TÉCNICO LISBOA

Mono2Micro: Enriching Access Sequences Static Analysis Collector with Branch Frequency

Paulo Miguel Jacinto Ferreira do Ó

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. António Manuel Ferreira Rito da Silva

Examination Committee

Chairperson: Prof. Valentina Nisi
Supervisor: Prof. António Manuel Ferreira Rito da Silva
Member of the Committee: Prof. Nuno Claudino Pereira Lopes

May 2024

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to thank my Mom, Dad. siblings and my Grandma, who deeply supported my academic path from the start.

Would also like to thank my partner and my friends, both from inside and outside Técnico, for being an important part of my journey and filling it with great moments and memories.

Last but not least, I'd like to thank my supervisor Prof. António Rito Silva, for his guidance and support throughout the development of this dissertation.

Institutional Acknowledgment:

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT), Portugal through projects UIDB/50021/2020 (INESC-ID) (DOI:10.54499/UIDB/50021/2020) and PTDC/CCI-COM/2156/2021 (DACOMICO) (DOI:10.54499/PTDC/CCI-COM/2156/2021).

Abstract

Microservice-centric development is one of the architectural styles that is more frequently applied in software development. Nevertheless, many applications start as monolith system that grow in size until the transition to the microservice architecture becomes necessary. This process requires highly specialized knowledge, and several tools over the years have attempted to provide automation or informed assistance to the developers responsible for the migration. The main techniques used by these tools for the analysis of the monolith system fall into two categories, Dynamic and Static Analysis. Each brings trade-offs when balancing execution time and amount of input data necessary with precision of the results. This thesis proposes a static analysis technique that borrows from dynamic analysis, in order to attempt to simulate its result accuracy while keeping the low resource requirements of static techniques. This approach is built on top of the Mono2Micro monolith decomposition tool, which already includes multiple decomposition strategies of both static and dynamic types, and is built as an extension to one that was previously developed - based on sequences of accesses. The new technique is evaluated in terms of the metrics of the decompositions generated along with the semantic analysis of said decompositions.

Keywords

Monolith decomposition strategies, microservices, static analysis, dynamic analysis

Resumo

Desenvolvimento centrado em microserviços é um dos estilos de arquitetura mais frequentemente aplicado em desenvolvimento de software. Apesar disso, muitas aplicações começam como sistemas monolíticos que crescem em tamanho ao ponto que a transição para a arquitetura de microserviços se torna necessária. Este processo requere conhecimento especializado, e muitas ferramentas ao longo dos anos vêm tentado oferecer métodos de automação ou de assistência aos desenvolvedores responsáveis para a migração. As principais técnicas utilizadas por estas ferramentas para a análise dos monólitos caem em duas categorias, Análise Dinâmica e Estática. Cada uma possui compromissos ao balancear tempo de execução e quantidade de informação de input necessária com a precisão dos resultados. Esta tese propõe uma técnica de análise estática que toma inspiração de análise dinâmica, de modo a tentar simular a sua exatidão dos resultados e ao mesmo tempo manter os requerimentos baixos de recursos das técnicas estáticas. esta técnica é criada como parte da ferramenta Mono2Micro para decomposição de monólitos, que já inclui várias técnicas de decomposição tanto de tipo estático como dinâmico, e consiste na extensão de outra técnica previamente desenvolvida - que se baseia em sequências de acessos. A nova técnica é avaliada em termos das métricas das decomposições geradas e da análise semântica das mesmas.

Palavras Chave

Estratégias de decomposição de monolitos, microserviços, análise estática, análise dinâmica

Contents

1 Introduction **3**

1.1 Problem . . . 4

1.2 Research Questions . . . 4

1.3 Organization of the Document . . . 5

2 Background and Related Work **7**

2.1 Current State of Microservice Identification . . . 7

2.2 Describing a Monolith . . . 8

2.3 Stages of the identification . . . 9

2.4 State of the art for data collection . . . 11

2.5 Static Branch Frequency . . . 12

3 Enriching Static Analysis **15**

3.1 Problem . . . 15

3.2 Static Collection . . . 17

3.2.1 If statements . . . 17

3.2.2 Loops . . . 21

3.2.3 Function calls . . . 23

3.2.4 Switch statements . . . 24

3.2.5 Abstract functions calls . . . 25

3.3 Branch Heuristics . . . 26

3.4 Graph generation . . . 29

3.4.1 If (*IfGraph*) . . . 31

3.4.2 Loops (*LoopGraph*) . . . 32

3.4.3 Call (*CallGraph*) . . . 34

3.4.4 Label (*LabelNode* . . . 35

3.4.5 Abstract Call (*AbstractCallGraph*) . . . 36

3.4.6 Switch (*SwitchGraph*) . . . 36

3.4.7 Access (*AccessNode*) . . . 37

3.4.8 Calculating the Branch Probabilities - Heuristics . . . 37
3.5 Similarity measure calculation . . . 41
4 Evaluation **45**
4.1 Evaluation Methodology . . . 46
4.1.1 MOJO . . . 47
4.1.2 Statistical Analysis . . . 48
4.2 Comparison of best candidate decompositions by metric . . . 49
4.3 MOJO comparison . . . 50
4.4 Statistical Analysis . . . 50
4.5 Graph vs Sequence vs Source of Truth . . . 51
4.6 Discussion . . . 60
4.7 Threats to Validity . . . 60
5 Conclusion **63**
Bibliography **64**

List of Figures

3.1 Domain model of Mono2micro's logic responsible for graph processing 31
3.2 Sample of possible combination of If components 33
3.3 Sample of possible combination of Loop components 34
3.4 Graph result of Call 35
3.5 Comparison between the connection of a Call without return, and another with 36
3.6 Graph result of an Abstract Call 37
3.7 Graph result of a Switch 38
3.8 Example of heuristics applicable in a branch decision. 40

List of Tables

3.1 Branch frequency for each heuristic, as observed by Wu and Larus [1]. 28

4.1 Comparing the best metric values achieved with each strategy and the source of truth . . 50

4.2 MoJoFM distance comparing both strategies and the expert decomposition. Each column stipulates the decompositions that are used in the comparison (eg: the ones with the best complexity of each strategy). 50

4.3 Statistical analysis of each of the best metric decompositions, per metric and strategy, performed with MOJO and comparing against the expert decomposition. 51

4.4 For the best complexity decomposition, using the Sequence Strategy, each cell contains the percentage of entities of the expert microservice (column) in the decomposition cluster (row). 52

4.5 For the best complexity decomposition, using the Graph Strategy, each cell contains the percentage of entities of the expert microservice (column) in the decomposition cluster (row). 52

4.6 For the best cohesion decomposition, using the Sequence Strategy, each cell contains the percentage of entities of the expert microservice (column) in the decomposition cluster (row). 52

4.7 For the best cohesion decomposition, using the Graph Strategy, each cell contains the percentage of entities of the expert microservice (column) in the decomposition cluster (row). 53

4.8 For the best coupling decomposition, using the Sequence Strategy, each cell contains the percentage of entities of the expert microservice (column) in the decomposition cluster (row). 53

4.9 For the best coupling decomposition, using the Graph Strategy, each cell contains the percentage of entities of the expert microservice (column) in the decomposition cluster (row). 53

4.10 Average size of the cluster containing each entity. Decompositions are grouped by strategy and the location (cluster) of each decomposition is recorded on a decomposition basis, and the size of that location is averaged to show entities with tendency to be placed in bigger clusters. 54
4.11 Number and percentage of functionalities in common between QuizAnswer and each of the other entities in the Answer microservice. Total number of functionalities of QuizAnswer is indicated in the last row. 55
4.12 Number and percentage of functionalities in common between QuestionAnswer and each of the other entities in the Answer microservice. Total number of functionalities of QuestionAnswer is indicated in the last row. 56
4.13 Number and percentage of functionalities in common between Question and each of the other entities in the Question microservice. Total number of functionalities of Question is indicated in the last row. 56
4.14 Similarity between QuizAnswer and the rest of the expert microservice Answer. Each column contains a different similarity measure, and each row represents an entity of the microservice. The values are not symmetric in both directions of the relationship. First graph shows the results for Sequence Strategy, and bottom shows the results for Graph Strategy. 57
4.15 Similarity between QuestionAnswer and the rest of the expert microservice Answer. Each column contains a different similarity measure, and each row represents an entity of the microservice. The values are not symmetric in both directions of the relationship. First graph shows the results for Sequence Strategy, and bottom shows the results for Graph Strategy. 58
4.16 Similarity between QuestionAnswer and the rest of the expert microservice Answer. Each column contains a different similarity measure, and each row represents an entity of the microservice. The values are not symmetric in both directions of the relationship. First graph shows the results for Sequence Strategy, and bottom shows the results for Graph Strategy. 59

1

Introduction

Contents

1.1 Problem 4

1.2 Research Questions 4

1.3 Organization of the Document 5

In order to keep up with quickly growing demands from markets and their users, many aging and critical enterprise applications grow in a monolithic way. Although this brings advantages, such as fast development and better handling of more complex domain models, the intertwinedness of the applications' functionalities hinders their independent scalability and imposes roadblocks on the team's ability to deliver new features quickly. This brought up the use of patterns such as modularity or microservices. A microservices-centered design is especially relevant in the era of cloud computing, as it not only effectively separates the different functionalities of an application, it also allows each one to be scaled individually, according to the changing needs and demand.

Although a microservice-centric design architecture is beneficial in these circumstances, the higher development cost that it brings is sometimes not worth it for some companies or smaller developers, due to both the need for highly specialized architects who need to evaluate the system and the high amount of work and money required to transition all the live systems in play. One of the main obstacles to this

process, and the reason why highly specialized personnel is required, is the difficulty in identification of the each of the multiple parts of the system that can be split into different microservices.

Multiple approaches have been studied over the years, proposing various criteria for the division, such as a focus on the domain model of the system, or the clear delimitation of user functionalities. As such, many tools have also been developed over the years, and along with them, techniques to collect data that can be applied to the analysis of such divisions.

These techniques differ in the way the data they require is collected. Dynamic analysis, for example, focuses on the inspection of logs generated from user activity to learn how functionalities are used together. This approach can give a closer look at how the application should be structured more efficiently for daily use. By examining the exact runtime behavior of the system, it provides a very precise representation of its properties; however, it requires a great amount of data to be collected and processed, which can be a very expensive task. Another technique is static analysis, which leverages introspective tools such as, in the case of Java, JavaParser, or Spoon to collect information about the access to domain entities during the execution of each functionality. Although this approach does not require a large set of data to analyze, like dynamic analysis, it does not provide an equally accurate or precise model of the workings of the system when utilized by average users. These approaches, although sometimes seen as competing and distinct, often present a very similar structure and can complement each other [2].

1.1 Problem

While dynamic analysis is limited by the cost of collecting and processing data, static analysis is hindered by the fact that it produces a less precise portrait of the evaluated system. This is because static analysis is limited in the collection of certain types of data that are captured or at least considered in the case of dynamic analysis.

A possible strategy to combat this it creating a hybrid approach that empowers static analysis with strengths similar to those of dynamic analysis, such as context regarding the control flow of the program, can create a better approach to study and evaluate monolithic systems and improve the quality of the microservice decompositions generated.

1.2 Research Questions

With this issue in mind, this document explores a new technique to approximate control flow data in static analysis, inspired by compiler branch prediction strategies.

A study will be conducted on a target codebase in order to evaluate the tool's division of the system with the new technique, which will be integrated in the current Mono2Micro pipeline.

The tool generates multiple decompositions according to different measures of similarity between the elements of the decomposition. To differentiate and select the optimal decomposition of the system, quality metrics have been proposed and adopted in the past. These metrics quantify the complexity, coupling, and cohesion of the proposed microservice decomposition. It is important to study whether the added information derived from the control flow of the system is used in the similarity measures generate better microservice candidate decompositions, considering the mentioned metrics.

Additionally, since we are analyzing a specific codebase, this also allows for a closer and more intimate reading of the results. In this case, we are looking to assess whether this new data collection technique creates a result closer to the system expert decomposition. Two approaches are taken for this purpose, a quantitative evaluation using a distance measure such as MoJoFM [3], and a qualitative evaluation that performs a semantic analysis of the candidate decomposition.

With this in mind, we are looking to answer the following questions:

- RQ1: Does control flow information affect the quality of microservice decompositions generated through static analysis?
- RQ2: How do candidate decompositions generated using static analysis, with and without control information, compare to an expert decomposition?

Through RQ1 we are looking to find if the new information derived from the codebase will, in fact, translate into an improvement in terms of generated decompositions, or if the current technique performs better with less information. At the same time, it is important to visualize how the newly proposed methods perform when compared to an expert decomposition, as the goal is not only to provide better results than the current approach but also to achieve decompositions that are closer to the vision of the architect of the system. This will be answered with RQ2.

1.3 Organization of the Document

This document is organized as follows: Chapter 2 will specify and describe the current work performed on static analysis, particularly the way it is implemented in the tool Mono2Micro as it will serve as the base for this study and the state of the art and the work on which this study is based. Chapter 3 describes the details of the proposed solution, including how it influences the pipeline of the original tool. In Chapter 4 the answer to the research questions is investigated, as we evaluate the output results of the modified tool, as well as compare it to results generated by the previous implementation of static analysis and dynamic analysis. It also discusses the various points of the study and what was achieved, along with possible threats to validity and future work. Finally, Chapter 5 presents the conclusion and summary of the acquired results.

2

Background and Related Work

Contents

2.1 Current State of Microservice Identification 7

2.2 Describing a Monolith 8

2.3 Stages of the identification 9

2.4 State of the art for data collection 11

2.5 Static Branch Frequency 12

2.1 Current State of Microservice Identification

In this section, we will go over details regarding the current implementation of the Mono2Micro tool.

When studying the transition between a monolith and microservices there are some concepts that are important to consider.

2.2 Describing a Monolith

A monolith is described by the set of functionalities that make up it. Each functionality is a set of domain entities accesses. The domain entities represent the monolith's persistent elements, the ones that correspond to database tables.

Therefore, a sequence of accesses triggered by the execution of a monolith functionality is executed as a single transaction.

However, when a monolith is decomposed into microservices, some of its functionalities may be executed in more than one microservice. In these situations, we say that the functionality is decomposed into a set of local transactions, where each local transaction occurs inside a single microservice. The functionality consists of consecutive accesses that associated to different local transactions correspond to a distributed remote invocation between their associated microservices.

Running and coordinating remote or distributed local transactions is a costly endeavor, due to the required maintenance of concurrency and fault tolerance between independent systems, so the transition to a microservice architecture requires balancing more, smaller microservices (easier to deploy and scale with the system needs, but employing a lot of expensive distributed transactions), with fewer, bigger microservices (harder to scale due to the large amount of resources they require, but bring a lot of cheaper local transactions).

With this in mind, Mono2Micro currently considers the following definitions [4]:

- A Monolith is a triple (F, E, G), as it is composed by F, a set of functionalities; E, a set of domain entities; and G, a group of call-graphs, one for each functionality and describing how the entities are accessed during it's execution.

- A Call Graph is a defined tuple (A, P), where A is a set of domain entity accesses, either read or write, therefore $A = E \times M$, $M = \{r, w\}$; and $P = A \times A$ a relation of precedence between elements of A in such way that each access has zero or one immediate predecessor, $\forall_{a \in A} \#\{(a1, a2) \in P : a1 = a\} \leq 1$, and there are no circularities, $\forall_{(a1,a2) \in P_T}(a1, a2) \notin P_T$, where P_T is the transitive closure of P.

To illustrate this we can consider monolith $M_1 = (F, E, G)$ with $F = \{f_1, f_2\}$, containing all its functionalities and $E = \{e_1, e_2, e_3\}$ containing all its entities. We can consider $G = \{cgf_1, cgf_2\}$ the set of call-graphs of $M1$ (cgf_1 being the call-graph of functionality f_1, and same between cgf_2 and f_2). We can further define $cgf_1 = \{(e_1, r), (e_2, w)\}$ meaning the functionality f_1 is a sequence of accesses where e_1 is read and then e_2 is written, in that order.

For this monolith a possible decomposition could be into two clusters/microservices $C_1 = \{e_1, e_3\}$ and $C_2 = \{e_2\}$. This would mean cgf_1 would be split into two separate local transactions, $cgf_1C_1 = \{(e_1, r)\}$

which happens first and $cgf_1C_2 = \{(e_2, w)\}$ which occurs second. This means that for this particular decomposition the two local transactions would need to be coordinated, which is costly in a network.

Therefore, we could consider a different decomposition, with $C_1 = \{e_1, e_2\}$ and $C_2 = \{e_3\}$. This way, since e_1 and e_2 remain in the same cluster, they can be accessed in the same local transaction ($cgf_1C_1 = \{(e_1, r), (e_2, w)\}$) and without the need for distributed coordination, reducing the overall cost of the functionality.

2.3 Stages of the identification

The Mono2Micro pipeline has multiple identifiable stages, which are shared with other microservice identification tools [5] [6] [7] [8] [9] [10] [11]. These stages are described as following [12]:

- **Collection** - As the name indicates, this stage is responsible for collecting data that represents the system, which will then be used to generate the decompositions. This data can come from multiple sources, including source code, version control activity logs, user logs, among many others.

  The Collection stage is where approaches such as static and dynamic analysis differ the most, since they usually end up following similar processes in the following stages.

  Mono2Micro employs a number of these techniques which can be used separately or paired in order to provide more informed representations of the monolith.

- **Decomposition generation** - During the decomposition generation stage, the tools apply clustering techniques in order to create candidate microservice decompositions for the system. In the case of Mono2Micro a Hierarchical Clustering Technique is used. This approach groups the domain entities of the system in Clusters in order to minimize the number of distributed transactions per functionality.

  For this purpose, the background work [4] defines similarity measures, which establish the distance between domain entities. Entities that are closer are often accessed together and therefore should be placed in the same microservice. Four similarity measures were defined: Access, Read, Write and Sequence.

  – Access Similarity: as the base principle states, entities that are accessed by the same functionalities should be placed together, and therefore should have a higher similarity score. Considering $funct(e)$ the set of functionalities of the monolith that access domain entity e, then the Access Similarity between two entities $e_1, e_2 \in E$ is therefore measured as:

$$sm_{access}(e_1, e_2) = \frac{\#(funct(e_1) \cap funct(e_2))}{\#funct(e_1)}$$

The value of this measure is within the interval 0..1, with 0 meaning that none of the functionalities that access e_1 access e_2, and 1 meaning that every functionality that accesses e_1 will also access e_2.

- Read and Write Similarities: Although the same set of entities is often accessed together, distinguishing whether those accesses are reads or writes is important, especially when considering a distributed environment. With this in mind, the Read and Write similarities are defined as following:

$$sm_{read}(e_1, e_2) = \frac{\#(funct(e_1, r) \cap funct(e_2, r))}{\#funct(e_1, r)}$$

$$sm_{write}(e_1, e_2) = \frac{\#(funct(e_1, w) \cap funct(e_2, w))}{\#funct(e_1, w)}$$

where $funct(e, m), m \in r, w$ defines the number of functionalities that access the domain entity e in read(r) or write(w) mode, respectively.

- Sequence Similarity: When domain entities are usually accessed one after the other, their relationship can be interpreted as stronger than registered by the previous similarities, as on top of being accessed by the same functionalities they are also repeatedly accessed in sequence, meaning that factors such as remote transaction latency become more prominent. This means that entities usually accessed in sequence need to be accounted for, so we define

$$sm_{sequence}(e_1, e_2) = \frac{sumPairs(e_1, e_2)}{maxPairs}$$

where $sumPairs(e_1, e_2) = \sum_{f \in F} \#\{(a_i, a_j) \in G_f.P : (a_i.e = e_1 \wedge a_j.e = e_2) \vee (a_i.e = e_2 \wedge a_j.e = e_1)\}$ is the number of consecutive accesses of e_1 and e_2, no matter the order, with $G_f.P$ being the precedence relation for functionality f, and $maxPairs = max_{e_i, e_j \in E}(sumPairs(e_i, e_j))$ is the maximum number of consecutive accesses between any two entities.

The values of these similarity measures range between 0 and 1, with 0 meaning a non-existent relationship between the pair of analyzed entities, which therefore means these entities do not need to be placed in the same cluster, while 1 reflects an absolute certain correlation between the accesses and that separating them into separate microservice groups would generate a high distributed transaction cost and heavily affect the system.

- **Quality Assessment and Comparison** - After the identification of the decompositions in the previous stage, they will be evaluated using quality metrics, of which there are four main groups:

  - Cohesion [13]: represents the strength of the relationship between entities of the service;

  - Complexity [14]: considers the effort to implement a functionality as a distributed transaction. This effort depends on the number of local transactions the functionality is decomposed on and the intermediate states that have to be managed due to the lack of isolation. Therefore a higher complexity is less ideal, as it represents a harder effort to migrate each monolith functionality from an atomic transaction to a set of distributed microservice transactions;
  - Coupling [15]: Higher modularity is prioritized for better maintainability. With that in mind, Coupling expressed the level of dependency between different microservices;
  - Size [14]: While a bigger Cluster size means harder maintainability, a smaller size means more Clusters and therefore more distributed transactions. Therefore a balance needs to be stricken between the number and the size of Clusters in the decomposition.

  The use of metrics is vital for the instant analysis of the decompositions, however, some tools opt to only use them *a posteriori* to compare the results with those achieved through other tools.

- **Visualization** - Visualizing the end result is a good way to reason regarding the decompositions. For this, multiple visualization formats have been proposed such as Cluster Graphs [16] [9] (node graphs where each node represents a microservice/cluster, and the edges between nodes represent the dependencies between the microservices), Class/entity Graphs [17] [5] [7] (also node graphs, however, each node represents a class or entity, with the edges showing the dependencies between them. Each entity is still related to a microservice) and Sequence of Accesses (less focused on microservices and more focused on the sequence of accesses made to perform a certain function of the system. This changes slightly depending on the type of analysis of the code, such as dynamic or static);

- **Editing and Modelling** - After visualizing the resulting decompositions it is also important to provide architects with a way to better tailor the decompositions to their needs and the business needs of the application. This comes through providing a wise array of editing and modelling operations such as creation of new microservices and moving entities between microservices, among others.

2.4 State of the art for data collection

The first step in the process of decomposing a monolith is always to gather information about it. The techniques used for data collection can be divided into two main groups: static analysis and dynamic analysis.

Static analysis is based on analysing the content of the program, and deriving from it information that allows learning specific properties common in most executions of the program [18] [4]. Dynamic, on the other hand, is the analysis of the program while it runs. This can be done through the collection of

user logs, keeping data relating to the classes and methods that are used and their order [19] [8]. With massive amounts of data, it is then possible to average precise information about the system, how it is used, and how it can be divided.

The choice between using one type of technique or the other is a trade off. Dynamic analysis is more precised and possibly more relevant to the user needs of the system, but it requires massive amounts of usage logs and processing time and resources to arrive to conclusions. Static analysis requires much less data to work and processes significantly faster however it provides more generic results, specially when compared with the precision of dynamic analysis.

Some authors suggest that the ideal method is to use both types of strategy in tandem, complementing each other's results and adjusting the weight of each strategy depending on the situation at hand [20]. Plenty of tools already allow this, including Mono2Micro which supports dynamic trace analysis and multiple types of static analysis simultaneously, and has been used to evaluate the trade off between both types of techniques [21]

2.5 Static Branch Frequency

Optimization is a primary goal in many disciplines of software development. However, one of the fields where it is the most prevalent is compilers. Compilers often have the role of not only transpiling the code from its original language into a lower-level or even machine-level one, but also of optimizing it the best they can. One of the optimization strategies that has been developed over the years has been ordering the compiled code according to what is going to be executed more frequently or with other sections of the code. By identifying these hot-paths in the code [22] [23] [24], compilers can reduce the amount of jumps to be preformed at runtime, reducing the overhead of executing the most traversed areas of the program.

This strategy can be based on data from various sources. Among those, Ball and Larus [25] proposed a program-based branch predictor for C and Fortran programs, which allowed us to predict the execution of branches in the code. Their program is based on a set of heuristics that, based on observable and proved properties in the code, allows the compiler to make a quick but fairly informed decisions on the hot-paths of the program.

Furthermore, in a more recent research, Wu and Larus [1] continued to work on these heuristics by performing a static analysis of the branch frequency in the situations contemplated in them. In their study, by analyzing the C and Fortran programs during execution, they were able to calculate the precision of each of the heuristics proposed in [25]. Their results provide a fairly accurate value for the probability of a given branch's execution when the conditions of the heuristics apply.

In Chapter 3, each of the heuristics is explained, along with its adaptation from C and Fortran into

Java and the reasoning behind them.

3

Enriching Static Analysis

Contents

3.1 Problem 15
3.2 Static Collection 17
3.3 Branch Heuristics 26
3.4 Graph generation 29
3.5 Similarity measure calculation 41

In this chapter, we present the architecture of the solution and its implementation. We will start by describing and justifying in a general manner the changes made to the tool, and then follow each change and the steps influenced in the order they occur in the Mono2Micro pipeline, starting with the data collection phase, and then moving on to the changes to decomposition generation.

3.1 Problem

To illustrate the changes needed to the Mono2Micro implementation, we introduce a small example:

```
if (A) {
    B;
} else {
    C;
}
```

where A, B and C constitute memory accesses to 3 different domain entities, all of the read type. Currently, the trace collected by the tool during the data collection phase is the following:

```
Trace File:
"functionality": {
    t: [{
        id: 0,
        a: [["R", A], ["R", B], ["R", C]]
    }]
}
```

The trace incorrectly indicates that entities A, B, and C are accessed in sequence. However, through the code, this is observed to be incorrect, as either the sequence A-B will happen or A-C, but never A-B-C. By analysing the code and assuming a non-null probability of the `If`'s condition having value false, the probability of B being accessed is ¡100%, and the same for C. Therefore, the need is recognized for a format that better represents these diverging and mutually exclusive access sequences.

However, another issue arises. In the case of the previous example, although we know that the probability of paths A-B and A-C being following is different, it is still unknown to the tool how likely the application is to follow each of the paths. What we require (a trace of accesses informed by the way the code of the application is run) is a current identifying feature of dynamic analysis [21]. The problem with dynamic analysis is that it requires a large amount of data from user-generated logs. How can static analysis approximate this behavior with enough accuracy while dealing with context knowledge of the code limited by what is written in the program and without requiring the usage logs?

The proposed solution strategy has 3 requirements:

- Gather control flow information into the functionality traces;
- Approximate path probabilities without requiring user logs;
- Use control flow information to better reflect the system in the similarity measures.

3.2 Static Collection

The gathering of information about the system starts in the *Collection* phase. Therefore, our solution must also start there. The solution previously presented considered the sequential occurrence of domain entity accesses and produced linear traces per functionality. This means that the json file representing the codebase has the following elements:

- Atomic Elements - the domain entity Accesses, which can be either reads or writes
- Sequences - arrays of Atomic Elements, in the the order in which the appear in the code
- Functionalities - each containing the Sequence of accesses that occur in the execution of said functionality

Our goal is to add control flow information collected from the program and, as such, we need to define where this flow conditionally branches. Knowing how to program, we know that control flow of the code splits in multiple identifiable points, such as if statements, for and while loops, and switch cases. On top of that, jump statements, such as breaks, continues, or returns, also affect the flow of the program in a significant way. All of these elements work in different ways and have different properties. Therefore, taking into account the previously defined elements, the new trace should have the following elements:

- *Atomic Elements* - remain the same as before: the domain entity Accesses, which can be either reads or writes;
- *Branching Elements* - programming elements that represent the conditional execution of the code (if, else, for, etc; even control instructions such as return are included in this group), Branching elements generate both the split and the join between sections of the code, therefore they apply to the *Sequence* element;
- *Sequences* - arrays of *Atomic Elements* and *Branching Elements*, in the the order in which the appear in the code;
- *Functionalities* - each functionality will now contain many *Sequences*, which are connected between each other through *Branching Elements*.

With this in mind, we identify the main Branching Elements we are trying to register.

3.2.1 If statements

If statements are the basis of computer logic, consider a condition and two blocks of code. If the condition's value is evaluated to true, the first block of code will be executed. However, if the condition

evaluates to false, the second block of code will be executed instead. In a single run, it is guaranteed that the condition will always be run once and that one and only one of the blocks of code will be executed. In this way we can consider an *If* element as being composed of *condition*, *then* and *else*.

Looking at our previous example:

```
if (A) {
    B;
} else {
    C;
}
```

The new trace representing this code excerpt would be the following:

```
Trace File:
"functionality": {
    t: [{
        id: 0,
        a: [["&if", 1]]
    },
    {
        id: 1,
        a: [["&cond", 2], ["&then", 3], ["&else", 4]]
    },
    {
        id: 2,
        a: [["R", A]]
    },
    {
        id: 3,
        a: [["W", B]]
    },
    {
        id: 4,
        a: [["W", C]]
    }
    ]
}
```

The Functionality contains multiple Sequences, each containing Atomic and Branching elements (in this case *If*, *then* and *else*). Note that each Sequence is identified by a number, with Sequence 0 (zero) being the main Sequence, from which all of the other will branch out of. Each of the Branching elements is represented by a string, declaring their type, and an integer, denoting the Sequence that describes the content of that element; $[\text{"}\&cond\text{"}, 2]$ means that the execution of the *condition* is defined in the sequence with id 2 and, therefore, contains the access to domain entity A. This format was selected to be similar to the previously available format for the Atomic elements, also described by a string (access mode) followed by an integer (entity id). A point to note is that although in this document we represent a read access to entity A as *["R", A]*, the actual program trace will contain *["R", ¡i¿]*, where *¡i¿* is the representation id of entity A. However, the use of integer instead of strings is only due to performance reasons and is not relevant for this presentation.

To register each Sequence and its index, the new collector uses a stack. The collector is based on Spoon [26], an open-source library that can be used to analyze, rewrite, transform, and transpile Java source code. This library generates an AST representation of the codebase that can then be analyzed using a recursive scanner.

This recursive scanner is the base behavior of the library and can be overridden with custom methods, which is how the previous iteration of the collector stores access data. In our case, we are overriding the scanner so that once it finds one of the targeted control branching elements, in this case an *If* statement, represented by a *CtIfImpl*, it will open a new "context" to which is assigned an *id* based on a global counter. An entry is open in the trace dictionary, unique per functionality, corresponding the context *id* to the array of accesses that happens in that context/sub-trace. Similarly, when the scanner finds a child branching component that plays a role (in this case *cond*, *then* and *else*, which are represented by enumerated values *CtRole.CONDITION*, *CtRole.THEN* and *CtRole.ELSE* and stored as the *CtRole* of the current evaluated entity). Due to the recursive nature of the scanner, all the accesses that happen in that context will be stored in the respective "if" Sequence. At the end of the *Collection* phase, the content of the dictionary represents the set of Sequences that occur within the respective functionality.

Returning to the *If* case, some other particular forms of *If* can be identified as well. The most important ones are *else if* and *If* statements that do not have child elements (such as *If* with no *else*). In reality, *else if* blocks work as if statements within the else component of an *If*, therefore, it can be split into

```
1 if (A) {
2     B;
3 } else if (C) {
4     D;
```

```
} else {
    E;
}

into

if (A) {
    B;
} else {
    if (C) {
        D;
    } else {
        E;
    }
}
```

and are equally represented in the trace. On the other hand, an *If* statement without an *else* portion can just be represented as:

```
...
    {
        id: 1,
        a: [["&cond", 2], ["&then", 3]]
    },
...
}
```

There are more situations where we consider that a child of a Branching element can be missing. In many situations, there will be code executed that is purely functional and does not read or write context data. Even in the previous iteration of the collector, these sections are not represented in the Sequence, as they will not affect the relationship between functionalities and domain entities, which is taken into account in the creation of the candidate decompositions due to their lack of Accesses. In the new collector, these data will be disregarded similarly. The collector will only register data that are relevant to the decomposition process, and the Mono2micro tool will be prepared to make assumptions based on the missing data. This also works as an optimization as it prevents the collector from unnecessarily increasing the trace file size.

3.2.2 Loops

Other important elements for controlling the flow of a program are loops. These can come in various formats (such as while loops, for loops, for-each loops, do-while loops, etc.), but they can be abstracted to a main concept: a single *condition*, which will be verified in each of the consecutive executions of the loop, and a body containing the functional logic of the loop.

Following this concept, the trace collection would take a simple loop:

```
for (A) {
    B;
}
```

And produce the following output:

```
Trace File:
"functionality": {
    t: [{
        id: 0,
        a: [["&for_loop", 1]]
    },
    {
        id: 1,
        a: [["&cond", 2], ["&body", 3]]
    },
    {
        id: 2,
        a: [["R", A]]
    },
    {
        id: 3,
        a: [["W", B]]
    }
    ]
}
```

While the previous elements are always registered as *If*, loop traces need to describe the type of the Branching Element as more than just ”loop”. By describing it as ”for_loop” or ”while_loop” and so on through the other implementations of loop, the collector leaves information in the file that Mono2Micro

can later utilize to better approximate the loop. In the case of a ”do_loop” for example, the trace representation of the do while loop. The tool will interpret that the body always occurs before the first execution of the *condition*, which affects the sequence of accesses. Similarly to the if trace, and all the other traces for that matter, some child elements of the loop can be missing as well, and the tool is ready to infer on that information later.

An important property of Loops is that, unlike *If* statements, their execution can be affected by their content. Instructions such as *continue*, *break*, or even *return* can stop or cause an early repetition of the loop. To register this kind of event in the code, we introduce Labels. Labels are consist of a single string that describes a piece of information regarding that particular point the code. In the case of the instructions previously mentioned, a label is created with the name of the instruction and marking that it happened in that particular place in the Sequence. As explained Later, Labels are also used to identify specific properties important to calculate the probability of execution of each code branch. To capture these properties, we create a set of *PropertyScanner*s that is executed at the end of each recursive run of the scanner. These property scanners are stored in a dynamically loaded array during the creation of the collector. They are responsible for the creation of Labels in the Sequence in the situation that their specific conditions are met.

For a small example of these instructions, consider the input

```
for (A) {
    if (B) {
        break;
    }
    C;
}
```

The output would be as follows:

```
Trace File:
"functionality": {
    t: [{
        id: 0,
        a: [["&for_loop", 1]]
    },
    {
        id: 1,
        a: [["&cond", 2], ["&body", 3]]
```

```
10      },
11      {
12          id: 2,
13          a: [["R", A]]
14      },
15      {
16          id: 3,
17          a: [["&if", 1]]
18      },
19      {
20          id: 4,
21          a: [["&cond", 4], ["&then", 5]]
22      },
23      {
24          id: 5,
25          a: [["R", B]]
26      },
27      {
28          id: 6,
29          a: [["#break"]]
30      },
31      {
32          id: 7,
33          a: [["W", C]]
34      }
35      ]
36  }
```

A detail that has not been explained yet is the first character in *"&if"* or *"#break"*. This character is used to distinguish between context references, labels and accesses, since they all have string descriptors. The ampersand "&" is used to identify contexts, "#" for labels and accesses have no character, to match the previous trace format.

3.2.3 Function calls

The last of the three main Branching Elements is function calls. A function encapsulates a particular section of the code, allowing it to be reused in other parts of the program. By default, this does not make it crucial in control flow handling; however, using it along with branching elements, such as the previous statements *If* and *Loop* and the return instructions creates a lot of situations that deeply affect

the course of the program and, therefore, its trace.

```
"functionality": {
    t: [{
        id: 0,
        a: [["&call", 1]]
    },{
        id: 1,
        a: [["R", A], ["#return"]]
    }
    ]
}
```

The trace representation of a function call is not complex on its own. The content of a function is composed of its body. Therefore, to continue the representation style followed for the previous elements, a call element will point to a Sequence containing the representation of the function call's content.

3.2.4 Switch statements

Switch statements work in many ways similarly to the *If* statements. This element evaluates the value of an argument against a set of cases defined by the developer, choosing the execution route based on this value. However, it is more limited than an *If* statement, since the evaluated variable must belong to a limited group of supported types (such as primitive types char, byte, short, int, or classes like Character, Byte, Short, Integer, or an enum type). In the case of our trace representation, a switch case is represented by an array containing an expression, representing the selector variable that is evaluated, and a number of cases corresponding to all the options of the switch case.

```
"functionality": {
    t: [{
        id: 0,
        a: [["&sw", 1]]
    },{
        id: 1,
        a: [["&case", 2], ["&case", 3]]
    },
    {
        id: 2,
```

```
            a: [["R", A]]
        },
        {
            id: 3,
            a: [["R", B], ["W", B]]
        }
        ]
}
```

Switch statements are also affected by keywords such as *break*, which greatly affects the flow inside the component, as it is necessary to prevent the execution of consecutive cases. These instructions are registered in the trace using the previously described label elements.

3.2.5 Abstract functions calls

We describe abstract function calls as any call of a function defined as abstract in its original class. These functions are usually overridden in inheriting super classes, which is part of object-oriented programming concepts. However, this means that often we lack information at compile time to determine which of the implementations of the function is to be considered in that particular place in the program. The previous iteration of the trace collector worked around this by considering every possible implementation of the function as registering the access trace of each in sequence. We keep this approach, however, in line with the changes described here to better inform diverging paths in the code, we register each implementation's Sequence separately. An Abstract Call's Sequence is stored as an array of Options, each referencing the access Sequence contained on each of the function's implementations.

```
"functionality": {
    t: [{
        id: 0,
        a: [["&ac", 1]]
    },{
        id: 1,
        a: [["&op", 2], ["&op", 3]]
    },
    {
        id: 2,
        a: [["R", A], ["#return"]]
    },
    {
```

```
14          id: 3,
15          a: [["R", B], ["W", B]]
16      }
17      ]
18  }
```

3.3 Branch Heuristics

Many compilers utilize branch prediction to identify the portions of code that are executed more frequently or in sequence with each other. Due to this, many techniques for branch prediction have surfaced over the years. One of these techniques, described by Ball and Larus [25], proposes a program-based branch predictor that uses a set of heuristics to identify the branches that are most likely to be taken.

The heuristics proposed in [25] are restricted to two-way conditional branching with fixed targets, which disqualifies dynamic targets such as the ones accessed by lookup in jump tables. In those two branches, the heuristics consider a target successor, which is the one that is taken when the condition described in the heuristic is true, and the fall-through successor, which is taken otherwise. Some of the criteria also make use of the relations of domination and post-domination between sections of the code, based on the control flow graph. Considering each instruction as a vertex in the graph, a vertex v1 dominated v2 if every path that leads to v2 includes v1. On the other hand, a vertex v2 post-dominates vertex v1 if all paths leaving v1 include v2.

As they mostly target compiler optimizations, this kind of technique has a focus on lower-level programming languages. In this particular case, the heuristics proposed by Ball and Larus are designed for programs written in C and Fortran. Despite differing from the language targeted by Mono2Micro for microservice identification, Java, this heuristic model was picked for two reasons:

- As will be shown later, these heuristics were later used by Wu and Larus [1] to study the static branch frequency. In that study, we can find data relating to the probability of taking a branch based on the rules defined by Ball and Larus.
- A look at the current state of heuristic-based branch prediction finds that, despite the age of the paper, there have not been big developments in this field and these rules still largely match the state of the art
- There are no similar criteria or rules defined for the Java language

With that in mind, the rules defined are still slightly adapted in order to match Java's language specification, as things such as memory addressing and handling that are considered in some of the heuristics are wildly different when compared to a low level language like C.

The heuristics proposed by Ball and Larus are the following, organized by category:

- For branches that control loop execution:

  Loop Branch (LBH) The Loop Branch heuristic predicts that an edge that returns to the head of a loop will be taken, while an edge that leads to an exit of the loop will not be taken. This means that, while evaluating a loop condition, the most likely path to be taken is the one returning to the loop's head.

- For branch comparisons and successors in non-loop branches:

  Pointer (PH) Pointer comparisons usually compare either a pointer to null or another pointer. Assuming that in pointer manipulating programs most pointers are non-null, it can expected that an equality comparison of two pointers will rarely be true. In the case of Java, since every Object handling or comparison is actually performed with its pointer reference, we can adapt this heuristic to consider that any comparisons between two Objects or between an object and a null value will fail.

  Opcode (OH) Many programs use negative integers to denote error values. This leads to the Opcode Heuristic, which predicts that a comparison of an integer for less than zero, less than, or equal to zero, or equal to a constant will fail.

  Guard (GH) The Guard Heuristic differs from the previous heuristics as it bases itself not only on the content of the comparison, but also on the context around it. This heuristic defines that, when considering a comparison:

  – Where one of the operands is a register;

  – The register is used before being defined in the successor block;

  – And the successor block does not post-dominate

  We can assume that the successor block will be taken, so that the assignment of the register value can proceed.

  Loop exit (LEH) The Loop Exit heuristic resembles the Loop Branch heuristic, as they are products of a subdivision of Ball and Larus' Loop Branch heuristic performed by Wu and Larus. This division is used to better differentiate the two use cases of the original Loop Branch heuristic. In this case, the Loop Exit heuristic predicts that if a comparison in which neither the successor nor the fall-through branches is a loop head, then the result of the comparison will not exit the loop. This means that, in the previous situation, if either the successor or the fall-through had a break instruction in their content, it would not be taken.

- For successors:

Table 3.1: Branch frequency for each heuristic, as observed by Wu and Larus [1].

Heuristics	Probability of taking branch
Loop branch (LBH)	88%
Pointer (PH)	60%
Opcode (OH)	84%
Guard (GH)	62%
Loop exit (LEH)	80%
Loop Header (SH)	75%
Call (CH)	78%
Store (SH)	55%
Return (RH)	72%

Loop header (LHH) This heuristic assumes that a loop is often executed rather than avoided, and therefore a successor that is a loop header or a loop pre-header, and that does not post-dominate the choice will be taken.

Call (CH) Programs usually make calls to perform useful work. However, Ball and Larus noticed that, for their study set, most function calls that occur conditionally are used for error handling, whereas useful calls occur unconditionally. Based on this, this heuristic considers that if a successor contains a call and does not post-dominates, then it will not be taken.

Store (SH) The Store heuristic predicts that a successor that contains store instructions (or in Java's case, write memory accesses) and does not post-dominate will not be taken.

Return (RH) Similarly to the Call and Store heuristics, it is predicted that a successor containing a return and is not post-dominant will not be taken, as it represents an early end to the block.

Wu and Larus [1] later made use of these heuristics to statically infer the execution frequency of each branch. They evaluated the execution of a set of programs and compared the branch predicted by the heuristics with the one that was actually taken, considering the times they matched as hits and the percentage of predictions that hit as its hit rate. The hit rate results can be found in Table 3.1, and these are the data we will use to determine the probability of each branch in our model.

There are some situations in which multiple heuristics may apply to a single branch. The original paper [25] suggests considering only the first applicable heuristic in the priority array [PH, CH, OH, RH, SH, LHH, GH]. This can become an issue when, for example, heuristics apply to the same branch and may provide opposing predictions. To fight this, Wu and Larus however, proposed a solution based on the Dempster-Shafer theory [27] that can be used to combine the probabilities of all applicable heuristics.

Considering a set A of exhaustive and mutually exclusive outcomes of the situation (or, in our case, a branch). For two probability assignments m_1 and m_2, the Dempster-Shafer algorithm calculates a combined probability assignment $m_1 \oplus m_2$. Considering B a subset of A,

$$m_1 \oplus m_2(B) = \frac{\sum m_1(X) \times m_2(Y)}{\sum m_1(U) \times m_2(W)}$$

where X and Y are all subsets of A whose intersection is B, and U and W are the subsets of A with at least one element in common.

Considering the case where a branch b is predicted and has two possible successors, b_1 and b_2. In the absence of a heuristic, we consider that both branches are equally probable, and that is the starting stage of the calculation. Now, considering a heuristic, we predict that b_1 will be taken 70% of the time and b_2 will be taken 30% of the time. The combined probabilities are as follows.

$$m_1 \oplus m_2(b_1) = \frac{0.5 \times 0.7}{0.5 \times 0.7 + 0.5 \times (1 - 0.7)} = \frac{0.5 \times 0.7}{0.5 \times 0.7 + 0.5 \times 0.3} = 0.7$$

$$m_1 \oplus m_2(b_2) = \frac{0.5 \times 0.3}{0.5 \times 0.3 + 0.5 \times 0.7} = 0.3$$

This theorem is then repeatable for any number of heuristics that apply to branch b; Considering a second heuristic that predicts that the b_1 branch will be taken 60% of time and b_2 will be taken 40% of the time ($m_3(b_1) = 0.6$ and $m_3(b_2) = 0.4$). Then

$$m_1 \oplus m_2 \oplus m_3(b_1) = \frac{0.7 \times 0.6}{0.7 \times 0.6 + 0.3 \times 0.4} = 0.778$$

$$m_1 \oplus m_2 \oplus m_3(b_2) = \frac{0.3 \times 0.4}{0.7 \times 0.6 + 0.3 \times 0.4} = 0.222$$

Considering both heuristics, the estimate becomes that b_1 will be executed 77.8% of the time, while b_2 will only be executed 22.2% of the time.

3.4 Graph generation

It is not only on the collection side that changes are required to be made. The Mono2micro tool needs to be adapted in order to truly take advantage of the control flow information that is now being recorded into the trace.

Mono2micro deals with codebases in representation files. Each code base can be represented by a set of representation files, each describing the codebase in a different way. The current available representation types are the following:

Accesses Sequence or Sequence describes each functionality of the codebase as a sequence of accesses, it is the type of representation of the system we are improving;

IdToEntity lists every functionality id and which entity of the system them represent;

Author is a type of representation that contains information relative to who authored each contribution to the code;

Commit holds information from the version control system of the codebase, grouping the changes done to the code in each commit;

EntityToId reverse of IdToEntity, relates each entity to its id

Code Embeddings represents the code in terms of code embeddings done by a machine learning algorithm.

These representations are used by Strategies. A Strategy defines the way a set of Representations is used in order to calculate the similarity measures between the monolith domain entities. Each Strategy may require multiple representations. The Repository Strategy, for example, requires the IdToEntity, Accesses Sequence, Author and Commit representations, while the Accesses Sequence similarity requires only IdToEntity and Sequence.

The new trace collection method, as explained in this chapter, is an upgrade made upon the previous access sequence collection. This means the new trace still describes the order in which the entity accesses of each functionality, represented by a pair access mode (read or write) and the entity's id, but expresses it in the form of a graph instead of a linear sequence. This means that, similarly, the Strategy that takes advantage of the new trace type will also retain multiple similarities with the current Sequence Strategy.

With this in mind, the Accesses Graph Strategy is introduced. While the Sequence Strategy required the IdToEntity and Sequence representations, the Graph Strategy instead requires the IdToEntity and Graph representations, with the latter following the aforementioned improved trace format. To maintain the maximum amount of common functionality with the Sequence Strategy, this new strategy will follow the same interface. Sequence utilizes a trace iterator to traverse and interpret the trace file, and therefore, a similar iterator is used in the Graph Strategy to translate and handle its json file.

This iterator works in two phases. The first phase interprets the translation of the json representation into a network of *CommandGraphs* with each of them being a 1-to-1 representation of a json element. *CommandGraphs* are divided in two different main subtypes, *CompositeCommandGraph* (*IfGraph*, *LoopGraph*, *SwitchGraph*, *CallGraph*, *AbstractCallGraph*), which are elements that can contain other subgraphs, and *AtomicCommandGraphs* (*AccessNode*, *LabelNode*). During the second phase, the network of *CommandGraphs* is translated into a graph with a single node type, *Access*. During this phase, the probabilities of each branch are also calculated using branch heuristics. Figure 3.1 displays the solution domain model, containing the entities described.

We are now going into detail about the method of creation of each *CommandGraph* and their processing.

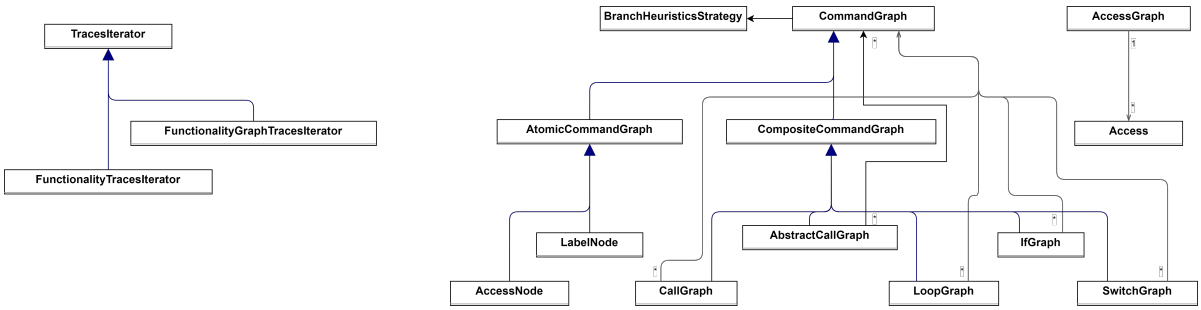


Figure 3.1: Domain model of Mono2micro's logic responsible for graph processing

3.4.1 If (*IfGraph*)

As described in the previous section, an *If* is considered to be composed of three possible elements: a *condition*, a *then* block, and an *else* block. Each of these elements corresponds to a sub-trace, of the accesses and other program logic that happens within them. Since the processing is performed recursively, the first step is to gather the full access sub-graph of each of these components. These sub-graphs are handled inside of a TraceGraph class. Each TraceGraph is defined by:

- A graph of accesses, composed only of Access instances connected by weighted edges using the jgrapht [28] framework. Each weighted edge equates to the floating value between 0 and 1 corresponding to the probability of an Access occurring after the previous one.
- The first and the last accesses in the graph, which need to be tracked in order to connect to elements in higher levels of the recursion
- The set of vertices that, in lower recursion levels, have been locked to further connections. This ensures that Accesses preceding a Return, for example, will not connect to any succeeding accesses, as even though the information of the Return will be nonexistent in the graph (only represented by the path it generates, as will be seen further) it needs to be prevented from connecting with other succeeding accesses.

In most situations *IfGraph* will have three sub graphs, one for each of the components of an If, however, there may be situations where 1) a component may be missing (eg: when an If only has a then block, but no else block, as it is also common to occur) or 2) the component is defined in the code but no entity accesses happen inside of it. In either situation, the translator simply ignores the missing component. In Figure 3.2 some of the possible combinations of these components are explained, with the weight between each node representing the probability of the successor. In the base case, where all the elements are present, the condition will be connected to both the then and the else successors. The probability of the branch to Then being taken is named P_{enter} and is calculated based on the heuristics

that apply to all three components. Therefore, it takes into account the probability of condition being true, the Then successor being taken and the Else successor not being taken. The probability of the Else successor being taken is the opposite, and so its value equals $1 - P_{enter}$. The sum of probabilities of leaving a node is always equal to 1 or 100%. Finally, since both the *then* and *else* components lead to the eventual exit from the If, they both connect to the Ending Node, with a probability of 1 each.

In another situation, the *condition* component may be missing. In that case, the Starting Node will connect to the *then* and *else* successors with the same probability as the *condition* would have. Both the *Starting Node* and the *Ending Node* exist with the goal of filling gaps created by missing child elements. They ensure that each sub-graph will always a single entry point, the *Starting Node*, and a single exit point, the *Ending Node*, and therefore making it easier to recursively attach the graphs from the inside and that no sub graph is disconnected from the main graph either that the start or the end.

Lastly, another possible situation is that one or more of the successor components is missing. In this case, the *condition* will connect to the *Then* with the same probability as before; however the connection with $1 - P_{enter}$ probability will instead be made directly to the *Ending Node*. This ensures that not only the sum of the edges departing from Condition remains 1, but also that the combined probability of all paths converging in the Ending Node is $\leq$1. For this case, two paths can be observed starting from *condition* and arriving at *Ending Node*, P_1 and P_2. P_1 is comprised of 2 branches, *Condition-Then* and *Then-Ending Node*, and therefore the probability of P_1 being taken from *condition* to *Ending Node* is $P_{P_1} = P_{enter} \times 1 = P_{enter}$. P_2, on the other hand, is composed solely by the branch *Condition-Ending Node*, whose probability is $P_{P_2} = 1 - P_{enter}$. This makes the combined probability of all branches converging on the Ending Node equal to $P_{P_1} + P_{P_2} = (P_{enter}) + (1 - P_{enter}) = 1 \leq 1$.

3.4.2 Loops (*LoopGraph*)

The structure of a *Loop* entity is composed of two components, an *Expression* which controls the loop execution in the *Loop*, and a *Body* which contains the main functionality of the loop. In its current implementation, only the structure of a loop with an *Expression* followed by a *Body* block is considered; however, the trace possesses information to consider other loop types (such as the *Do-While Loop*) in future improvements.

A *Loop* represents a challenge when performing static analysis, as it can be executed any number of times, including none. It is impossible to statically predict how many times a loop will execute, as it often depends on dynamic information that varies and is only available at runtime. With this in mind, when considering the evaluation of a sequence of accesses, we consider the two possible situations after the *Expression* is evaluated: the loop is not taken/entered or the loop is entered. When the loop is not taken, the *Expression* is evaluated only once, and then the flow proceeds to what follows the loop. When the loop is taken, however, the expression is evaluated once and evaluates to true, followed

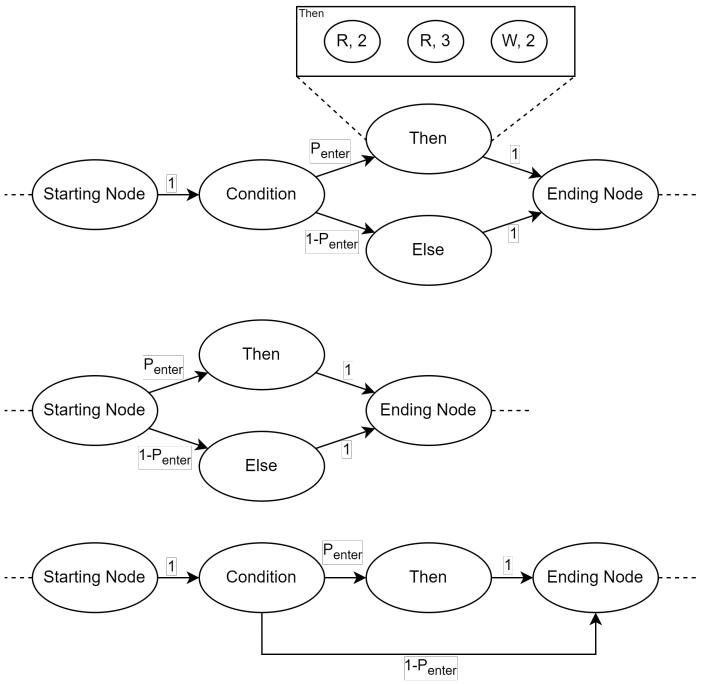


Figure 3.2: Sample of possible combination of If components

by the execution of the *Body* block and then the *Expression* is evaluated again. This means that in a single loop execution we consider the sub-sequences *Expression-Body* and *Body-Expression*. If the *Expression* were to evaluate as true once again, the next sub-sequence to occur would be *Sequence-Body*, which was already considered during the first loop.

As such we can conclude that by considering the situations ”Loop is not taken” and ”Loop is taken and executed only once” we are already taking into account the majority of all possible sub-sequences that a Loop entity can produce.

With this in mind, the program contemplates the three cases described in Figure 3.3, similarly to *If*: the *Loop* has an *Expression* and a *Body*; *Loop* has no *Expression* and a *Body*; and *Loop* has no *Expression* and no *Body*.

Similar to *If*, the *Starting Node* is connected to the *Expression* with probability 1. Then, two paths depart from there: one for entering the loop, and one for avoiding it. The first path will first connect *Expression* to *Body*, and the probability is that of entering the loop, P_{enter}. Creating cycles in the graph is something we intended to avoid. Cycles would only increase the complexity of the graph, and require additional logic to traverse the graph and ensure each node is considered the correct amount of times. Therefore, instead of now connecting *Body* to *Expression* and making that edge bidirectional, we will instead connect *Body* to a copy of the *Expression* graph. With this method we ensure the content of the paths remains the same with lower required complexity, at the cost of repeating a portion of the graph

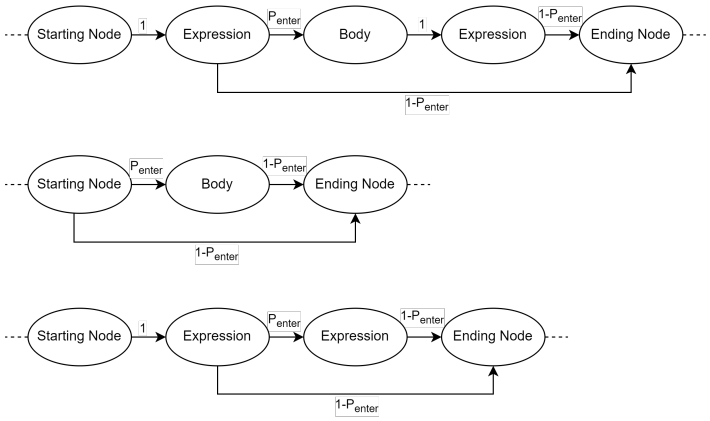


Figure 3.3: Sample of possible combination of Loop components

and slightly increasing its memory usage. Lastly, this copy of *Expression* is connected to *Ending Node*, with the probability of the *Expression* evaluating to false - which is $1 - P_{enter}$. The second path is the one avoiding the loop, so it will connect the first *Expression* directly to *Ending Node* with $1 - P_{enter}$ probability.

The second case of Figure 3.3 considers a situation where the *Loop* has no *Expression* but still has a *Body*. In this case the same logic is followed, however with no *Expression* most of the graph will be simplified. For the loop-entering path, *Starting Node* connects to *Body* with P_{enter}, which then connects to *Ending Node* with $1 - P_{enter}$ probability - as it is the probability of *Ending Node* being reached from *Body* in the base case. For the loop-avoiding path, *Starting Node* simply connects to *Ending Node* with $1 - P_{enter}$ probability, as that is the probability of the loop content being avoided entirely.

The third and last case considers the *Loop* in question has an *Expression* but no *Body*. In this situation, both the base *Expression* and its copy will connect with *Starting Node* and *Ending Node* like in the first case. However, with no *Body* they will instead connect to each other directly, with probability 1.

3.4.3 Call (*CallGraph*)

The translation of a *Call* elements is simpler than the previous components as it only has a single component, a *Body* (Figure 3.4,). The *Starting Node* is connected to the function's *Body* graph which then connects to *Ending Node*, all with probability 1. There is a situation where the *Body* component can be missing, as then the function call will simply not be considered in the graph. The real complexity that is brought about by the use of *return* instructions, which is covered in 3.4.4.

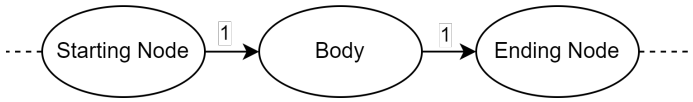


Figure 3.4: Graph result of Call

3.4.4 Label (*LabelNode*

In the previous section we introduced the concept of *Labels*, strings included in the json representation of the trace, to either mark the position of control instructions, such as *return*, or to hold information important to the calculations of the branch heuristics. When translating the json file, these elements will have a specific class responsible for their behaviour, similarly entitled *Label*.

In terms of control instructions, we currently consider three instructions: *return*, *break* and *continue*. All three of them follow the same behavior of redirecting the flow of the program to a specific point:

- *Return* redirects to the end of the current call context;
- *Break*, similarly, redirects the flow not to the end of the call but to outside the current loop context, bypassing any *Expression* logic;
- *Continue* finishes the current iteration of the *Loop* and redirects the flow to the start of the next execution of the *Expression*.

The points where the flow is directed can correspond to nodes of the access graph: the end of the current call is the *Ending Node* of the last call to be processed in the recursive stack; the outside of the current loop is the *Ending Node* of the last loop to be processed; and the start of the next *Expression* is the first node of *Expression Copy*. When a *LabelNode* is being processed, it does not generate a node for the final graph, even though it is an atomic element. Instead, it receives the references to the previously mentioned Nodes and connects the last Access of the graph created so far to the respective one, completing the jump equivalent to the return, break or continue it represents.

The information regarding these jumps is kept only on the *LabelNode*, therefore disappearing once this element is gone. In order to prevent accesses from other places from accidentally connecting to an access involved in a return, for example, the latter is ”locked”. This is achieved by using an array at the level of the graph, storing all of said graph’s nodes that have been locked.

On the other hand, the other types of labels, the ones responsible for providing information essential to branch heuristic calculation, include *zero_comparison*(used to signal a comparison that fits the Op Code Heuristic), *object_comparison* (which marks comparisons between objects for the Pointer Heuristic) and *later_changed_c_variable* (which informs that one of the previous condition’s variables is changed in the current branch). The detection of these and the previous label types will activate a flag signaling

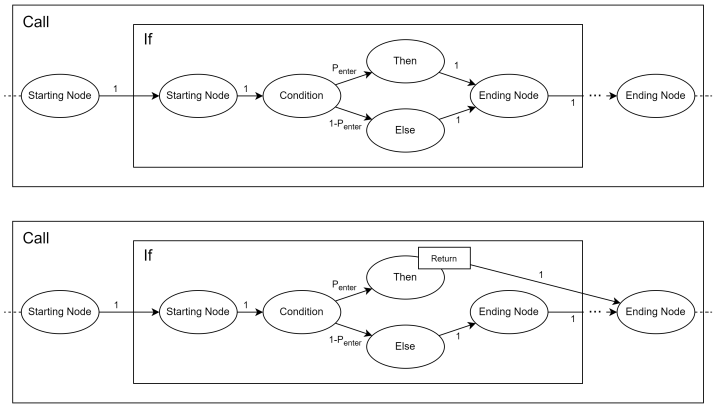


Figure 3.5: Comparison between the connection of a Call without return, and another with

their particular property, which will make it so that it is taken into account during the branch heuristic calculations.

3.4.5 Abstract Call (*AbstractCallGraph*)

As mentioned above, Abstract Call works in many ways as a normal function call. The main difference being that an Abstract Call has multiple overrides of the main function's body and it is close to impossible to predict at compile time which of the overrides will be executed. Because of this, each of them has to be treated as having the same chance of occurring. In order to achieve this the *Starting Node* is connected the sub graph of each of the possible *Override Options*. The base weight of this connection will be equal to $1/\#options$. Although, there may be cases where an override exists but no accesses are performed inside of it (eg: only has labels). In other situations, these cases would be disregarded as they do not affect the overall graph. In this situation, this should still mean a reduction of the probability of the other override options occurring. To achieve this, the probability for all the empty override options is collected and assigned to a connection directly from *Starting Node* to *Ending Node*. The override options, are always connected directly to the *Ending Node* with probability 1, as nothing can prevent that branch from happening. Both these cases can be observed in Figure 3.6.

3.4.6 Switch (*SwitchGraph*)

The *Switch* combines the *If* with *Abstract Call*. It holds an *Expression* that will be evaluated, followed by multiple *Cases*, one of which will be executed depending on the result of the evaluation. Each of the *Cases* has a different chance of occurring, similar to the *If* element, based on the *Expression* and their own content. Heuristics are possible to be adapted and applied to *Switch*, however for this Strategy implementation this was not done. Instead, it works in the same way as an *Abstract Call* with the *Starting*

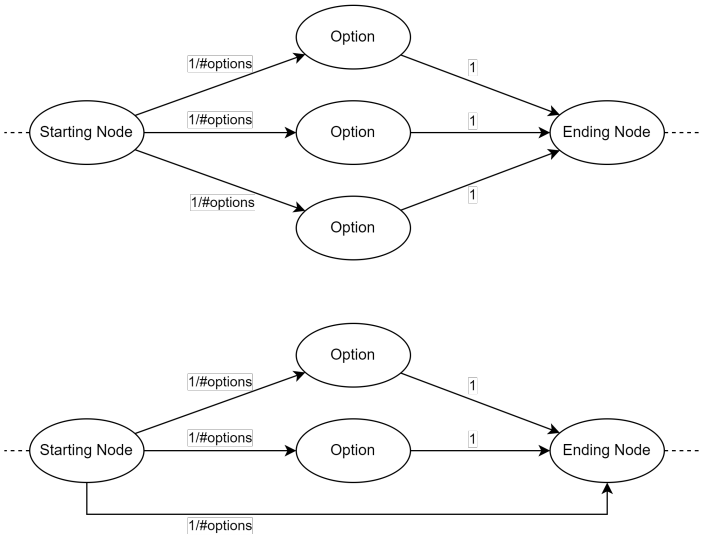


Figure 3.6: Graph result of an Abstract Call

Node connecting to *Expression* with probability one, which will then connect to each of the *Cases* with $1/\#cases$ probability. Similarly, the *Cases* connect to *Ending Node* with probability 1, and any *Case* with no accesses will have its probability summed in a direct branch between *Starting Node* and *Ending Node*.

3.4.7 Access (*AccessNode*)

The Access element contains the basic level of information of the graph. It describes only an access mode (read or write) and the id of the entity accessed, and both are kept in the *AccessNode*. During processing, it generates an *Access* element. It is the only *CommandGraph* that generates a node in the final graph, as the others will only provide information for the connections between these nodes and the probabilities of each path.

3.4.8 Calculating the Branch Probabilities - Heuristics

The translation from the temporary classes to the Access graph is handled in a recursive manner. Whenever a Branch Element is translated, it first computes the access graphs of each of its components. This means that each Branch Element will have all the necessary information in order to assign the correct probabilities to each of the successors. This comes in the form of an *HeuristicsFlags* object containing flags for each of the possible properties that affect the probability of branches. These are filled when processing *Labels* (as was seen before) or when processing specific elements that have properties related to them (*Call*, *Loops* and Write Accesses, for example). These property flags are combined in order to determine which Branch Heuristics apply to the specific situation, which are stored by name in

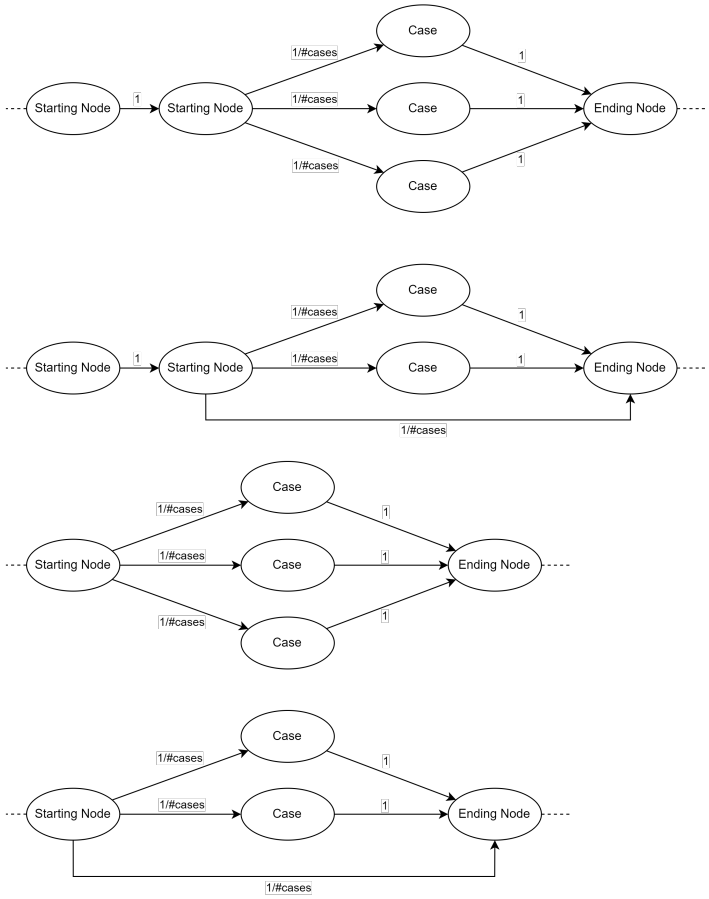


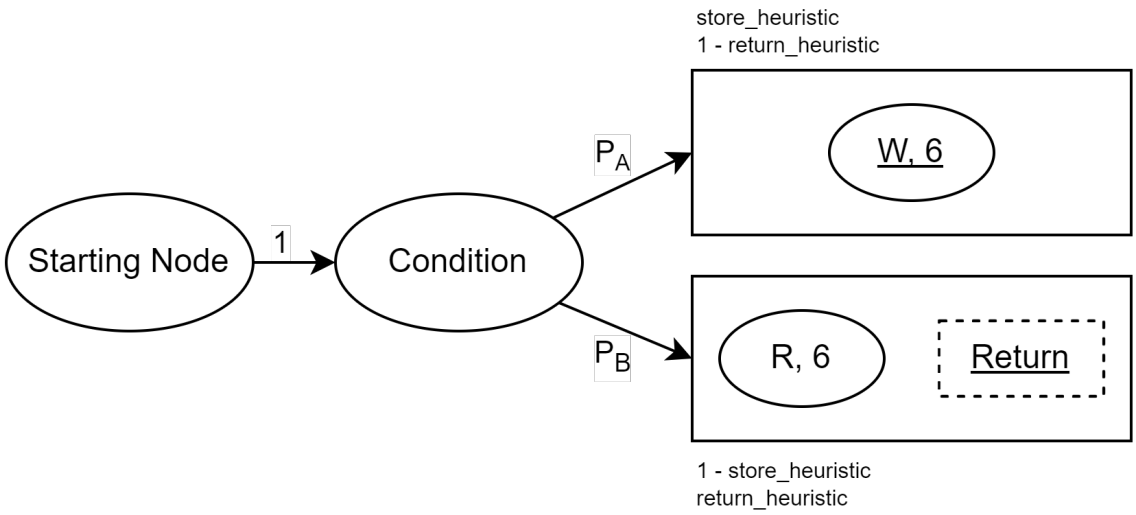
Figure 3.7: Graph result of a Switch

a string array. When considering a branch, the probability of each successor will take into account both the heuristics that apply to it and the heuristics that apply to the other successor. Considering a branch that has only two possible successors, the heuristics that apply to one successor will affect its probability in a direct way, meaning that they will indirectly affect the probability of the other successor (as one successor's probability increasing/decreasing will mean the opposite change for the other). To take this into account, the list of applicable heuristics for a successor contains the Branch Heuristics identified for that successor, along with the sum of 1 minus each of the probabilities of the opposite successor.

An example of this process is described in Figure 3.8. In this example, we consider two possible successors to a condition, successors A and B. Successor A contains a store operation and therefore the Store Heuristic will apply. However, since successor B, the branch opposite to A in the decision, contains a return instruction and the Return Heuristic will be applied to it, then 1 minus the probability of the Return heuristic should be applied to successor A. This means that the array containing all the heuristics applicable to successor A will be $[\text{"}store\_heuristic\text{"}, \text{"}1 - return\_heuristic\text{"}]$. The contrary will happen to successor B which, being opposite to A in the decision, will have applied the heuristics $[\text{"}1 - store\_heuristic\text{"}, \text{"}return\_heuristic\text{"}]$.

After identifying the applicable heuristics, we are ready to apply the Dempster-Shafer theorem [27] to calculate the probability of that successor, using the following code:

```
1  public static float calculateBranchProbability(float initialValue, List<String>
       heuristics) {
2          // calculate probability based on the Dempster-Shafer theory
3          // x*y/(x*y + (1-x)*(1-y))
4
5          float result = initialValue; // initial probability of each branch is 0.5
               ; 0.5 is also neutral in the operation
6          for (String h : heuristics) {
7              float currentHeuristicProb;
8              if (heuristicProbabilities.containsKey(h)) {
9                  currentHeuristicProb = heuristicProbabilities.get(h);
10             } else if (heuristicProbabilities.containsKey(h.split("_i")[0])) {
11                 currentHeuristicProb = 1 - heuristicProbabilities.get(h.split("_i
                       ")[0]);
12             } else {
13                 currentHeuristicProb = 0;
14             }
15
```



Heuristic	Probability of Branch being taken
Store Heuristic	55%
Return Heuristic	72%
1 - Store Heuristic	100% - 55% = 45%
1 - Return Heuristic	100% - 72% = 28%

Applicable heuristics - A:
store_heuristic
1 - return_heuristic

$$P_A = \frac{0.55 \times 0.28}{0.55 \times 0.28 + 0.45 \times 0.72} = 0.322$$

Applicable heuristics - B:
1 - store_heuristic
return_heuristic

$$P_B = \frac{0.45 \times 0.72}{0.55 \times 0.28 + 0.45 \times 0.72} = 0.678$$

Figure 3.8: Example of heuristics applicable in a branch decision.

```
16              result = result*currentHeuristicProb/(result*currentHeuristicProb +
                    (1-result)*(1-currentHeuristicProb));
17
18          }
19
20          return result;
21
22      }
```

The result is a value between 0 and 1, which represents the probability of that successor being taken from the previous node. As each of the successors will take into account its heuristics and the inverse of the heuristics of the other possible successors, the sum of the probabilities leaving their parent node will always be less than or equal to 1.

3.5 Similarity measure calculation

In Chapter 2 we described the similarity measures [4] used during the generation of decompositions to group the domain entities of the system in Clusters in order to minimize the number of distributed transactions per functionality. These similarity measures (Access, Read, Write and Sequence), were therefore calculated through the following formulas:

$$sm_{access}(e_1, e_2) = \frac{\#(funct(e_1) \cap funct(e_2))}{\#funct(e_1)}$$

$$sm_{read}(e_1, e_2) = \frac{\#(funct(e_1, r) \cap funct(e_2, r))}{\#funct(e_1, r)}$$

$$sm_{write}(e_1, e_2) = \frac{\#(funct(e_1, w) \cap funct(e_2, w))}{\#funct(e_1, w)}$$

$$sm_{sequence}(e_1, e_2) = \frac{sumPairs(e_1, e_2)}{maxPairs}$$

where $funct(e, m), m \in r, w$ defines the number of functionalities that access the domain entity e in read(r) or write(w) mode and $sumPairs(e_1, e_2) = \sum_{f \in F} \#\{(a_i, a_j) \in G_f.P : (a_i.e = e_1 \wedge a_j.e = e_2) \vee (a_i.e = e_2 \wedge a_j.e = e_1)\}$ is the number of consecutive accesses of e_1 and e_2, regardless of the order, with $G_f.P$ being the precedence relation for functionality f, and $maxPairs = max_{e_i, e_j \in E}(sumPairs(e_i, e_j))$ is the maximum number of consecutive accesses between any two entities in the monolith.

When taking into account a basic branching segment of code, such as

```
1 if (A) {
2     B;
3 } else {
4     C;
5 }
```

where the condition has 70% probability of being evaluated true, the access pair counts would be the following, applying the former sequences of accesses:

$$access(e_A, e_B) = access(e_B, e_A) = 1$$

$$access(e_B, e_C) = access(e_C, e_B) = 1$$

Immediately, the research problems become apparent. First, we notice $access(e_B, e_C) = 1$, which is impossible, as they cannot be accessed on the same execution. Second, there is no access pair $access(e_A, e_C)$, even though we know this is a possible sequence. Third, when calculating the Access Similarity, for example, between entities A and B and considering only a single functionality, we would learn that

$$sm_{access}(e_A, e_B) = \frac{\#(funct(e_A) \cap funct(e_B))}{\#funct(e_A)} = \frac{1}{1} = 1$$

We know that this is wrong because $sm_{access}(e_A, e_B) = 1$ would mean *A* and *B* have 100% probability of being accessed together in this functionality, and due to the nature of the *If* that is false. Therefore in order to consider the branching information changes are required to be made to the formulas.

Simply by introducing the Graph representation of the code, the access pair counts would look like this:

$$access(e_A, e_B) = access(e_B, e_A) = 1$$

$$access(e_A, e_C) = access(e_C, e_A) = 1$$

Not only is $access(e_A, e_C)$ no longer missing, but also since the accesses *B* and *C* are no longer connected in the representation $access(e_B, e_C)$ no longer exists. The only problem that remains is that the pairs *A*-*B* and *A*-*C* both have a similarity value of 100%, meaning they are certain to be accessed in the same run, which is impossible.

To fix this, we propose the following formula for Access Similarity

$$sm_{access}(e_1, e_2) = \frac{Prob(funct(e_1) \cap funct(e_2))}{Prob(funct(e_1))}$$

where $Prob(funct(e_1))$ describes the probability that a functionality accesses entity e_1 and $Prob(funct(e_1) \cap funct(e_2)(e_1))$ the probability of a functionality accessing e_1 and e_2 in sequence. Since $funct(e_1) \cap funct(e_2)$ is a subset of $funct(e_1)$ then we are assured that $Prob(funct(e_1) \cap funct(e_2)) \leq Prob(funct(e_1))$, meaning the value for the Access Similarity between entities e_1 and e_2 will range between 0 and 1.

This method eliminates the concept of access pair counts, since a count assigns the weight of 1 to every access. Instead, each access will have a different weight, based on the probability of its occurrence. If we consider the previous code excerpt as an example, we have the following.

$$access(e_A, e_B) = access(e_B, e_A) = 0.7$$

$$access(e_A, e_C) = access(e_C, e_A) = 0.3$$

and therefore

$$sm_{access}(e_A, e_B) = \frac{Prob(funct(e_A) \cap funct(e_B))}{Prob(funct(e_A))} = \frac{0.7}{1} = 0.7$$

giving the Access Similarity between entities *A* and *B* equal to 0.7.

With the same goal, we also propose new formulas for the remaining functionalities:

$$sm_{read}(e_1, e_2) = \frac{Prob(funct(e_1, r) \cap funct(e_2, r))}{Prob(funct(e_1, r))}$$

$$sm_{write}(e_1, e_2) = \frac{Prob(funct(e_1, w) \cap funct(e_2, w))}{Prob(funct(e_1, w))}$$

$$sm_{sequence}(e_1, e_2) = \frac{sumProbs(e_1, e_2)}{maxPairProb}$$

$sumProbs(e_1, e_2) = \sum_{f \in F} \#\{(a_i, a_j) \in G_f.P : (a_i.e = e_1 \wedge a_j.e = e_2) \vee (a_i.e = e_2 \wedge a_j.e = e_1)\}$ is the summed probability of consecutive accesses of e_1 and e_2, no matter the order, with $G_f.P$ being the precedence relation for functionality f, and $maxPairProb = max_{e_i, e_j \in E}(sumProbs(e_i, e_j))$ is the maximum probability of any two entities being accessed consecutively.

Another advantage of the improved formulas is that they maintain the functionality of the Sequence Strategy. Note that the new formulas are an extension of the previous ones. When considering the case of the Sequence Strategy, where the weight of each pair is 1 (or 100%), and applying to the formulas, we will notice that

$$\frac{Prob(funct(e_1) \cap funct(e_2))}{Prob(funct(e_1))} = \frac{\#(funct(e_1) \cap funct(e_2))}{\#funct(e_1)}$$

$$\frac{sumProbs(e_1, e_2)}{maxPairProb} = \frac{sumPairs(e_1, e_2)}{maxPairs}$$

meaning that the new formulas are compatible with both Strategies and can be used interchangeably.

4

Evaluation

Contents

4.1 Evaluation Methodology 46
4.2 Comparison of best candidate decompositions by metric 49
4.3 MOJO comparison 50
4.4 Statistical Analysis 50
4.5 Graph vs Sequence vs Source of Truth 51
4.6 Discussion 60
4.7 Threats to Validity 60

In this section, we evaluate whether the use of the Graph Strategy translates into the generation of better quality decompositions.

The evaluation data is extracted by processing the Quizzes Tutor[1] monolith through Mono2Micro. Quizzes Tutor is a questionnaire platform that has been in development since February 2019. The platform allows teachers to share and reuse software engineering questions in the creation of quizzes, and also gives students the ability to self-generate quizzes to practice using a pool of available questions.

[1] https://github.com/socialsoftware/quizzes-tutor

It has 112.473 lines of code, authored by 25 contributors and divided between frontend and backend components, with 46.522 being attributed to the backend over which our evaluation is going to focus.

The backend of Quizzes-Tutor is based on the Java Spring-boot framework and utilizes Hibernate JPA to access persistent domain context entities. It counts 108 functionalities, 18 controllers, and 46 domain entities, spanning from authentication to questions.

The application is a monolith implemented in Spring Boot, making it a good target for the Mono2Micro tool. Although it is designed as a monolith, the authors have developed a modular version of the system[2] that is being used as a source of truth for what the microservice decomposition of the program should look like. This decomposition is composed of the following 10 modules, with their respective entities:

- Answer - (12 entities) AnswerDetails, CodeFillInAnswer, CodeFillInAnswerItem, CodeOrderAnswer, CodeOrderAnswerItem, CodeOrderAnswerSlot, MultipleChoiceAnswer, MultipleChoiceAnswerItem, QuestionAnswer, QuestionAnswerItem, QuizAnswer, QuizAnswerItem
- Auth (4 entities) AuthDemoUser, AuthExternalUser, AuthTecnicoUser, AuthUser
- Dashboard - (4 entities) Dashboard, DifficultQuestion, FailedAnswer, WeeklyScore
- Discussion - (2 entities) Discussion, Reply
- Execution - (3 entities) Assessment, CourseExecution, TopicConjunction
- Question - (12 entities) CodeFillInOption, CodeFillInQuestion, CodeFillInSpot, CodeOrderQuestion, CodeOrderSlot, Course, Image, MultipleChoiceQuestion, Option, Question, QuestionDetails, Topic
- Questionsubmission - (2 entities) QuestionSubmission, Review
- Quiz - (2 entities) Quiz, QuizQuestion
- Tournament - (1 entity) Tournament
- User - (4 entities) DemoAdmin, Student, Teacher, User

4.1 Evaluation Methodology

The evaluation is built on the steps of the pipeline. Both the Collection and Decomposition steps apply to the system under analysis, once using Graph Strategy and again using Sequence Strategy. These steps will produce several dendrograms, each generated by varying the weights of the similarity measures, Access (A), Write (W), Read (R), and Sequence (S), in intervals of 10 on a scale from 0 to 100, with the

[2] https://github.com/socialsoftware/quizzes-tutor/tree/modular

sum of the weights totaling 100. Each dendrogram results in multiple candidate decompositions for the system with a varying number of clusters. For the sake of comparison, for most data, we only consider candidate decompositions that have the same number of clusters as the expert decomposition, which is 10. For each candidate decomposition, Mono2Micro calculates the values for complexity, cohesion, and coupling, which we use in the evaluation process.

The evaluation data is extracted by processing the Quizzes Tutor monolith through Mono2Micro. Starts with the creation of the traces in the Collection step, where each strategy has its own collector that produces a JSON representation in the respective format. That is followed by the Decomposition Generation step where, similarly, each Strategy has its own code and handles their logic differently and uses it to calculate the similarity measures, as described in the previous chapter. However, the following stage, Quality Assessment and Comparison, is different.

The metrics are calculated per functionality based on the cost of each local transaction within that functionality. Local transactions are considered based on a graph of local transactions (GLT) derived from the trace. Due to time constraints, it was not possible to adapt and properly test the creation of the GLT based on Graph traces, and therefore the metrics are always calculated based on the Sequence representation of the codebase. The remaining steps, Visualization, Editing, and Modeling, work the same regardless of the strategy applied, meaning they are not based on either specific representation.

The data used to compare the results of each strategy is gathered during the Quality Assessment stage and is complemented by manual close inspection of the decompositions.

To objectively compare the qualities of each strategy, not just against each other but also generally, we built a Source of Truth for the Quizzes Tutor system. This Source of Truth or Expert Decomposition is a decomposition of the system designed by its author. In the case of Quizzes Tutor, the authors have developed a modular version of the system[3] which is being used as a reference. Therefore, our method was to map each module to a microservice.

4.1.1 MOJO

Evaluation based on their metrics only allows comparing the entities and functionalities identified by each Strategy. To evaluate according to the ones that may not have been but should have, a measure like MoJoFM [3] is necessary. MoJoFM is a distance measure between two architectures of the same system, expressed as a percentage. The key operations in transforming a system architecture into another, in our case expressed as transforming one candidate decomposition into another, are moves of entities between clusters (Move), and merges of clusters (Join). Given two decompositions, A and B, MoJoFM can be defined as:

[3] https://github.com/socialsoftware/quizzes-tutor/tree/modular

$$MoJoFM(A,B) = (1 - \frac{mno(A,B)}{max(mno(\forall A,B))}) \times 100\%$$

with $mno(A,B)$ being the minimum number of Move and Join operations necessary to transform decomposition A into B, and $max(mno(\forall A,B))$ the highest amount of minimum operations needed to transform any decomposition into B (or by other words, the amount of operations needed to transform the decomposition most different from B into B). A 100% MoJoFM value means that both decompositions are exactly the same, while 0% means they are completely different. This means MoJoFM can be used as a way to compare the candidates generated by each of the strategies against eachother, to determine the difference between the approaches. It also means each decomposition can be compared against the Expert decomposition, to determine which of the strategies offers a better accuracy to the vision of the architect.

4.1.2 Statistical Analysis

Similarly, we can analyze the quality of the techniques through the lens of pattern recognition software. Considering the expert decomposition as the source of truth, asserting how any two given entities e1 and e2 are positioned in the same cluster when the expert had them in the same cluster (true positives), are not positioned in the same cluster even though the expert has them in the same cluster (false negatives), has them in the same cluster even though the expert has them in different clusters (false positives) and how many times e1 and e2 are not positioned in the same clusters in both the candidate and the expert (true negative). Through this, we can calculate:

- Accuracy - percentage of times the decomposition was right, according to the source of truth (true positives and true negatives)
- Precision - percentage of times the decomposition placed an entity correctly (true positives out of all true and false positives)
- Recall - ability of the tool of finding the relevant cases (true positives divided by true positives plus false negatives)
- Specificity - ability of the tool of finding the relevant negative cases (similarly to recall, true negatives divided by true negatives plus false positives)
- F-score - measure of the predictive performance, based on the precision and recall values

Circling back to our research questions, at the beginning we wanted to answer the following:

- RQ1: Does control flow information affect the quality of microservice decompositions generated through static analysis?

- RQ2: How do candidate decompositions generated using static analysis, with and without control information, compare to an expert decomposition?

To answer RQ1, we first evaluate the best decompositions, generated using the new Graph Strategy (empowered with control flow information) for each of the metrics (complexity, cohesion, and coupling) versus against the previous Sequence Strategy. As for answering RQ2, we introduce the Expert Decomposition into the comparison, and extend the process to take into account other metrics between the generated decompositions.

4.2 Comparison of best candidate decompositions by metric

The first step in evaluating whether one approach provided better decompositions than the other was to gather the best candidate decompositions generated by each strategy, according to each of the metrics.

As seen in the results found in 4.1, the sequence strategy generates the best decompositions for each of the three metrics. The best decompositions generated through the graph strategy consistently have 2 to 3 times higher complexity and 1.5 times higher coupling and lower cohesion (5 to 10% lower) than those offered by the Sequence Strategy, which is far from ideal. This suggests that Sequence Strategy might be more suitable for microservice decomposition, despite relying on incomplete information; nevertheless, additional data will show that this assumption may not hold true.

Table 4.1 also provides the values for the metrics of the expert decomposition, which surprisingly displayed the worst metric results. On the other hand, these results may mean that Graph Strategy provides metric values for its decompositions closer to the source of truth than Sequence Strategy due to the richer collected data. This data leads us to raise a couple questions, such as whether the Complexity, Cohesion and Coupling metrics are ideal for evaluating microservice decomposition?.

Note that the complexity measure correlates with the number of distributed transactions. Actually, the monolith has complexity 0, because all transactions are ACID, therefore, this metric does not provide an absolute measure for the quality of a decomposition. The same applies to the coupling measure. Therefore, the evaluation of the best decomposition should take into account not only the quantitative aspects but also a qualitative analysis.

Since the analysis of the decompositions using quantitative analysis through metrics has been investigated [4], we intend to experiment with another focus: *What semantically characterizes a decomposition that minimizes some of the metrics when compared with an expert decomposition?*

But answering to this question we also intend to identify new challenges in the research on the automatic identification of microservices in monolith systems.

Table 4.1: Comparing the best metric values achieved with each strategy and the source of truth

	Complexity	Cohesion	Coupling
Expert	**611**	**0.565**	**0.353**
Sequence (lowest complexity)	145	0.780	0.131
Graph (lowest complexity)	225	0.703	0.185
Sequence (highest cohesion)	184	0.810	0.144
Graph (highest cohesion)	318	0.714	0.203
Sequence (lowest coupling)	145	0.780	0.131
Graph (lowest coupling)	300	0.681	0.183

Table 4.2: MoJoFM distance comparing both strategies and the expert decomposition. Each column stipulates the decompositions that are used in the comparison (eg: the ones with the best complexity of each strategy).

	Best Complexity	Best Cohesion	Best Coupling
Graph-Sequence	77.5	77.5	57.5
Sequence-Expert	50.0	47.5	50.0
Graph-Expert	47.5	45.0	62.5

4.3 MOJO comparison

In Table 4.2 we compare the MoJoFM distances between the decompositions generated for each of the strategies, considering the maximization of each of the metrics: complexity, cohesion, and coupling. Graph and Sequence strategies have a distance ranging between 57.5 and 77.5%, depending on which metric is optimized. This is not a major difference; in fact, because it is considerably higher than 50% we can assert that the decompositions are more similar than not. When it comes to how well each matches the Expert decomposition, starting with Sequence, we can observe that the distance to the Expert is between 47.5 and 50%. Meanwhile, Graph displays a distance between 45 and 62.5%. This means that, on average, for all metrics, Graph presents a closer proximity to the expert decompositions with 51.7% average distance, compared to Sequence's 49.2%.

4.4 Statistical Analysis

For the candidate decompositions that we consider in this evaluation, the results can be found in Table 4.3. It is observable that the decompositions generated using Graph similarity usually show an improvement across most values. The concerning value for both Sequence and Graph strategies is F-score, as any values below 0.5 are considered less than ideal and most decompositions are located in that range, with the exceptions of the lowest coupling decomposition from the Graph strategy.

Table 4.3: Statistical analysis of each of the best metric decompositions, per metric and strategy, performed with MOJO and comparing against the expert decomposition.

	Accuracy	Precision	Recall	Specificity	F-score
Lowest Complexity (Seq)	0,74	0,3	0,51	0,79	0,38
Lowest Complexity (Graph)	0.78	0.28	0.31	0.86	0.29
Highest Cohesion (Seq)	0,73	0,27	0,47	0,77	0,34
Highest Cohesion (Graph)	0.8	0.32	0.26	0.9	0.29
Lowest Coupling (Seq)	0,74	0,3	0,51	0,79	0,38
Lowest Coupling (Graph)	0.85	0.5	0.71	0.88	0.59

4.5 Graph vs Sequence vs Source of Truth

The previous two experiments showed that although candidate decompositions generated using the sequence strategy perform better according to the evaluation metrics, using the graph strategy creates results with apparent better proximity to the expert decomposition - although the very low F-score still does not allow us to come to a conclusion in that front. Through a more semantic approach, this may be explained.

When comparing the best complexity decompositions, an immediately noticeable thing in Sequence strategy's case (Table 4.4) is cluster 3's 92% of expert microservice Question. Ideally clusters would be very evenly distributed, so accumulation is already surprising, especially considering Question's 12 entity size, making it the biggest expert microservice tied with Answer. On top of that, cluster 3 also has a portion of 8 total cluster microservices, which could indicate a high dependency between the entities on those microservices and Question. Graph's best complexity decomposition's (Table 4.5) cluster that most closely resembles this situation is cluster 4 which contains parts of 7 microservices but only a 50% percentage if Question. Apart from the change in the division of Question between strategies (which went from a 92%-8% in Sequence division to 50%-33%-17% in Graph), the only other microservice to change in terms of distribution was Auth, going from 75%-25% in Sequence to 50%-25%-25% in Graph. In addition to the percentage changes, the other major change was in Answer. Although its two main representation groups in the clusters retained the same percentage of 33 and 42%, their contents vary slightly. For Sequence, the first cluster is composed of entities CodeFillInAnswerItem, CodeOrderAnswerItem, MultipleChoiceAnswerItem and their super class QuestionAnswerItem, while the second cluster was composed of entities CodeFillInAnswer, CodeOderAnswer, MultipleChoiceAnswer, their super class AnswerDetails and QuizAnswerItem. In Graph's case the groups had almost the exact same content, but switching overall percentage due to the move of QuizAnswerItem to the QuestionAnswerItem's group.

In analyzing the best cohesion decompositions, many of the same situations are encountered. Sequence's decomposition (Table 4.6) once again displayed a cluster containing 83% of the Question and involving a high number of other expert microservices. Similar situations are also observed when com-

Table 4.4: For the best complexity decomposition, using the Sequence Strategy, each cell contains the percentage of entities of the expert microservice (column) in the decomposition cluster (row).

Cluster	answer	auth	dashboard	discussion	execution	question	questionsubmission	quiz	tournament	user
0	0.42									
1					0.67	0.08	1.00		1.00	
2		0.75								0.25
3	0.17	0.25	0.25	1.00	0.33	0.92		1.00		0.50
4	0.33									
5	0.08									
6										0.25
7			0.25							
8			0.25							
9			0.25							

Table 4.5: For the best complexity decomposition, using the Graph Strategy, each cell contains the percentage of entities of the expert microservice (column) in the decomposition cluster (row).

Cluster	answer	auth	dashboard	discussion	execution	question	questionsubmission	quiz	tournament	user
0	0.33									
1					0.67	0.17	1.00		1.00	
2	0.08	0.50								0.25
3		0.25								
4	0.17	0.25		1.00	0.33	0.50		1.00		0.50
5	0.42									
6						0.33				
7			0.75							
8			0.25							
9										0.25

pared with the graph decomposition (Table 4.7) regarding both Auth and Answer, including the group content change on the latter. A difference can be found with microservice User, going from a 25%-75% split in Sequence to 25%-50%-25% in Graph due to the entity Teacher separating from Student and the super class of both, User.

The best coupling decomposition for Sequence (Table 4.8) shows similar results to the previous two cases. Graph's decomposition (Table 4.9) however shows a bigger change, as with the priority on coupling the Answer microservice went from a 42%-17%-33%-8% distribution in Sequence to a much more one-sided 92%-8%.

When taking into account the same decompositions we are studying, we can look at the average sizes of the generated decompositions' clusters that contain each entity, Table 4.10. For both expert mi-

Table 4.6: For the best cohesion decomposition, using the Sequence Strategy, each cell contains the percentage of entities of the expert microservice (column) in the decomposition cluster (row).

Cluster	answer	auth	dashboard	discussion	execution	question	questionsubmission	quiz	tournament	user
0	0.42									
1					0.67	0.17	1.00		1.00	
2		0.75								
3	0.17	0.25	0.25	1.00	0.33	0.83		1.00		0.75
4	0.33									
5	0.08									
6										0.25
7			0.25							
8			0.25							
9			0.25							

Table 4.7: For the best cohesion decomposition, using the Graph Strategy, each cell contains the percentage of entities of the expert microservice (column) in the decomposition cluster (row).

Cluster	answer	auth	dashboard	discussion	execution	question	questionsubmission	quiz	tournament	user
0	0.33		0.25							
1					0.67	0.17	1.00		1.00	
2	0.08	0.50								0.25
3		0.25								
4	0.17	0.25	0.50		0.33	0.08		1.00		0.50
5	0.42									
6						0.33				
7				1.00		0.42				
8										0.25
9			0.25							

Table 4.8: For the best coupling decomposition, using the Sequence Strategy, each cell contains the percentage of entities of the expert microservice (column) in the decomposition cluster (row).

Cluster	answer	auth	dashboard	discussion	execution	question	questionsubmission	quiz	tournament	user
0	0.42									
1					0.67	0.08	1.00		1.00	
2		0.75								0.25
3	0.17	0.25	0.25	1.00	0.33	0.92		1.00		0.50
4	0.33									
5	0.08									
6										0.25
7			0.25							
8			0.25							
9			0.25							

Table 4.9: For the best coupling decomposition, using the Graph Strategy, each cell contains the percentage of entities of the expert microservice (column) in the decomposition cluster (row).

Cluster	answer	auth	dashboard	discussion	execution	question	questionsubmission	quiz	tournament	user
0	0.92	0.25			0.33			1.00		0.50
1					0.67	0.17			1.00	
2		0.50								
3		0.25								
4				1.00		0.83				
5	0.08									0.25
6			0.75							
7			0.25							
8							1.00			
9										0.25

croservices and specially Question, their entities are usually contained in large clusters, with an average of 22 entities per cluster when considering the Sequence strategy, a value that is well above the original 12. This is justified by the fact that many entities even in other microservices depend on it. Therefore, optimization of metrics leads to these outside entities being placed in the same microservice, increasing its size.

Table 4.10: Average size of the cluster containing each entity. Decompositions are grouped by strategy and the location (cluster) of each decomposition is recorded on a decomposition basis, and the size of that location is averaged to show entities with tendency to be placed in bigger clusters.

Microservices	Entity	Avg cluster size (sequence)	Avg cluster size (graph)
answer	CodeFillInAnswer	5.00	8.67
answer	CodeOrderAnswer	5.00	8.67
answer	MultipleChoiceAnswer	5.00	8.67
answer	CodeFillInAnswerItem	4.00	9.00
answer	CodeOrderAnswerItem	4.00	9.00
answer	MultipleChoiceAnswerItem	4.00	9.00
answer	QuestionAnswerItem	4.00	9.00
answer	AnswerDetails	5.00	8.67
answer	CodeOrderAnswerSlot	1.00	3.33
answer	QuestionAnswer	22.00	14.67
answer	QuizAnswer	22.00	14.67
answer	QuizAnswerItem	5.00	9.00
auth	AuthDemoUser	3.67	3.33
auth	AuthExternalUser	3.67	1.00
auth	AuthTecnicoUser	3.67	3.33
auth	AuthUser	22.00	14.67
dashboard	Dashboard	22.00	5.67
dashboard	FailedAnswer	1.00	3.67
dashboard	DifficultQuestion	1.00	4.33
dashboard	WeeklyScore	1.00	2.33
discussion	Discussion	22.00	11.67
discussion	Reply	22.00	11.67
execution	Assessment	6.33	6.33
execution	CourseExecution	22.00	14.67
execution	TopicConjunction	6.33	6.33
question	CodeFillInQuestion	22.00	11.67
question	CodeOrderQuestion	22.00	11.67
question	MultipleChoiceQuestion	22.00	11.67
question	CodeFillInOption	22.00	6.67
question	CodeFillInSpot	22.00	6.67
question	CodeOrderSlot	22.00	6.67
question	Course	17.00	6.33
question	Image	22.00	11.67
question	Option	22.00	6.67
question	Question	22.00	13.00
question	QuestionDetails	22.00	11.67
question	Topic	6.33	6.33
questionsubmission	QuestionSubmission	6.33	5.33
questionsubmission	Review	6.33	5.33
quiz	Quiz	22.00	14.67
quiz	QuizQuestion	22.00	14.67
tournament	Tournament	6.33	6.33
user	Student	22.00	14.67
user	Teacher	10.00	1.00
user	DemoAdmin	1.00	3.33
user	User	22.00	14.67

When evaluating the same results for the Graph Strategy similar relationships are still observed, however, lessened. The most important points are still observed to be placed into the bigger clusters, such as entities QuestionAnswer, QuizAnswer, Question and QuestionDetails, although overall the size of the clusters is more balanced, with the bigger clusters reducing in size and the smaller ones increasing. This result is justified by the richer trace, with the added information allowing for more accurate

Table 4.11: Number and percentage of functionalities in common between QuizAnswer and each of the other entities in the Answer microservice. Total number of functionalities of QuizAnswer is indicated in the last row.

	Functionalities in Common	**% in common**
CodeFillInAnswer	7	25.93%
CodeOrderAnswer	7	25.93%
MultipleChoiceAnswer	7	25.93%
CodeFillInAnswerItem	7	25.93%
CodeOrderAnswerItem	7	25.93%
MultipleChoiceAnswerItem	7	25.93%
QuestionAnswerItem	9	33.33%
AnswerDetails	7	25.93%
CodeOrderAnswerSlot	1	3.70%
QuestionAnswer	21	**77.78%**
QuizAnswerItem	5	18.52%

Total Functionalities	**27**

evaluations that better reflect the system.

Something that can be noticed is the difference between Question and Answer services. Both were identified as the main and the bigger ones of the system, but most of Answer's entities are placed in smaller clusters. To explore this situation, we look further into the similarity values of each service.

Based on the previous table and an inspection of the system, we can identify the main entities of Answer as QuestionAnswer and QuizAnswer, and Question as the main entity of the service Question. We can evaluate the relationship of these entities with the remaining cluster by analyzing the number of functionalities in common with their neighboring entities. Through tables 4.11, 4.13 and 4.13 we find the number and percentage of functionalities in common that these entities have with the others in their respective microservice.

Question not only has a much bigger number of functionalities (58 total, versus 27 and 21 from QuizAnswer and QuestionAnswer respectively), but it also shares a higher percentage of them with their neighbours, making the relationships inside microservice Question stronger and therefore with higher priority in the eyes of the clustering algorithm when compared to Answer.

It is also necessary to understand how each the Sequence and Graph strategies capture these relationships. In tables 4.14, 4.15 and 4.16 we find the values for the similarity measures between QuizAnswer, QuestionAnswer and Question and the remaining entities in their microservice. Each column represents a type of similarity of the existing four (access, write, read and sequence). The values span between 0 and 1, with 1 representing maximum similarity and 0 the minimum. Similarity is calculated in both directions as, if you consider two entities e_1 and e_2, if 100% of the times that e_1 is accessed is with e_2, it does not mean that 100% of the times that e_2 is accessed then e_1 is too. This is immediately observable in any of the rows, but for instance when comparing the access similarity of 1 of CodeFillI-

Table 4.12: Number and percentage of functionalities in common between QuestionAnswer and each of the other entities in the Answer microservice. Total number of functionalities of QuestionAnswer is indicated in the last row.

	Functionalities in Common	**% in common**
CodeFillInAnswer	7	30.43%
CodeOrderAnswer	7	30.43%
MultipleChoiceAnswer	7	30.43%
CodeFillInAnswerItem	6	26.09%
CodeOrderAnswerItem	6	26.09%
MultipleChoiceAnswerItem	6	26.09%
QuestionAnswerItem	8	34.78%
AnswerDetails	7	30.43%
CodeOrderAnswerSlot	1	4.35%
QuizAnswer	21	**91.30%**
QuizAnswerItem	5	21.74%

Total Functionalities	**23**

Table 4.13: Number and percentage of functionalities in common between Question and each of the other entities in the Question microservice. Total number of functionalities of Question is indicated in the last row.

	Functionalities in Common	**% in common**
CodeFillInQuestion	32	**55.17%**
CodeOrderQuestion	32	**55.17%**
MultipleChoiceQuestion	32	**55.17%**
CodeFillInOption	15	25.86%
CodeFillInSpot	10	17.24%
CodeOrderSlot	8	13.79%
Course	24	41.38%
Image	32	**55.17%**
Option	11	18.97%
QuestionDetails	34	**58.62%**
Topic	15	25.86%

Total Functionalities	**58**

Table 4.14: Similarity between QuizAnswer and the rest of the expert microservice Answer. Each column contains a different similarity measure, and each row represents an entity of the microservice. The values are not symmetric in both directions of the relationship. First graph shows the results for Sequence Strategy, and bottom shows the results for Graph Strategy.

	Sequence (QuizAnswer vs other)				Sequence (other vs QuizAnswer)			
Entity	**access**	**write**	**read**	**sequence**	**access**	**write**	**read**	**sequence**
CodeFillInAnswer	0.25926	0.30000	0.25926	0	1	1	1	0
CodeOrderAnswer	0.25926	0.30000	0.25926	0	1	1	1	0
MultipleChoiceAnswer	0.25926	0.30000	0.22222	0	1	1	1	0
CodeFillInAnswerItem	0.25926	0	0.22222	0	1	0	1	0
CodeOrderAnswerItem	0.25926	0	0.22222	0	1	0	1	0
MultipleChoiceAnswerItem	0.25926	0	0.22222	0.00581	1	0	1	0.00581
QuestionAnswerItem	0.33333	0.40000	0.29630	0.02326	0.81818	0.88889	0.80000	0.02326
AnswerDetails	0.25926	0.30000	0.25926	0.00727	1	1	1	0.00727
CodeOrderAnswerSlot	0.03704	0.05000	0	0	1	1	0	0
QuestionAnswer	0.77778	0.90000	0.77778	0.58140	0.91304	0.90000	0.91304	0.58140
QuizAnswerItem	0.18519	0.25000	0.14815	0.01890	1	1	1	0.01890

	Graph (QuizAnswer vs other)				Graph (other vs QuizAnswer)			
Entity	**access**	**write**	**read**	**sequence**	**access**	**write**	**read**	**sequence**
CodeFillInAnswer	0.03309	0.03540	0.04093	0	0.12763	0.97864	0.98650	0
CodeOrderAnswer	0.03309	0.03540	0.04093	0.00559	0.12763	0.97864	0.98650	0.00559
MultipleChoiceAnswer	0.03309	0.03540	0.04093	0	0.12763	0.97864	0.98650	0
CodeFillInAnswerItem	0.01319	0	0.00105	0	0.05088	0	0.94507	0
CodeOrderAnswerItem	0.01319	0	0.00105	0.00003	0.05088	0	0.94507	0.00003
MultipleChoiceAnswerItem	0.01319	0	0.00105	0.00003	0.05088	0	0.94507	0.00003
QuestionAnswerItem	0.18948	0.26012	0.18858	0.01245	0.46508	0.78534	0.69440	0.01245
AnswerDetails	0.09927	0.10619	0.12280	0.00960	0.38289	0.97864	0.98650	0.00960
CodeOrderAnswerSlot	0	0	0	0	0	0.00002	0	0
QuestionAnswer	0.45524	0.59480	0.56317	0.29362	0.53441	0.78537	0.82696	0.29362
QuizAnswerItem	0.15313	0.26129	0.17293	0.02077	0.82688	0.97509	0.97278	0.02077

nAnswer with QuizAnswer, while QuizAnswer only has 0.3 write similarity with CodeFillInAnswer.

Comparing the similarity values for each of the strategies through the multiple similarity measures, it is possible to observe the same general structure of the relationships, with the values obtained through Graph strategy being a bit more disperse. Most values en down, with some previously already low becoming zero, a result of paths that were previously wrongly considered now being removed. Although a couple that had a similarity of 0 in the old strategy have grown to a very low but not null value, due to some paths that were not being considered before are now taken into account. Both situations are a direct consequence of the initial problem being solved. Overall, Graph strategy appears to provide a more elaborate and informed view of the similarity between entities, as proved by the better approximation of the source of truth, as shown earlier.

Through this information we can also conclude the difference between Question and Answer. Tables 4.14, 4.15 and 4.16 show that there isn't a major difference in the similarity values between QuizAnswer, QuestionAnswer and Question and the other entities in their microservice. However, considering a similar similarity and the high disparity in the number and percentage of functionalities in common with the microservice, it still makes sense for microservice Question to stick together in general than microservice Answer.

Table 4.15: Similarity between QuestionAnswer and the rest of the expert microservice Answer. Each column contains a different similarity measure, and each row represents an entity of the microservice. The values are not symmetric in both directions of the relationship. First graph shows the results for Sequence Strategy, and bottom shows the results for Graph Strategy.

	Sequence (QuestionAnswer vs other)				Sequence (other vs QuestionAnswer)			
Entity	**access**	**write**	**read**	**sequence**	**access**	**write**	**read**	**sequence**
CodeFillInAnswer	0.30435	0.30000	0.30435	0.01599	1	1	1	0.01599
CodeOrderAnswer	0.30435	0.30000	0.30435	0.01744	1	1	1	0.01744
MultipleChoiceAnswer	0.30435	0.30000	0.26087	0.01744	1	1	1	0.01744
CodeFillInAnswerItem	0.26087	0	0.26087	0	0.85714	0	1	0
CodeOrderAnswerItem	0.26087	0	0.26087	0	0.85714	0	1	0
MultipleChoiceAnswerItem	0.26087	0	0.26087	0	0.85714	0	1	0
QuestionAnswerItem	0.34783	0.40000	0.34783	0.00581	0.72727	0.88889	0.80000	0.00581
AnswerDetails	0.30435	0.30000	0.30435	0.20785	1	1	1	0.20785
CodeOrderAnswerSlot	0.04348	0.05000	0	0	1	1	0	0
QuestionAnswer	0.91304	0.90000	0.91304	0.58140	0.77778	0.90000	0.77778	0.58140
QuizAnswerItem	0.21739	0.25000	0.17391	0.01163	1	1	1	0.01163

	Graph (QuestionAnswer vs other)				Graph (other vs QuestionAnswer)			
Entity	**access**	**write**	**read**	**sequence**	**access**	**write**	**read**	**sequence**
CodeFillInAnswer	0.02424	0.01870	0.03751	0.00320	0.07964	0.39164	0.61555	0.00320
CodeOrderAnswer	0.02424	0.01870	0.03751	0.00320	0.07964	0.39164	0.61556	0.00320
MultipleChoiceAnswer	0.02424	0.01870	0.03751	0.00320	0.07964	0.39164	0.61556	0.00320
CodeFillInAnswerItem	0.00063	0	0.00098	0	0.00208	0	0.60246	0
CodeOrderAnswerItem	0.00063	0	0.00098	0	0.00208	0	0.60246	0
MultipleChoiceAnswerItem	0.00063	0	0.00098	0	0.00208	0	0.60246	0
QuestionAnswerItem	0.07791	0.14952	0.12055	0.00050	0.16289	0.34190	0.30231	0.00050
AnswerDetails	0.07271	0.05611	0.11252	0.06176	0.23892	0.39164	0.61556	0.06176
CodeOrderAnswerSlot	0	0	0	0	0	0.00002	0	0
QuestionAnswer	0.53441	0.78537	0.82696	0.29362	0.45524	0.59480	0.56317	0.29362
QuizAnswerItem	0.07133	0.13690	0.10020	0.00753	0.32812	0.38693	0.38386	0.00753

Table 4.16: Similarity between QuestionAnswer and the rest of the expert microservice Answer. Each column contains a different similarity measure, and each row represents an entity of the microservice. The values are not symmetric in both directions of the relationship. First graph shows the results for Sequence Strategy, and bottom shows the results for Graph Strategy.

	Sequence (Question vs other)				Sequence (other vs Question)			
Entity	**access**	**write**	**read**	**sequence**	**access**	**write**	**read**	**sequence**
CodeFillInQuestion	0.55172	0.16667	0.56140	0.01308	1	1	1	0.01308
CodeOrderQuestion	0.55172	0.16667	0.56140	0.00872	1	1	1	0.00872
MultipleChoiceQuestion	0.55172	0.16667	0.56140	0.01744	1	1	1	0.01744
CodeFillInOption	0.25862	0.16667	0.21053	0.00145	1	1	1	0.00145
CodeFillInSpot	0.17241	0.16667	0.17544	0	1	1	1	0
CodeOrderSlot	0.13793	0.16667	0.14035	0.00291	1	1	1	0.00291
Course	0.41379	0.29167	0.40351	0.12645	0.70588	0.50000	0.67647	0.12645
Image	0.55172	0.20833	0.56140	0.27907	1	1	1	0.27907
Option	0.18966	0.16667	0.19298	0.00291	1	1	1	0.00291
QuestionDetails	0.58621	0.16667	0.59649	0.16279	1	1	1	0.16279
Topic	0.25862	0.12500	0.26316	0.05523	0.65217	0.27273	0.65217	0.05523

	Graph (Question vs other)				Graph (other vs Question)			
Entity	**access**	**write**	**read**	**sequence**	**access**	**write**	**read**	**sequence**
CodeFillInQuestion	0.10016	0.05070	0.13676	0.01593	0.18154	0.90847	0.91084	0.01593
CodeOrderQuestion	0.10016	0.05070	0.13676	0.00889	0.18154	0.90847	0.91084	0.00889
MultipleChoiceQuestion	0.10016	0.05070	0.13676	0.00889	0.18154	0.90847	0.91084	0.00889
CodeFillInOption	0.02459	0.01963	0.02568	0.00235	0.09508	0.90847	0.85939	0.00235
CodeFillInSpot	0.01958	0.02231	0.02673	0	0.11355	0.90847	0.86335	0
CodeOrderSlot	0.01944	0.02231	0.02655	0.00481	0.14095	0.90847	0.86253	0.00481
Course	0.26512	0.32482	0.33846	0.11613	0.46597	0.45043	0.58794	0.11613
Image	0.27980	0.17693	0.38205	0.23167	0.50714	0.93672	0.92293	0.23167
Option	0.03184	0.04462	0.04347	0.00470	0.16787	0.90847	0.88474	0.00470
QuestionDetails	0.24354	0.05070	0.33254	0.08540	0.41545	0.90847	0.89236	0.08540
Topic	0.15707	0.00679	0.21447	0.05347	0.39610	0.01317	0.46370	0.05347

4.6 Discussion

Recapping the research questions proposed at the start:

- RQ1: Does control flow information affect the quality of microservice decompositions generated through static analysis?
- RQ2: How do candidate decompositions generated using static analysis, with and without control information, compare to an expert decomposition?

In response to RQ1, Graph Strategy is the method proposed that enriches current static analysis processes with control flow information. As seen in the previous section, the decompositions using this strategy achieved worse results in terms of metrics (Complexity, Cohesion and Coupling) than the previous approach. It is still important to note, as previously mentioned, that the metrics do not provide an absolute measure of the quality of the decomposition. Through qualitative analysis, it was identified that the new strategy retained the main points of the previous one, while offering more in-depth results.

Which takes us to RQ2: When comparing both strategies against the expert decomposition, both strategies (with or without control information) displayed very similar results, with Graph Strategy offering slightly better and closer results.

4.7 Threats to Validity

A problem in particular with the new proposed strategy is the structure of the Branch Heuristics, described in chapter 3. The heuristics, proposed by Ball and Larus [25], were designed for C and Fortran languages of much lower language than Java. This means that some of the heuristic rules do not apply to the same extent to the Spring-boot programs that Mono2micro evaluates. The new Graph strategy would benefit from implementing heuristics designed specifically for Java, and also from having the Branch Frequency [1] for each of the heuristics studied and calculated with Spring-boot applications in mind. However, Java was designed based on the C syntax and some of the heuristics are independent of the programming. For instance, the Guard Heuristic, which applies when variables are only assigned a value after participating in a condition, is still valid in Java. However, others such as the Opcode Heuristic, which predicts that a comparison of an integer for less than or equal to zero will fail, is not as accurate, as its origin is based on integer values lower than zero being used as error codes, a practice that does not apply to Java. Another language-specific problem is partially or fully missing support for some Java-specific component logic. Components such as try-catch, exceptions or a full implementation of the switch branching component are not considered in the current iteration of the technique, and their prevalence in programs written using the language means the results may not be entirely accurate

As mentioned in the Evaluation chapter, due to time constraints, it was not possible to calculate the metrics based on the graph representation; instead, it was done based on the old sequence representation of the codebase. A consequence of this is that the values for the complexity, cohesion and coupling metrics, specifically for the decompositions of the Graph Strategy, could suffer some changes from what was used in the comparison. Furthermore, in the study of the metrics between both strategies, only a single codebase, Quizzes-Tutor, was used. Although it allowed to perform a closer and more qualitative analysis of the products of each strategy, the analysis of the metrics could have benefited from using a much higher number of codebases, in order to have more varied study data and produce more accurate results.

5

Conclusion

As programs and monoliths grew larger and more complex and their restructuring became necessary, many tools have been developed over the years to attempt to fix the problem of automatic microservice decomposition. One of these tools, Mono2Micro, has been in development for several years. As its peers, it employs multiple techniques, which can be split in dynamic analysis and static analysis. While dynamic analysis performs a more thorough evaluation of the target codebases according to their use, it requires a massive amount of data and time to function. On the hand, static analysis runs much faster and requires fewer data, but the results may be more generic. Therefore, the question is whether it would be possible to create a hybrid technique, borrowing the quick processing and low data requirements of static analysis, and the more informed result of dynamic analysis.

A possible way to do this is to enrich the current static analysis methods with more context data, allowing it to better approximate how the program will run. With this, control flow was chosen as the type of information to be imbued in the representations analyzed by the static tool. We propose a new trace representation that adds this data into the previously used sequence of accesses, transforming into a graph of accesses. This process is divided into the first steps of the decomposition pipeline, Collection and Decomposition Generation. The first step involves the creation of a new collector, which identifies branching elements in the code and creates descriptions of them in the codebase's text representation. This will then be used in the generation step, to create a Java based graph of accesses. Through the

use of Ball and Wu's Branch Frequency approximation, it is possible to heuristically approximate the probability of each of the branches in the graph of accesses, therefore calculating similarities between codebase entities with values closer to the ones that would be obtained during the system execution.

As a result, it is concluded that the added information does not provide a best solution in terms of the decomposition metrics. Although they are not an absolute measure of the quality of the results, the new technique created decompositions with metric values worse than the previous technique. On the other hand, it also generated results that are more closely aligned with the architect decomposition of the studied system. However, the validity of these results is still in question, as both the previous and current techniques have a below average F-score value on their results, putting in question their accuracy. Besides this, when it comes to the semantic analysis of the generated decompositions it was found that the new proposed strategy allows for a bigger separation between entities, which translates into a bigger distance shared with the more central domain entities of the codebase. In the previous approach, these entities shared high similarity values with many other entities, promoting the creation of few large clusters and the rest remaining very small. Lower similarity values mean that the bigger clusters get smaller and the smaller clusters get larger, while retaining the importance of the relations.

However, research has many paths to explore. Although control flow data are an important thing to keep in mind in the trace, there may be other information sources in the database that would lead to the generation of better decompositions.

Some future work related to the new technique, in particular, besides addressing the problems described in 4.7, could include the implementation of support for the proposed trace representation during the Visualization stage, allowing the users of the tool not only to see the sequence of accesses of each functionality (as they can do now), but also to see them as the new graph of accesses along with the probability of every path. Furthermore, the evaluation performed considers only the comparison with the previous static technique. Comparison with dynamic analysis would also allow us to show whether the new process is an improvement of static analysis.

Bibliography

[1] Y. Wu and J. R. Larus, “Static branch frequency and program profile analysis,” in *Proceedings of the 27th annual international symposium on Microarchitecture*, 1994, pp. 1–11.

[2] M. Ernst, “Static and dynamic analysis: Synergy and duality,” *WODA 2003: ICSE Workshop on Dynamic Analysis*, 05 2003.

[3] Z. Wen and V. Tzerpos, “An effectiveness measure for software clustering algorithms,” in *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, 2004, pp. 194–203.

[4] S. Santos and A. R. Silva, “Microservices identification in monolith systems: Functionality redesign complexity and evaluation of similarity measures,” *Journal of Web Engineering*, vol. 21, no. 5, pp. 1543–1582, 2022.

[5] A. K. Kalia, J. Xiao, R. Krishna, S. Sinha, M. Vukovic, and D. Banerjee, “Mono2micro: a practical and effective tool for decomposing monolithic java applications to microservices,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1214–1224.

[6] A. K. Kalia, J. Xiao, C. Lin, S. Sinha, J. Rofrano, M. Vukovic, and D. Banerjee, “Mono2micro: an ai-based toolchain for evolving monolithic enterprise applications to a microservice architecture,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1606–1610.

[7] R. Nakazawa, T. Ueda, M. Enoki, and H. Horii, “Visualization tool for designing microservices with the monolith-first approach,” in *2018 IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 2018, pp. 32–42.

[8] D. Taibi and K. Systä, “From monolithic systems to microservices: A decomposition framework based on process mining,” in *International Conference on Cloud Computing and Services Science*. SciTePress, 2019, pp. 153–164.

[9] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, “Service cutter: A systematic approach to service decomposition,” in *Service-Oriented and Cloud Computing: 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings 5*. Springer, 2016, pp. 185–200.

[10] G. Mazlami, J. Cito, and P. Leitner, “Extraction of microservices from monolithic software architectures,” in *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 2017, pp. 524–531.

[11] G. Mazlami, “Algorithmic extraction of microservices from monolithic code bases,” Ph.D. dissertation, MS thesis, University of Zurich, 2017.

[12] T. Lopes and A. R. Silva, “Monolith microservices identification: Towards an extensible multiple strategy tool,” in *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*, 2023, pp. 111–115.

[13] M. Abdelkader, M. Malki, and S. M. Benslimane, “A heuristic approach to locate candidate web service in legacy software,” *International journal of computer applications in technology*, vol. 47, no. 2-3, pp. 152–161, 2013.

[14] J. Bogner, S. Wagner, and A. Zimmermann, “Automatically measuring the maintainability of service- and microservice-based systems: a literature review,” in *Proceedings of the 27th international workshop on software measurement and 12th international conference on software process and product measurement*, 2017, pp. 107–115.

[15] M. Perepletchikov, C. Ryan, K. Frampton, and Z. Tari, “Coupling metrics for predicting maintainability in service-oriented designs,” in *2007 Australian Software Engineering Conference (ASWEC’07)*. IEEE, 2007, pp. 329–340.

[16] L. Nunes, N. Santos, and A. Rito Silva, “From a monolith to a microservices architecture: An approach based on transactional contexts,” in *Software Architecture: 13th European Conference, ECSA 2019, Paris, France, September 9–13, 2019, Proceedings 13*. Springer, 2019, pp. 37–52.

[17] T. C. Matias, “Streamlined refactoring of modern web frameworks to microservices,” Ph.D. dissertation, Universidade do Porto (Portugal), 2019.

[18] J. Correia and A. Rito Silva, “Identification of monolith functionality refactorings for microservices migration,” *Software: Practice and Experience*, vol. 52, no. 12, pp. 2664–2683, 2022.

[19] J. Marshall and G. Kotonya, “A runtime visualizer for microservices,” in *2021 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2021, pp. 72–80.

[20] T. Ball, “The concept of dynamic analysis,” *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 216–234, 1999.

[21] B. Andrade, S. Santos, and A. R. Silva, “A comparison of static and dynamic analysis to identify microservices in monolith systems,” in *Software Architecture*, B. “Tekinerdogan, C. Trubiani, C. Tibermacine, P. Scandurra, and C. E. Cuesta, Eds. Cham: Springer Nature Switzerland, 2023, pp. 354–361.

[22] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau, “Parallel processing: A smart compiler and a dumb machine,” in *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, 1984, pp. 37–47.

[23] W. G. Morris, “Ccg: A prototype coagulating code generator,” in *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, 1991, pp. 45–58.

[24] R. L. Sites, *Alpha architecture reference manual*. Digital press, 1998.

[25] T. Ball and J. R. Larus, “Branch prediction for free,” *ACM SIGPLAN Notices*, vol. 28, no. 6, pp. 300–313, 1993.

[26] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, “Spoon: A library for implementing analyses and transformations of java source code,” *Software: Practice and Experience*, vol. 46, no. 9, pp. 1155–1179, 2016.

[27] G. Shafer, *A mathematical theory of evidence*. Princeton university press, 1976, vol. 42.

[28] D. Michail, J. Kinable, B. Naveh, and J. V. Sichi, “Jgrapht—a java library for graph data structures and algorithms,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 46, no. 2, pp. 1–29, 2020.