DOI: 10.1109/SCAM.2012.20.

[10] K. Beck. *Extreme Programming Explained: Embrace Change*. An Alan R. Apt Book Series. Addison-Wesley, 2000. ISBN: 9780201616415.

[11] A. J. Bernstein. "Analysis of Programs for Parallel Processing". In: *IEEE Transactions on Electronic Computers* EC-15.5 (1966), pp. 757–763. ISSN: 0367-7508. DOI: 10.1109/PGEC.1966.264565.

[12] Aggelos Biboudis et al. "Streams à la carte: Extensible pipelines with object algebras". In: *29th European Conference on Object-Oriented Programming*

(ECOOP 2015). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2015. DOI: 10.4230/DARTS.1.1.9.

[13] Brent van Bladel and Serge Demeyer. "Test Behaviour Detection as a Test Refactoring Safety". In: *Proceedings of the 2nd International Workshop on Refactoring (IWoR*. Montpellier, France, 2018, pp. 22–25. DOI: 10.1145/3242163.3242168.

[14] Joshua Bonn, Konrad Foegen, and Horst Lichter. "A Framework for Automated Combinatorial Test Generation, Execution, and Fault Characterization". In: *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2019, pp. 224–233. DOI: 10.1109/ICSTW.2019.00057.

[15] Pietro Braione et al. "Combining Symbolic Execution and Search-Based Testing for Programs with Complex Heap Inputs". In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. Santa Barbara, CA, USA, 2017, pp. 90–101. DOI: 10.1145/3092703.3092715.

[16] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. "ASM: a code manipulation tool to implement adaptable systems". In: *Adaptable and extensible component systems* 30.19 (2002).

[17] Christian Bunse and Sebastian Stiemer. "On the energy consumption of design patterns". In: (2013). DOI: 10.1007/s40568-013-0020-6.

[18] Christian Bunse et al. "Choosing the "Best" Sorting Algorithm for Optimal Energy Consumption". In: *International Conference on Software and Data*

Technologies. 2009. URL: https://api.semanticscholar.org/CorpusID:9874889.

[19] Christian Bunse et al. "Exploring the Energy Consumption of Data Sorting Algorithms in Embedded and Mobile Environments". In: *2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware*. 2009, pp. 600–607. DOI: 10.1109/MDM.2009.103.

[20] Salvatore Calanna et al. "A (Reverse) Mutation Testing Approach to Automatically generate parallel C/C++ Code". In: *Proceedings of IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. 2021, pp. 159–164. DOI: 10.1109/WETICE53228.2021.00040.

[21] Salvatore Calanna et al. "Automatic Generation of Parallel Java Programs and their Validation using Combinatorial Testing Suites". In: *2021 IEEE 6th International Conference on Computer and Communication Systems (ICCCS)*. 2021, pp. 1142–1146. DOI: 10.1109/ICCCS52626.2021.9449249.

[22] Andrea Calvagna, Andrea Fornaia, and Emiliano Tramontana. "Random versus Combinatorial Effectiveness in Software Conformance Testing: A Case Study". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC)*. Salamanca, Spain, 2015, pp. 1797–1802. ISBN: 9781450331968. DOI: 10.1145/2695664.2695905.

[23] Andrea Calvagna and Emiliano Tramontana. "Automated Conformance Testing of Java Virtual Machines". In: *Proocedings of the Seventh International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS)*. 2013, pp. 547–552. DOI: 10.1109/CISIS.2013.99.

[24] Andrea Calvagna and Emiliano Tramontana. "Delivering Dependable Reusable Components by Expressing and Enforcing Design Decisions". In: *2013 IEEE 37th International Computer Software and Applications Conference Workshops (COMPSACW)*. Los Alamitos, CA, USA: IEEE Computer Society, 2013, pp. 493–498. DOI: 10.1109/COMPSACW.2013.113.

[25] Yu Chan et al. "A distributed stream library for Java 8". In: *IEEE Transactions on Big Data* 3.3 (2017), pp. 262–275. DOI: 10.1109/TBDATA.2017.2666201.

[26] Tej Bahadur Chandra, VK Patle, and Sanjay Kumar. "New horizon of energy efficiency in sorting algorithms: green computing". In: *Proceedings of National Conference on Recent Trends in Green Computing. School of Studies in Computer in Computer Science & IT, Pt. Ravishankar Shukla University, Raipur, India*. 2013, pp. 24–26.

[27] Carlos A. Coello Coello, Gary B. Lamont, and David A. Van Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems (Genetic and Evolutionary Computation)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387332545.

[28] Diego Costa et al. "Empirical Study of Usage and Performance of Java Collections". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE)*. L'Aquila, Italy, 2017, pp. 389–400. DOI: 10.1145/3030207.3030221.

[29] Will Crichton, Georgia Gabriela Sampaio, and Pat Hanrahan. "Automating Program Structure Classification". In: *Proceedings of 52nd ACM Technical*

Symposium on Computer Science Education (SIGCSE). Virtual Event, USA, 2021, pp. 1177–1183. ISBN: 9781450380621. DOI: 10.1145/3408877.3432358.

[30] Brett Daniel et al. "Automated Testing of Refactoring Engines". In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE)*. Dubrovnik, Croatia, 2007, pp. 185–194. ISBN: 9781595938114. DOI: 10.1145/1287624.1287651.

[31] *Amazon Book Reviews dataset*. https://www.kaggle.com/datasets/mohamedbakhet/amazon-books-reviews. Accessed: 2023-07-21.

[32] Danny Dig. "Refactoring for Asynchronous Execution on Mobile Devices". In: *IEEE Software* 32.6 (2015), pp. 52–61. DOI: 10.1109/MS.2015.133.

[33] Danny Dig, John Marrero, and Michael D. Ernst. "How Do Programs Become More Concurrent: A Story of Program Transformations". In: *Proceedings of the 4th International Workshop on Multicore Software Engineering (IWMSE)*. Waikiki, Honolulu, HI, USA, 2011, pp. 43–50. ISBN: 9781450305778. DOI: 10.1145/1984693.1984700.

[34] Danny Dig, John Marrero, and Michael D. Ernst. "Refactoring Sequential Java Code for Concurrency via Concurrent Libraries". In: *31st IEEE International Conference on Software Engineering (ICSE)*. USA, 2009, pp. 397–407. ISBN: 9781424434534. DOI: 10.1109/ICSE.2009.5070539.

[35] Danny Dig et al. "Relooper: Refactoring for Loop Parallelism in Java". In: *24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. Orlando, Florida,

USA, 2009, pp. 793–794. ISBN: 9781605587684. DOI: 10.1145/1639950.1640018.

[36] Marko Dimjaševic and Zvonimir Rakamaric. "JPF-Doop: Combining concolic and random testing for Java". In: *Collections (org. apache. commons. collections)* 422.3894 (2013), p. 58470.

[37] Pascal A. Felber. "Semi-automatic Parallelization of Java Applications". In: *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*. Springer Berlin Heidelberg, 2003, pp. 1369–1383. ISBN: 978-3-540-39964-3. DOI: 0.1007/978-3-540-39964-3_86.

[38] Andrea Fornaia, Stefano Scafiti, and Emiliano Tramontana. "JSCAN: Designing an Easy to use LLVM-Based Static Analysis Framework". In: *2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. 2019, pp. 237–242. DOI: 10.1109/WETICE.2019.00058.

[39] Andrea Fornaia and Emiliano Tramontana. "DeDuCT: A Data Dependence Based Concern Tagger for Modularity Analysis". In: *IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. 2017, pp. 463–468. DOI: 10.1109/COMPSAC.2017.98.

[40] Andrea Fornaia and Emiliano Tramontana. "Is My Code Easy to Port? Using Taint Analysis to Evaluate and Assist Code Portability". In: *IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. 2017, pp. 269–274. DOI: 10.1109/WETICE.2017.51.

[41] Andrea Fornaia et al. "Automatic Generation of Effective Unit Tests based on Code Behaviour". In: *2020 IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. 2020, pp. 213–218. DOI: 10.1109/WETICE49692.2020.00049.

[42] Martin Fowler. *Refactoring*. Addison-Wesley Professional, 2018.

[43] Geoffrey C Fox, Roy D Williams, and Paul C Messina. *Parallel computing works!* Elsevier, 2014.

[44] Scott Frame and John W Coffey. "A comparison of functional and imperative programming techniques for mathematical software development". In: *Journal of Systemics, Cybernetics and Informatics* 12.2 (2014), pp. 49–53. ISSN: 1690-4524.

[45] Lyle Franklin et al. "LAMBDAFICATOR: from imperative to functional programming through automated refactoring". In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 1287–1290. DOI: 10.1109/ICSE.2013.6606699.

[46] Gordon Fraser and Andrea Arcuri. "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software". In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*. Szeged, Hungary, 2011, pp. 416–419. ISBN: 9781450304436. DOI: 10.1145/2025113.2025179.

[47] Davide Fucci et al. "A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last?" In: *IEEE Transactions on Software Engineering* 43.7 (2017), pp. 597–614. DOI: 10.1109/TSE.2016.2616877.

[48] David K. Gifford and John M. Lucassen. "Integrating Functional and Imperative Programming". In: *Proceedings of the 1986 ACM Conference on LISP and Functional Programming (LFP)*. Cambridge, Massachusetts, USA, 1986, pp. 28–38. ISBN: 0897912004. DOI: 10.1145/319838.319848.

[49] Satyajit Gokhale, Alexi Turcotte, and Frank Tip. "Automatic Migration from Synchronous to Asynchronous JavaScript APIs". In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021). DOI: 10.1145/3485537.

[50] Arnaud Gotlieb and Dusica Marijan. "FLOWER: Optimal Test Suite Reduction as a Network Maximum Flow". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*. San Jose, CA, USA, 2014, pp. 171–180. ISBN: 9781450326452. DOI: 10.1145/2610384.2610416.

[51] Giovanni Grano et al. "An Empirical Investigation on the Readability of Manual and Generated Test Cases". In: *Proceedings of Conference on Program Comprehension (ICPC)*. Gothenburg, Sweden, 2018, pp. 348–351. DOI: 10.1145/3196321.3196363.

[52] Kate Gregory and Ade Miller. *C++ AMP: accelerated massive parallelism with Microsoft Visual C++*. Microsoft Press, 2012. ISBN: 0735664730.

[53] William G. Griswold and William F. Opdyke. "The Birth of Refactoring: A Retrospective on the Nature of High-Impact Software Engineering Research". In: *IEEE Software* 32.6 (2015), pp. 30–38. DOI: 10.1109/MS.2015.107.

[54] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. "SPEED: Precise and Efficient Static Estimation of Program Computational Complexity". In: *SIGPLAN Not.* 44.1 (2009), pp. 127–139. ISSN: 0362-1340. DOI: 10.1145/1594834.1480898.

[55] Jan Gustafsson et al. "The Mälardalen WCET benchmarks: Past, present and future". In: *10th International Workshop on Worst-Case Execution Time Analysis (WCET)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2010. DOI: 10.4230/OASIcs.WCET.2010.136.

[56] Alex Gyori et al. "Crossing the Gap from Imperative to Functional Programming through Refactoring". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. Saint Petersburg, Russia, 2013, pp. 543–553. ISBN: 9781450322379. DOI: 10.1145/2491411.2491461.

[57] Michael Haidl and Sergei Gorlatch. "PACXX: Towards a Unified Programming Model for Programming Accelerators Using C++14". In: *LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 2014, pp. 1–11. DOI: 10.1109/LLVM-HPC.2014.9.

[58] Shuai Hao et al. "Estimating mobile application energy consumption using program analysis". In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 92–101. DOI: 10.1109/ICSE.2013.6606555.

[59] Samir Hasan et al. "Energy Profiles of Java Collections Classes". In: *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. Austin, Texas, 2016, pp. 225–236. ISBN: 9781450339001. DOI: 10.1145/2884781.2884869.

[60] Yoshiki Higo et al. "kGenProg: A High-Performance, High-Extensibility and High-Portability APR System". In: *Proocedings of the 25th Asia-Pacific Software Engineering Conference (APSEC)*. 2018, pp. 697–698. DOI: 10.1109/APSEC.2018.00094.

[61] Michael Hilton et al. "Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Singapore, Singapore, 2016, pp. 426–437. ISBN: 9781450338455. DOI: 10.1145/2970276.2970358.

[62] Abram Hindle. "Green Mining: A Methodology of Relating Software Change and Configuration to Power Consumption". In: *Empirical Softw. Engg.* 20.2 (2015), pp. 374–409. ISSN: 1382-3256. DOI: 10.1007/s10664-013-9276-6.

[63] Takashi Ishio et al. "Cloned Buggy Code Detection in Practice Using Normalized Compression Distance". In: *Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018, pp. 591–594. DOI: 10.1109/ICSME.2018.00022.

[64] Kazuaki Ishizaki, Shahrokh Daijavad, and Toshio Nakatani. "Refactoring Java Programs Using Concurrent Libraries". In: *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*. Toronto, Ontario, Canada, 2011, pp. 35–44. ISBN: 9781450308090. DOI: 10.1145/2002962.2002970.

[65] Nishtha Jatana et al. "Test Suite Reduction by Mutation Testing Mapped to Set Cover Problem". In: *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies (ICTCS)*. Udaipur, India, 2016. ISBN: 9781450339629. DOI: 10.1145/2905055.2905094.

[66] Hojun Jaygarl, Kai-Shin Lu, and Carl K. Chang. "GenRed: A Tool for Generating and Reducing Object-Oriented Test Cases". In: *Proocedings of the IEEE*

34th Annual Computer Software and Applications Conference (COMPSAC). 2010, pp. 127–136. DOI: 10.1109/COMPSAC.2010.19.

[67] Alan Kaminsky. "Parallel Java Library". In: *International Conference for high performance computing, networking, storage and analysis*. 2014.

[68] Hironori Kasahara et al. "Multicore Cache Coherence Control by a Parallelizing Compiler". In: *41st IEEE Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. 2017, pp. 492–497. DOI: 10.1109/COMPSAC.2017.174.

[69] Raffi Khatchadourian, Yiming Tang, and Mehdi Bagherzadeh. "Safe automated refactoring for intelligent parallelization of Java 8 streams". In: *Science of Computer Programming* 195 (2020), p. 102476. ISSN: 0167-6423. DOI: 10.1016/j.scico.2020.102476.

[70] Keiji Kimura, Gakuho Taguchi, and Hironori Kasahara. "Accelerating Multicore Architecture Simulation Using Application Profile". In: *10th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*. 2016, pp. 177–184. DOI: 10.1109/MCSoC.2016.16.

[71] Per Larsen et al. "Parallelizing more Loops with Compiler Guided Refactoring". In: *2012 41st International Conference on Parallel Processing (ICPP)*. 2012, pp. 410–419. DOI: 10.1109/ICPP.2012.48.

[72] Owolabi Legunsen et al. "An Extensive Study of Static Regression Test Selection in Modern Software Evolution". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. Seattle, WA, USA, 2016, pp. 583–594. ISBN: 9781450342186. DOI: 10.1145/2950290.2950361.

[73] Ding Li et al. "Calculating Source Line Level Energy Information for Android Applications". In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*. Lugano, Switzerland, 2013, pp. 78–89. ISBN: 9781450321594. DOI: 10.1145/2483760.2483780.

[74] Yu Lin, Cosmin Radoi, and Danny Dig. "Retrofitting concurrency for android applications through refactoring". In: *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. Hong Kong, China, 2014, pp. 341–352. DOI: 10.1145/2635868.2635903.

[75] Andreas Litke et al. "Energy consumption analysis of design patterns". In: *Proceedings of the International Conference on Machine Learning and Software Engineering*. 2005, pp. 86–90. DOI: 10.5281/zenodo.1057717.

[76] Kenan Liu, Gustavo Pinto, and Yu David Liu. "Data-Oriented Characterization of Application-Level Energy Optimization". In: *Fundamental Approaches to Software Engineering*. Ed. by Alexander Egyed and Ina Schaefer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 316–331. ISBN: 978-3-662-46675-9. DOI: 10.1007/978-3-662-46675-9_21.

[77] Xiangjun Liu and Ping Yu. "Randoop-TSR: Random-Based Test Generator with Test Suite Reduction". In: *Proceedings of the 13th Asia-Pacific Symposium on Internetware*. Hohhot, China, 2022, pp. 221–230. DOI: 10.1145/3545258.3545280.

[78] P. Lokuciejewski et al. "A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models". In: *International Symposium on Code Generation and Optimization (CGO)*. 2009, pp. 136–146. DOI: 10.1109/CGO.2009.17.

[79] Paul Lokuciejewski and Peter Marwedel. "Combining Worst-Case Timing Models, Loop Unrolling, and Static Loop Analysis for WCET Minimization". In: *21st Euromicro Conference on Real-Time Systems (ECRTS)*. 2009, pp. 35–44. DOI: 10.1109/ECRTS.2009.9.

[80] Walid Maalej et al. "On the Comprehension of Program Comprehension". In: *ACM Transactions on Software Engineering Methodology* 23.4 (2014). ISSN: 1049-331X. DOI: 10.1145/2622669.

[81] Birk Martin Magnussen et al. "Performance Evaluation of OSCAR Multi-Target Automatic Parallelizing Compiler on Intel, AMD, Arm and RISC-V Multicores". In: *Languages and Compilers for Parallel Computing: 34th International Workshop, LCPC 2021, Newark, DE, USA, October 13–14, 2021, Revised Selected Papers*. Newark, DE, USA: Springer-Verlag, 2021, pp. 50–64. ISBN: 978-3-030-99371-9. DOI: 10.1007/978-3-030-99372-6_4.

[82] Irene Manotas, Lori Pollock, and James Clause. "SEEDS: A Software Engineer's Energy-Optimization Decision Support Framework". In: *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. Hyderabad, India, 2014, pp. 503–514. ISBN: 9781450327565. DOI: 10.1145/2568225.2568297.

[83] Shane A Markstrum, Robert M Fuhrer, and Todd D Millstein. "Towards concurrency refactoring for x10". In: *ACM Sigplan Notices* 44.4 (2009), pp. 303–304. DOI: 10.1145/1594835.1504226.

[84] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. "Efficient and Exact Data Dependence Analysis". In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI)*.

Toronto, Ontario, Canada, 1991, pp. 1–14. ISBN: 0897914287. DOI: 10.1145/113445.113447.

[85] T. Mens and T. Tourwe. "A survey of software refactoring". In: *IEEE Transactions on Software Engineering* 30.2 (2004), pp. 126–139. DOI: 10.1109/TSE.2004.1265817.

[86] Alessandro Midolo and Emiliano Tramontana. "A Robust and Automatic Approach for Matching Algorithms". In: *15th Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE)*. Vol. 3483. 2023, pp. 57–65.

[87] Alessandro Midolo and Emiliano Tramontana. "An API for and Classifying Data Dependence in View of Parallelism". In: *Proceedings of the 10th International Conference on Computer and Communications Management (ICCCM)*. Okayama, Japan, 2022, pp. 61–67. ISBN: 9781450396349. DOI: 10.1145/3556223.3556232.

[88] Alessandro Midolo and Emiliano Tramontana. "An Automatic Transformer from Sequential to Parallel Java Code". In: *Future Internet* 15.9 (2023). ISSN: 1999-5903. DOI: 10.3390/fi15090306. URL: https://www.mdpi.com/1999-5903/15/9/306.

[89] Alessandro Midolo and Emiliano Tramontana. "Automatic Generation of Accurate Test Templates based on JUnit Asserts". In: *Proceedings of the 7th International Conference on Algorithms, Computing and Systems (ICACS)*. Larissa, Greece, 2023, pp. 125–131. ISBN: 9798400709098. DOI: 10.1145/3631908.3631926.

[90] Alessandro Midolo and Emiliano Tramontana. "Refactoring Java Loops to Streams Automatically". In: *Proceedings of the 4th International Conference*

on Computer Science and Software Engineering (CSSE). Singapore, Singapore, 2021, pp. 135–139. ISBN: 9781450390675. DOI: 10.1145/3494885.3494910.

[91] Audris Mockus, Nachiappan Nagappan, and Trung T. Dinh-Trong. "Test coverage and post-verification defects: A multiple case study". In: *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. 2009, pp. 291–301. DOI: 10.1109/ESEM.2009.5315981.

[92] Melina Mongiovi et al. "Making refactoring safer through impact analysis". In: *Science of Computer Programming* 93 (2014). Special Issue with Selected Papers from the Brazilian Symposium on Programming Languages (SBLP 2011), pp. 39–64. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2013.11.001.

[93] Misael Mongiovì, Andrea Fornaia, and Emiliano Tramontana. "A Network-Based Approach for Reducing Test Suites While Maintaining Code Coverage". In: *Complex Networks and Their Applications VIII*. Cham: Springer International Publishing, 2020, pp. 164–176. ISBN: 978-3-030-36683-4. DOI: 10.1007/978-3-030-36683-4_14.

[94] Misael Mongiovì, Andrea Fornaia, and Emiliano Tramontana. "REDUNET: reducing test suites by integrating set cover and network-based optimization". In: *Applied Network Science* 5.1 (2020), pp. 1–21. DOI: 10.1007/s41109-020-00323-w.

[95] Christian Murphy, Zoher Zoomkawalla, and Koichiro Narita. "Automatic test case generation and test suite reduction for closed-loop controller software". In: (2013).

[96] William F Opdyke. *Refactoring object-oriented frameworks*. University of Illinois at Urbana-Champaign, 1992.

[97] Ali Ouni et al. "Multi-Criteria Code Refactoring Using Search-Based Software Engineering: An Industrial Case Study". In: *ACM Trans. Softw. Eng. Methodol.* 25.3 (2016). ISSN: 1049-331X. DOI: 10.1145/2932631.

[98] Burcu Kulahcioglu Ozkan, Michael Emmi, and Serdar Tasiran. "Systematic asynchrony bug exploration for android apps". In: *International Conference on Computer Aided Verification*. Springer International Publishing", 2015, pp. 455–461. DOI: 10.1007/978-3-319-21690-4_28.

[99] Carlos Pacheco and Michael D. Ernst. "Randoop: Feedback-Directed Random Testing for Java". In: *Proocedings of Object-Oriented Programming Systems and Applications companion (OOPSLA)*. Montreal, Quebec, Canada, 2007, pp. 815–816. DOI: 10.1145/1297846.1297902.

[100] Candy Pang et al. "What Do Programmers Know about Software Energy Consumption?" In: *IEEE Software* 33.3 (2016), pp. 83–89. DOI: 10.1109/MS.2015.83.

[101] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. "Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets". In: *IEEE Transactions on Software Engineering* 44.2 (2018), pp. 122–158. DOI: 10.1109/TSE.2017.2663435.

[102] Annibale Panichella et al. "Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities". In: *Proocedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2020, pp. 523–533. DOI: 10.1109/ICSME46990.2020.00056.

[103] Harrie Passier, Lex Bijlsma, and Christoph Bockisch. "Maintaining Unit Tests During Refactoring". In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ)*. Lugano, Switzerland, 2016. DOI: 10.1145/2972206.2972223.

[104] Mauro Pezzè and Michal Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008. ISBN: 8126517735.

[105] Duy Loc Phan, Yunho Kim, and Moonzoo Kim. "MUSIC: Mutation Analysis Tool with High Configurability and Extensibility". In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2018, pp. 40–46. DOI: 10.1109/ICSTW.2018.00026.

[106] Gustavo Pinto, Fernando Castor, and Yu David Liu. "Mining Questions about Software Energy Consumption". In: *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*. Hyderabad, India, 2014, pp. 22–31. ISBN: 9781450328630. DOI: 10.1145/2597073.2597110.

[107] Gustavo Pinto, Francisco Soares-Neto, and Fernando Castor. "Refactoring for Energy Efficiency: A Reflection on the State of the Art". In: *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software*. 2015, pp. 29–35. DOI: 10.1109/GREENS.2015.12.

[108] Gustavo Pinto, Weslley Torres, and Fernando Castor. "A Study on the Most Popular Questions about Concurrent Programming". In: *6th Workshop on Evaluation and Usability of Programming Languages and Tools*. Pittsburgh, PA, USA, 2015, pp. 39–46. DOI: 10.1145/2846680.2846687.

[109] Aleksandar Prokopec et al. "FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction". In: *Languages and Compilers for Parallel Computing*. Ed. by Hironori Kasahara and Keiji Kimura. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 158–173. ISBN: 978-3-642-37658-0. DOI: 10.1007/978-3-642-37658-0_11.

[110] Napol Rachatasumrit and Miryung Kim. "An empirical investigation into the impact of refactoring on regression testing". In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. 2012, pp. 357–366. DOI: 10.1109/ICSM.2012.6405293.

[111] Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. "A Comparison of Code Similarity Analysers". In: *Empirical Software Engineering* 23.4 (2018), pp. 2464–2519. ISSN: 1382-3256. DOI: 10.1007/s10664-017-9564-7.

[112] Khandoker Rahad, Zejing Cao, and Yoonsik Cheon. "A Thought on Refactoring Java Loops Using Java 8 Streams". In: (2017).

[113] Mohammad Rashid, Luca Ardito, and Marco Torchiano. "Energy Consumption Analysis of Algorithms Implementations". In: *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2015, pp. 1–4. DOI: 10.1109/ESEM.2015.7321198.

[114] Brian Robinson et al. "Scaling up Automated Test Generation: Automatically Generating Maintainable Regression Unit Tests for Programs". In: *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2011, pp. 23–32. DOI: 10.1109/ASE.2011.6100059.

[115] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. "Automated Unit Test Generation during Software Development: A Controlled Experiment

and Think-Aloud Observations". In: *Proocedings of the International Symposium on Software Testing and Analysis (ISSTA)*. Baltimore, MD, USA, 2015, pp. 338–349. DOI: 10.1145/2771783.2771801.

[116] G. Rothermel et al. "Prioritizing test cases for regression testing". In: *IEEE Transactions on Software Engineering* 27.10 (2001), pp. 929–948. DOI: 10.1109/32.962562.

[117] Cagri Sahin et al. "Initial explorations on design pattern energy usage". In: *2012 First International Workshop on Green and Sustainable Software (GREENS)*. 2012, pp. 55–61. DOI: 10.1109/GREENS.2012.6224257.

[118] Nedhal A Al-Saiyd. "The Impact of Reusing Open-Source Software Model in Software Maintenance". In: *International Journal of Computer Theory and Engineering* 9.1 (2017), p. 6. DOI: 10.7763/IJCTE.2017.V9.1101.

[119] Abdelilah Sakti, Gilles Pesant, and Yann-Gaël Guéhéneuc. "Instance Generator and Problem Representation to Improve Object Oriented Code Coverage". In: *IEEE Transactions on Software Engineering* 41.3 (2015), pp. 294–313. DOI: 10.1109/TSE.2014.2363479.

[120] Max Schäfer et al. "Refactoring Java Programs for Flexible Locking". In: *33rd International Conference on Software Engineering (ICSE)*. Waikiki, Honolulu, HI, USA, 2011, pp. 71–80. DOI: 10.1145/1985793.1985804.

[121] Norbert Schmitt et al. "Energy-Efficiency Comparison of Common Sorting Algorithms". In: *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2021, pp. 1–8. DOI: 10.1109/MASCOTS53633.2021.9614299.

[122] Rajni Sehgal et al. "Green software: Refactoring approach". In: *Journal of King Saud University - Computer and Information Sciences* 34.7 (2022), pp. 4635–4643. ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2020.10.022.

[123] Maged Shalaby et al. "Automatic Algorithm Recognition of Source-Code Using Machine Learning". In: *Proceedings of 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 2017, pp. 170–177. DOI: 10.1109/ICMLA.2017.00033.

[124] Shweta Sharma and S Srinivasan. "A Survey on Software Design Based and Project Based Metrics". In: *International Journal of Computer Theory and Engineering* 14.2 (2022), pp. 54–61. DOI: 10.7763/IJCTE.2022.V14.1310.

[125] Mohammed A Shehab et al. "An accumulated cognitive approach to measure software complexity". In: *Journal of Advances in Information Technology* 6.1 (2015), pp. 27–34. DOI: 10.12720/jait.6.1.27-33.

[126] Abdullah Sheneamer and Jugal Kalita. "Code clone detection using coarse and fine-grained hybrid approaches". In: *Proceedings of 7th IEEE International Conference on Intelligent Computing and Information Systems (ICICIS)*. 2015, pp. 472–480. DOI: 10.1109/IntelCIS.2015.7397263.

[127] Janet Siegmund. "Program Comprehension: Past, Present, and Future". In: *Proceedings of 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 5. 2016, pp. 13–20. DOI: 10.1109/SANER.2016.35.

[128] Moritz Sinn, Florian Zuleger, and Helmut Veith. "A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis".

In: *International Conference on Computer Aided Verification (CAV)*. Cham: Springer International Publishing, 2014, pp. 745–761. ISBN: 978-3-319-08867-9. DOI: 10.1007/978-3-319-08867-9_50.

[129] Nicholas Smith, Danny Van Bruggen, and Federico Tomassetti. "Javaparser: visited". In: *Leanpub, oct. de* (2017).

[130] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. "Automated Behavioral Testing of Refactoring Engines". In: *IEEE Transactions on Software Engineering* 39.2 (2013), pp. 147–162. DOI: 10.1109/TSE.2012.19.

[131] Shashank Srikant and Varun Aggarwal. "Automatic Grading of Computer Programs: A Machine Learning Approach". In: *Proceedings of 12th International Conference on Machine Learning and Applications (ICMLA)*. Vol. 1. 2013, pp. 85–92. DOI: 10.1109/ICMLA.2013.22.

[132] Benno Stein et al. "Safe stream-based programming with refinement types". In: *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2018, pp. 565–576. DOI: 10.1145/3238147.3238174.

[133] Ahmad Taherkhani. "Recognizing Sorting Algorithms with the C4.5 Decision Tree Classifier". In: *Proceedings of 18th IEEE International Conference on Program Comprehension (ICPC)*. 2010, pp. 72–75. DOI: 10.1109/ICPC.2010.11.

[134] Ahmad Taherkhani and Lauri Malmi. "Beacon-and Schema-Based Method for Recognizing Algorithms from Students' Source Code". In: *Journal of Educational Data Mining* 5.2 (2013), pp. 69–101. DOI: 10.5281/zenodo.3554635.

[135] Ahmad Taherkhani, Lauri Malmi, and Ari Korhonen. "Algorithm Recognition by Static Analysis and Its Application in Students' Submissions Assessment". In: *Proceedings of 8th ACM International Conference on Computing Education Research (ICER)*. 2008, pp. 88–91. ISBN: 9781605583853. DOI: 10.1145/1595356.1595372.

[136] Suresh Thummalapenta et al. "Synthesizing Method Sequences for High-Coverage Testing". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. Portland, Oregon, USA, 2011, pp. 189–206. ISBN: 9781450309400. DOI: 10.1145/2048066.2048083.

[137] Kai Tian, Meghan Revelle, and Denys Poshyvanyk. "Using Latent Dirichlet Allocation for automatic categorization of software". In: *Proceedings of 6th IEEE International Working Conference on Mining Software Repositories (MSR)*. 2009, pp. 163–166. DOI: 10.1109/MSR.2009.5069496.

[138] Emiliano Tramontana. "Automatically Characterising Components with Concerns and Reducing Tangling". In: *2013 IEEE 37th Annual Computer Software and Applications Conference Workshops*. 2013, pp. 499–504. DOI: 10.1109/COMPSACW.2013.114.

[139] Nikolaos Tsantalis, Davood Mazinanian, and Giri Panamoottil Krishnan. "Assessing the Refactorability of Software Clones". In: *IEEE Transactions on Software Engineering* 41.11 (2015), pp. 1055–1090. DOI: 10.1109/TSE.2015.2448531.

[140] Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. "Clone Refactoring with Lambda Expressions". In: *2017 IEEE/ACM 39th International*

Conference on Software Engineering (ICSE). 2017, pp. 60–70. DOI: 10.1109/ICSE.2017.14.

[141] Secil Ugurel, Robert Krovetz, and C. Lee Giles. "What's the Code? Automatic Classification of Source Code Archives". In: *Proceedings of 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. Edmonton, Alberta, Canada, 2002, pp. 632–638. DOI: 10.1145/775047.775141.

[142] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft. *Java 8 in action*. Manning publications, 2014. ISBN: 1617291994.

[143] Hans Van Vliet, Hans Van Vliet, and JC Van Vliet. *Software engineering: principles and practice*. Vol. 13. John Wiley & Sons Hoboken, NJ, 2008. ISBN: 0470031468.

[144] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. "The paralax infrastructure: Automatic parallelization with a helping hand". In: *Proceedings of 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2010, pp. 389–399. DOI: 10.1145/1854273.1854322.

[145] Sebastian Vogl et al. "Evosuite at the SBST 2021 Tool Competition". In: *Proocedings of Workshop on Search-Based Software Testing (SBST)*. 2021, pp. 28–29. DOI: 10.1109/SBST52555.2021.00012.

[146] Frens Vonken and Andy Zaidman. "Refactoring with Unit Testing: A Match Made in Heaven?" In: *2012 19th Working Conference on Reverse Engineering*. 2012, pp. 29–38. DOI: 10.1109/WCRE.2012.13.

[147] Claas Wilke et al. "Energy Consumption and Efficiency in Mobile Applications: A User Feedback Study". In: *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*. 2013, pp. 134–141. DOI: 10.1109/GreenCom-iThings-CPSCom.2013.45.

[148] Jan Wloka, Manu Sridharan, and Frank Tip. "Refactoring for reentrancy". In: *7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2009, pp. 173–182. DOI: 10.1145/1595696.1595723.

[149] Michael Wolfe and Utpal Banerjee. "Data dependence and its application to parallel processing". In: *International Journal of Parallel Programming* 16.2 (1987), pp. 137–178. DOI: 10.1007/BF01379099.

[150] Shengwei Xu, Huaikou Miao, and Honghao Gao. "Test Suite Reduction Using Weighted Set Covering Techniques". In: *13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. 2012, pp. 307–312. DOI: 10.1109/SNPD.2012.87.

[151] S. Yoo and M. Harman. "Regression Testing Minimization, Selection and Prioritization: A Survey". In: *Software Testing, Verification & Reliability* 22.2 (2012), pp. 67–120. ISSN: 0960-0833. DOI: 10.1002/stv.430.

[152] Yang Zhang. "Improving the learning of parallel programming using software refactoring". In: *Computer Applications in Engineering Education* 25.1 (2017), pp. 112–119. DOI: 10.1002/cae.21784.

[153] Yang Zhang, Liuxu Li, and Dongwen Zhang. "A survey of concurrency-oriented refactoring". In: *Concurrent Engineering* 28.4 (2020), pp. 319–330. DOI: 10.1177/1063293X20958932.

[154] Yang Zhang et al. "Automated Refactoring for Stampedlock". In: *IEEE Access* 7 (2019), pp. 104900–104911. DOI: 10.1109/ACCESS.2019.2931953.

[155] Yang Zhang et al. "FineLock: automatically refactoring coarse-grained locks into fine-grained locks". In: *ACM International Symposium on Software Testing and Analysis (ISSTA)*. 2020, pp. 565–568. DOI: 10.1145/3395363.3404368.

[156] Yang Zhang et al. "Refactoring Java Programs for Customizable Locks Based on Bytecode Transformation". In: *IEEE Access* 7 (2019), pp. 66292–66303. DOI: 10.1109/ACCESS.2019.2919203.

[157] Yucheng Zhang and Ali Mesbah. "Assertions Are Strongly Correlated with Test Suite Effectiveness". In: *Proocedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE)*. Bergamo, Italy, 2015, pp. 214–224. DOI: 10.1145/2786805.2786858.