

UNIVERSITY OF BAHRAIN

College of Information Technology



Dynamic Feature Location Framework for Software Project

A Thesis Submitted in Partial Fulfilment of the Requirements for the
Doctor of Philosophy Degree in Computing and Information Sciences

Submitted By

Faisal Buzaid

202011222

Supervised By

Dr. Fawzi Albalooshi

(Associate Professor)

University of Bahrain

August 2024



Dissertation Deposit Form

Author Name:	Faisal Mohammed Buzaid		
Email:	202011222@stu.uob.edu.bh	Contact No:	32066399
College:	Information Technology		
Department:	Computer Science		
Thesis Title:	Dynamic Feature Location Framework for Software Project		
Year:	2024		

Author's Declaration: I agree to the following conditions:

The above thesis will be available (in the University of Bahrain library and externally) and reproduced as necessary at the discretion of the University of Bahrain. It may also be digitized by the University of Bahrain and made available via the university's repository or via other parties on the Internet.

This condition shall apply to ALL copies including the electronic copies

The above thesis has been provided on the understanding that it is copyright protected material and that no quotation from this thesis may be published without proper acknowledgement.

Signature of the Author	Faisal Mohammed Buzaid	Date:	Apr-24
-------------------------	------------------------	-------	--------

10 Approved at the University Council 6th meeting, 2013
In accordance with Decision No. 736/2013 dated 14/3/2013
Modified by University Council Decision No. 2043/2013 dated 20/11/2013.

DECLARATION

I declare that this work is the result of my own investigation and that it has not already been accepted in substance for any degree, nor is it currently submitted for any degree.

ABSTRACT

The Dynamic Feature Location Techniques (DFLTs) aim to automate the process of identifying the source code responsible for executing specific features within software systems. Manual implementation of DFLT is time-consuming and demanding for developers, leading to the proposal of semi-automated approaches. One common approach involves generating execution traces by executing multiple scenarios for each software feature and then mapping the corresponding source code based on these traces. However, the execution traces are often large and contain irrelevant data to the software feature, requiring solutions to reduce their size and eliminate irrelevant data. One such solution involves minimizing the number of scenarios needed to exercise a software feature, but little work has been done in this area.

To address this gap, a generic framework called Aggregation of Execution Traces to Formulate a Scenario (AETFS) is introduced in this work. AETFS leverages runtime software output and employs textual analysis techniques to extract relevant data from the execution trace for scenario creation. It explores textual analysis, including topic modeling, as a means to select accurate scenarios for DFLT. The performance of AETFS is characterized in terms of execution trace granularity, enabling the identification of meaningful terms that can filter the execution trace using textual analysis techniques such as Latent Semantic Indexing (LSI). The evaluation encompasses eight subject systems with 600 features, making it more extensive than previous studies.

The study identifies certain attributes of execution traces and text queries that impact AETFS's performance. Two distinct groups emerge, one achieving superior Feature Location (FL) using AETFS and the other achieving better FL using a conventional baseline method. Combining AETFS with the baseline method significantly enhances performance, with the top results surpassing the baseline by 45% and the lowest by 12% over AETFS.

In conclusion, this work highlights the importance of rigorously characterizing the proposed DFLT framework to identify optimal scenarios for exercising software features. It emphasizes the need to differentiate between scenarios and their characterizations to generate necessary insights. The findings demonstrate the effectiveness of AETFS while providing valuable insights for further advancements in the field of DFLT.

Contents

APPROVAL	A
DECLARATION	B
ABSTRACT	C
TABLE OF CONTENTS	D
LIST OF FIGURES	J
LIST OF TABLES	M
LIST OF LISTINGS	O
LIST OF ALGORITHMS	P
ACKNOWLEDGMENTS	Q
DEDICATION	R
PUBLICATIONS	S
LIST OF ABBREVIATIONS	U
1 Introduction	1
1.1 Software Features and Feature Implementation	1
1.1.1 The Applications of Feature Location	4
1.1.2 Type of Feature Location Techniques	4
1.2 The Motivation of Using Dynamic Feature Location Tech- niques	7
1.3 Monitoring the Software System at Runtime	9
1.3.1 Formulating the Scenarios in Feature Location . . .	11
1.3.2 Instrumentation in Feature Location	12

1.3.3	Execution Traces in Feature Location	13
1.4	The Research Gap	14
1.5	The Research Objectives	16
1.6	The Research Questions	18
1.6.1	Questions	18
1.6.2	Initial Proposed Solution to Answer Each Research Question	23
1.7	Contributions	34
1.8	Thesis Outline	35
2	Literature Review	38
2.1	Literature Review	38
2.2	Dynamic Feature Location Background	39
2.2.1	Definitions of Feature Location	41
2.2.2	Input of DFLTs	42
2.2.3	Instrumentation Techniques in DFLTs	43
2.2.4	Type of Analysis in DFLTs	43
2.3	Review of Publications	45
2.3.1	Footstep to Comprehend Source Code	45
2.3.2	Full Dynamic Approach of Feature Location	46
2.3.3	Hybrid Approach in Dynamic Feature Location	47
2.4	Systematic Literature Review	48
2.4.1	General Setting for Search Techniques	49
2.4.2	General Setting for Reading Process	50
2.5	Analysing the Finding of the Systematic Review	51
2.5.1	Common FL Techniques Used in Publications	51
2.5.2	Classification of DFLTs	52
2.5.3	A Common Shared Applications in DFLTs	53
2.5.4	Challenges	56
2.5.5	Benchmark Creation	57

2.5.6	Data Collection	57
2.5.6.1	Obtaining the Execution Trace Configurations	58
2.5.6.2	Aggregating Execution Traces by Granularity and Based on the Partitioned Scenarios	61
2.5.7	Indexing the Content of the Collected Execution Traces	69
2.6	Future Directions	70
2.7	Conclusion	71
3	Proposed Methodology	73
3.1	Overview	73
3.2	The AETFS Approach	74
3.2.1	Preliminaries and Notations	77
3.2.2	Major Steps in the AETFS Approach	77
3.3	AETFS Technique in Details	78
3.3.1	The Running Example	79
3.3.2	Partitioning of the formulated Scenarios	83
3.3.3	Annotating Partitioned Scenarios with the Relevant Execution Traces	84
3.3.3.1	AETFS Configuration: Grouping and Classification of the Content of the Execution Traces based on Using Bytecode Instrumentation	84
3.3.3.2	Defining Accuracy in the Context of AETFS	86
3.4	The Baseline Approach: DFLT's baseline	86
3.5	Implementation details	90
4	Experiment and Case Studies	91
4.1	Mapping features to source code based on execution trace . .	93

4.1.1	Overview	93
4.1.2	Design The experiment	95
4.1.2.1	Generating Execution Trace Process . . .	96
4.1.2.2	Filtering Execution Trace by Query Re- formulation	99
4.1.2.3	Measuring Methods Distance	103
4.2	Scenario Sifter to Filter Irrelevant Inputs in DFLTs	105
4.2.1	Overview	105
4.2.2	Experiment Design	107
4.2.2.1	Data input in Scenario Sifter system . . .	111
4.2.2.2	Scenario segmentation	113
4.2.2.3	Instrumented Feature	113
4.2.2.4	Processing Execution Trace	117
4.2.2.5	Selected Execution Trace	124
4.2.2.6	Data Output in Scenario Sifter System . .	125
4.2.3	Implementing and Execution Phase	126
4.2.3.1	The case study targeted systems	126
4.2.3.2	Formulate the input scenario and query . .	128
4.2.3.3	Instrumentation	131
4.2.3.4	Execution trace analysis	135
4.3	The Use of UML Diagrams to Enhance Dynamic Feature Location Techniques	138
4.3.1	Overview	139
4.3.2	Design The experiment	141
4.3.2.1	Text Scenario Conversion	142
4.3.2.2	Parsing Text Stories	142
4.3.2.3	Plugin Creation	143
4.3.2.4	Use Case Diagram Extraction	143
4.3.2.5	Execution Trace Collection	146

4.3.2.6	Execution Trace Filtering	148
5	Assessment and Evaluation of the AETFS Methodology	153
5.1	The Initial Investigation	154
5.1.1	The Research Questions of the Initial Investigation .	154
5.1.2	Preliminary Findings and Examination	156
5.2	Assessment and Evaluation of the Mapping features to source code based on execution trace	161
5.2.1	Results and Discussion of using MappFL to config- ure AETFS	167
5.2.2	Threat to Validity	170
5.3	Assessment and Evaluation of Scenario Sifter to Filter Irrel- evant Inputs	171
5.3.1	Results and Discussion of using Scenario Sifter to configure AETFS	174
5.3.2	Threat to Validity	177
5.4	Assessment and Evaluation of The Use of UML Diagrams to Enhance DFLTs	181
5.4.1	Converting the Text Scenario into a Use Case Diagram	183
5.4.2	Generating and Collecting Execution Traces	185
5.4.3	Trace Filtering	186
5.4.4	Visualizing the Object Diagram by Using Execution Trace	187
5.4.5	Visualizing the Class Diagram by Using Execution Trace	189
5.4.6	Results and Discussion of using UML Use Case Di- agram to configure AETFS	190
5.4.7	Threat to Validity	195
5.5	Evaluation Summary for the Three Experiments Configura- tion of the AETFS	196

6	Conclusions and Future Research Work	201
6.1	Contributions	202
6.2	Addressing the Study's Inquiries	203
6.3	Conclusion	207
6.4	Future Work	209
	References	212
	Abstract (in Arabic)	

List of Figures

1.1	The mapping relationship between the software features and their relevant source code.	3
1.2	The main required elements to implement DFLT's to map the source code to the relevant feature.	11
1.3	The focus of this research.	16
2.1	General steps to implement DFLT's.	40
2.2	The number of scientific publications referring to the search of FLT's and DFLT's in the period 2012-2020. Source: Scopus (accessed on 31 March 2021).	50
2.3	Number of techniques used.	52
2.4	The main classification of DFLT's Taxonomy.	53
2.5	Java code snippet that demonstrates how you can use Java agent bytecode instrumentation to capture mouse click events and generate execution traces.	64
2.6	Aggregation of Execution Traces for partitioned scenarios for the running example.	66
2.7	Generic sub-scenarios aggregation approach.	67
3.1	The AETFS approach.	75
3.2	Random Scenario to exercise Activity Diagram in ArgoUML.	

3.3	Random Scenarios that are formulated by using Gherkin framework to exercise Activity Diagram in ArgoUML. . . .	80
3.4	The baseline DFLT's approach to index the content of the execution traces	89
4.1	The main steps of designing the experiment to map feature to the source code by using execution trace.	96
4.2	Steps of generating execution trace process.	99
4.3	Corpus indexing process.	101
4.4	Scenario Sifter Proposed Methodology Diagram to Filter Irrelevant Scenario's Inputs.	109
4.5	The final output of the Scenario Sifter to show important part of the initial Scenario.	111
4.6	DV to show how the method is represented with terms. . . .	123
4.7	A brief methodology overview entails four key steps: text scenario conversion, execution trace collection, trace filtering, and diagram extraction.	142
5.1	Scenarios to exhibit two features in ArgoUML.	162
5.2	Java source code after instrumentation.	164
5.3	Process to instrument XML files.	165
5.4	Presenting DFLT's Evaluation Results of Using MappFL to Configure AETFS by Using Line Graph	171
5.5	Evaluation process to measure the P, R, and F for each implemented DFLT's	173
5.6	Comparison between the performance of DFLT's with using Scenario Sifter and without using Scenario Sifter.	179
5.7	Comparison between the performance of DFLT's with using Scenario Sifter and without using Scenario Sifter.	180

5.8	The generated UML use case diagram based on the text scenario.	184
5.9	Snapshot of the generated execution trace by using AspectJ to profile a software functionality.	185
5.10	Snapshot of the generated UML Object Diagram by using JIVE tool.	188
5.11	The UML class diagram template of Meta-model to generate use case diagram 1231212312	190
5.12	Performance Comparison of DFLTs with and without UML Use Case Techniques	194
5.13	Comparison of Case Study Evaluation for Three AETFS Configurations	200

List of Tables

2.1	Table of Search Parameters for Feature Location Publications	49
2.2	A CLASSIFICATION VIEW AMONG THE MOST COMMON TECHNIQUES THAT ARE USED IN DFLT _s	55
60		
3.1	Comparison between the collected execution trace that are generated by using different scenarios to exercise Activity Diagram Feature in the running examples	82
4.1	The basic symbols and definitions used in Scenario Sifter . .	110
4.2	Method-Term matrix based on LSI concept	121
4.3	Application systems that are used to experiment Scenario Sifter	128
4.4	Illustration to shows the number of execution traces for each scenario	130
4.5	Execution traces analysis for every sub-scenario by using IR techniques.	137
5.1	Evaluation Metrics for the Configuration of AETFS in Applying DFLTS	158
159		
5.3	Comparison of P, R, and F-Measure with and without AETFS	160
5.4	Feature, Scenarios, LOC, and Number of Methods (M) . . .	163

5.5	Feature with Number of methods by using reformulating query technique	166
5.6	DFLTs Evaluation Results of Using MappFL to Configure AETFS	168
5.7	An evaluation of using Scenario Sifter to enhance the performance of DFLT's against three Java open-source benchmark	174
5.8	An evaluation of using DFLT's against three Java open-source benchmark without using Scenario Sifter	176
5.9	An evaluation of using state-of-art DFLT's that enhanced by using Scenario Sifter against three Java open-source benchmark	177
5.10	An evaluation of using DFLT called MappFL against three Java open-source benchmark without using Scenario Sifter .	178
5.11	Summary of ATM System Components	181
5.12	Detailed Analysis of ATM System Methods	182
5.13	Tracematch technique uses the list of prefix to filter the execution trace.	187
5.14	Comparison of DFLT's Performance with and without UML Use Case Techniques in AETFS Configuration	192
5.15	Comparison of Case Study Evaluation for Three AETFS Configuration	197

Listings

- 4.1 Javassist instrumentation technique to profile data 133
- 4.2 Snapshot of ClassFileTransform to profile invoked methods . 134
- 4.3 Handle mouse button click event to invoke instrumentation . 138
- 147
- 4.5 A snapshot of tracematch technique to filter an execution trace 148

List of Algorithms

1	Proposed Algorithm to construct UML use case diagram from the user text stories	144
2		149

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Dr. Fawzi Al-balooshi for his great support, help, and motivation. His guidance helped me in all the time of my research and during the writing of this thesis.

DEDICATION

To my parents for their encouragements.

PUBLICATIONS

This thesis is based on previously published papers written jointly with my Supervisor, Dr Fawzi Albalooshi, and others, Dr. Mustafa Hammad, Dr. Wael Elmedany, and Dr. Mohsen Rouached.

JOURNAL PAPERS

1. (To be Submitted) Buzaid, Faisal, Fawzi Albalooshi, and Mustafa Hammad. "Scenario Sifter to Enhance the Input of Dynamic Feature Location Techniques." International Journal of Computing and Digital Systems, pp. In Press. University Of Bahrain, 2024

CONFERENCE PAPERS

1. (Published) Buzaid, Faisal, and Mustafa Hammad. "Reviewing dynamic feature location techniques: Basic elements and challenges." In 2021 International Conference on Innovation and Intelligence for Informatics, Computing and Technologies (3ICT), pp. 242–249. IEEE, 2021.

URL: <https://ieeexplore.ieee.org/document/9581921>

2. (Published) Buzaid, Faisal, and Mustafa Hammad. "Mapping features to source code based on execution trace." 4th Smart Cities Symposium (SCS 2021), pp. 72–77. IET, 2021.

URL: <https://ieeexplore.ieee.org/document/9770600>

3. (Published) Buzaid, Faisal, Fawzi Albalooshi and Wael Elmedany. "The Use of UML Diagrams to Enhance Dynamic Feature Location Techniques." In 2021 International Conference on Innovation and Intelligence for Informatics, Computing and Technologies (3ICT), pp. 285–292. IEEE, 2022.

URL: <https://ieeexplore.ieee.org/document/9990768>

4. (Published) Buzaid, Faisal, Fawzi Albalooshi and Mohsen Rouached. "Utilizing the Output of Azure IoT Hub to Comprehend the Software Functionalities." 6th Smart Cities Symposium (SCS 2022), pp. 258–263. IET, 2022.

URL: <https://ieeexplore.ieee.org/document/10137500>

LIST OF ABBREVIATIONS

AETFS	Aggregation of Execution Traces to Formulate a Scenario
AI	Artificial Intelligence
API	Application Program Interface
BCI	Bytecode Instrumentation
BDD	Behavior-Driven Development
BL	Bug Localization
CAS	Context-Aware Stemming
DFT	Dynamic Feature Trace
DFLTs	Dynamic Feature Location Techniques
DL	Deep Learning
F	Feature
FL	Feature Location
GQM	Goal-Question-Metric
GUI	Graphical User Interface
HFLTs	Hybrid Feature Location Techniques
IoT	Internet of Things
IR	Information Retrieval
JVM	Java Virtual Machine

LDA	Latent Dirichlet Allocation
LSI	Latent Semantic Indexing
NL	Natural Language
NLP	Natural Language Processing
OOPL	Object Oriented Programming Language
OSIA	Online Synchronous Instrumentation Approach
PN	Plural Noun or Proper Noun
POS	Part Of Speech
VVP	Past Participle Verb
Q	Query
Q2P	Query Quality Prediction
SC	Software Comprehension
SFLT s	Static Feature Location Techniques
SM	Software Maintenance
SN	Singular Noun
SPR	Scenario-Based Probabilistic Ranking
SR	Software Reconnaissance
SVD	Singular Value Decomposition
TF-IDF	Term Frequency–Inverse Document Frequency

TFLT s	Textual Feature Location Techniques
TR	Textual Retrieval
UI	User Interface
UML	Unified Modeling Language
V	Verb
VPD	Verb with a Direct Object
VSM	Vector Space Modelling

Chapter 1

Introduction

1.1 Software Features and Feature Implementation

In today's era of software development, the development of software projects is driven by the needs of the users. The task of software developers is to transform these documented needs into tangible features within the software systems. A feature (F) is a term used to denote a unique aspect of a software's functionality that is realized in its source code (IEEE 1983). In this context, a software feature is defined as a system functionality that is visible and can be activated by users (Razzaq, Wasala, Exton, & Buckley, 2018). For instance, in Microsoft Word¹, users can activate a specific software function, such as *Save As*², by navigating through the program's Graphical User Interface (GUI), thereby demonstrating the software feature (*Save As*).

However, the feature implementation is a tangled process. Despite the invoked software feature being observable by the end-users in the User Interface (UI) layer, the fact is that the software project is designed to be architecture based on several layers, such as the Application layer, Business

¹Microsoft Word is a word processor developed by Microsoft.

²<http://microsoft.com/en-us/microsoft-365/word>

layer, Persistence layer, and Database layer (Márquez, Astudillo, & Kazman, 2023). Consequently, the software feature is implemented across the software architecture layers. Each software architecture layer may have elements of the observable feature. In other words, there is a certain orthogonality between software features and software architecture; the latter can provide clues as to where features are implemented. For instance, the Application layer contains the source code (methods/classes or any other level of the source code granularity) of the observable feature in the UI layer. In contrast, the Database layer may contain some related attributes to the F. Hence, this work primarily focuses on mapping software's artifacts between the UI layer (software functionalities) and the Application layer (source code). Other software architecture layers and Internet of Things (IoT) software architecture (Buzaid, Albalooshi, & Rouached, 2022) are excluded from this work.

The characteristics of software features and their implementation can be described as follows:

- An abstract feature can be defined as a high-level description of a software feature that presents its concept without any associated physical source code entity.
- A functional software feature can be defined as those features of the software that are apparent to the end user in the UI layer.
- An observable software feature refers to the ability of end-users to identify functional software features and locate their corresponding source code.

The goal of implementing software features is to link the functional software feature with the corresponding source code entity. This mapping provides the necessary source code for the targeted software feature at various

levels of granularity, such as methods, classes, and so on. Moreover, it's rare to find a modern software system designed based on a monolithic architecture these days. Most software projects are now developed using Object Oriented Programming Language (OOPL) (Flageol, Menaud, Gu  h  neuc, Badri, & Monnier, 2023; Jaspan et al., 2018). As a result, finding the relevant source code for a software feature may involve multiple files, including external libraries and packages. For instance, software developers might need to locate various source code entities like classes, methods, variables, etc., across several files to identify a near-complete set of features, rather than just pinpointing the initial point of implementation (Razzaq, Le Gear, Exton, & Buckley, 2020).

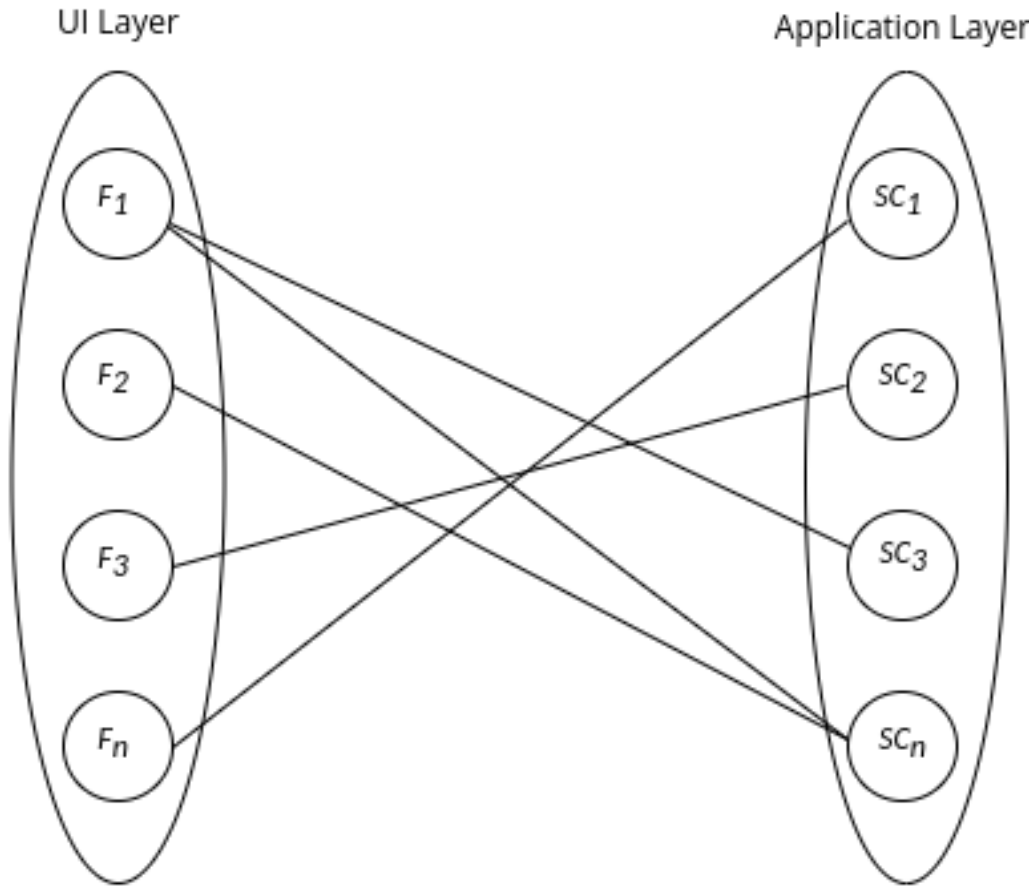


Figure 1.1: The mapping relationship between the software features and their relevant source code.

Figure 1.1 illustrates the mapping relation between the software features and their required source code for implementation. The software system may have several (F_n), where each (F) requires a certain Source Code (SC_n) entity to be implemented. There are three types of relation between the (F) and (SC), one-to-one, one-to-many, and many-to-many (Sinkala & Herold, 2021). For instance, Figure 1.1 shows one to one relation between (F_3), and (SC_2). While one to many is presented between (F_1) and (S_3), (SC_n). Note, the edges that connect the software features and the source code entities express the relationships between (F_n) and (SC_n) as illustrated in Figure 1.1.

1.1.1 The Applications of Feature Location

The term Feature Location (FL) is used to describe the process of identifying or pinpointing a section of code that executes a particular software function. This concept is applicable in various stages of software development, such as Software Comprehension (SC), Software Maintenance (SM), and Bug Localization (BL). Software developers aim to locate the necessary source code before undertaking tasks related to SM, SC, or BL. For instance, if a software system reports a defect in certain functionalities, programmers may employ the FL concept to streamline the process of locating the source code that might be responsible for the defect (Qayum, Khan, Akhunzada, et al., 2022). Hence, the term FL is relevant in different phases of the software engineering life cycle.

1.1.2 Type of Feature Location Techniques

The principle of FL can be utilized in every phase of software product development. The specific application of FL can be tailored according to the objectives of the software developers. Numerous Feature Location Tech-

niques (FLT) exist, with the goal of identifying at least one relevant source code entity through automated or semi-automated methods.

However, it's nearly impossible to locate the complete set of source code using just one configuration of FLT. Various strategies are used in FLT to find a starting point or a nearly complete feature location for source code entities (Razzaq et al., 2018). Each strategy can enhance the process of using FL, offering specific benefits to software developers in achieving their desired tasks. There are four common approaches to FLT (Buzaid & Hammad, 2021b; Rubin & Chechik, 2013), which are described as follows:

- Static Feature Location Techniques (SFLT) are designed to link the software feature with the appropriate source code by examining the dependencies within the source code. SFLT do not necessitate system compilation and instead investigate the system's control data flow. Developers need to identify starting points (either arbitrary or non-arbitrary), after which SFLT strive to locate relevant data based on static dependencies in the source code (Williams, Ren, De Carli, Lu, & Smith, 2021). For instance, a flow graph could be utilized to represent source code, and SFLT could traverse this graph to identify all accessible, relevant nodes and map them to a feature (Kazato, Hayashi, Kobayashi, et al., 2013).
- Textual Feature Location Techniques (TFLT) involve the use of significant textual data within the source code to identify the software feature. TFLT primarily employ three techniques: pattern matching, Information Retrieval (IR), and Natural Language Processing (NLP). For example, software developers might use a tool like `grep`³ or a text query to search the source code and retrieve the necessary source code for implementing the desired software feature (Saifan & Obei-

³<https://www.gnu.org/software/grep/>

dat, 2021). Hence, system compilation is not required to carry out TFLT's.

- **Dynamic Feature Location Techniques (DFLTs)** involve the use of gathered execution traces to display the source code relevant to executed software features (Michelon et al., 2022; Koschke & Quante, 2005; Wilde & Scully, 1995). In DFLT's, a scenario or test case, which is a series of actions that activate the software feature in the system, exercises the software feature. Developers need to execute a series of scenarios or test cases in DFLT's to generate the necessary execution traces, which can later be used to indicate the source code relevant to the displayed feature (Buzaid, Albalooshi, & Elmedany, 2022). In other words, DFLT's run the system's code by exercising the behaviors of the subject feature and monitoring runtime events such as object creation/deletion, method invocation, and thread creation/termination (Michelon, Linsbauer, Assunção, Fischer, & Egyed, 2021). The collected execution traces are used to locate the starting point or nearly complete set of source code that implements the software feature.
- **Hybrid Feature Location Techniques (HFLT's)** are a fusion of multiple FLT's. The idea of using a single FLT solution to achieve near-complete FL is not feasible (Razzaq et al., 2018; Eyal Salman, Seriai, & Dony, 2022). For instance, DFLT's can be utilized to identify an initial point (foothold) in the source code, which is then used to execute SFLT's, instead of starting from a random point in the code (Arwan, Rochimah, & Fatichah, 2022; Kazato, Hayashi, Kobayashi, et al., 2013; Savage, Revelle, & Poshyvanyk, 2010). Subsequently, another FLT can be employed to discover the majority of the necessary source code.

1.2 The Motivation of Using Dynamic Feature Location Techniques

In today's software project development landscape, open source software is widely adopted in both academia and the software industry, despite the difficulty of maintaining a stable team of software developers (Eghbal, 2020). Approximately 60% of the effort in the software life-cycle is dedicated to software maintenance activities (MILOUDI, CHEIKHI, ABRAN, & IDRI, 2022). A typical challenge arises when a new developer wishes to contribute to an open-source software system. For instance, the developer might be tasked with modifying, adding, or removing a specific functionality in the software system. The developer must first identify the feature before carrying out the requested software maintenance. Without appropriate tool support, finding the software feature can quickly become an insurmountable task. Hence, this work aims to assist developers in utilizing and choosing the necessary FLT, with a primary focus on enhancing the process of selecting the required DFLT.

FL can be accomplished through static, textual, or dynamic analysis of a software system. Dynamic analysis involves observing the system during its operation. DFLT involves conducting FL based on information obtained from execution traces (Li, Cao, Ma, Shen, & Wong, 2022). The context of these execution traces depends on the scenario or test suite that triggers the software feature (Bittner et al., 2021). Developers can extract data relevant to the displayed software feature using various techniques, such as execution trace slicing and sequential pattern mining (Sartipi & Safyallah, 2010). Therefore, static and textual analyses can be integrated with dynamic analysis to carry out hybrid DFLT.

Identifying the necessary source code for software feature maintenance can be a daunting task when using a static solution. Solely relying on SFLT

without analyzing the software’s runtime behavior may not offer comprehensive assistance to developers (Razzaq, Ventresque, Koschke, De Lucia, & Buckley, 2021). SFLT requires significant effort from developers to locate all dependencies in the source code. Developers strive to calculate all assignment relations between program identifiers of methods, variables, classes, and modules to implement SFLT. A call graph can be used to visualize the static dependencies in the source code to locate the software feature (Salis, Sotiropoulos, Louridas, Spinellis, & Mitropoulos, 2021). However, call graphs are typically large and difficult for developers to interpret. On the other hand, DFLT is needed to analyze only a portion of the source code. DFLT fundamentally relies on reverse engineering, where developers analyze the execution trace to map the displayed software feature to the relevant source code (Ghaleb, Alturki, & Aljasser, 2018). For an optimal solution, developers may combine SFLT and DFLT to locate a near-complete set feature. For instance, a call graph can be constructed based on the collected execution trace using DFLT to locate all dependencies in the source code.

TFLT can assist developers by conducting textual search analysis in the source code, but they are not capable of displaying call frequencies in the source code. Textual analysis involves searching the entire source code, which can be a complex task in a large program (Saifan & Obeidat, 2021). In contrast, dynamic analysis examines only a portion of the system (Michelon, Obermann, et al., 2021). It only analyzes the execution trace collected based on the execution of a specific scenario. However, the research community has acknowledged the necessity to integrate textual and dynamic techniques into hybrid techniques to enhance feature location effectiveness (Michelon, Linsbauer, et al., 2021). For instance, dynamic analysis monitors scenarios where system features are exercised, resulting in execution traces of code executed while exercising software features. In these collected traces,

textual analysis refines the traces by statically analyzing common artifacts and corresponding features. The goal of this textual analysis is to filter out source code that is associated with multiple features.

Indeed, the integration of SFLT, TFLT, DFLT enhances accuracy by leveraging the benefits of information retrieval, execution tracing, and dependency analysis pruning. Advanced Information Retrieval (IR) techniques such as Vector Space Modelling (VSM), Latent Dirichlet Allocation (LDA), and Latent Semantic Indexing (LSI) have proven highly effective in determining relevance based on the similarity between terms used in concern descriptions and program elements (Florez, Perry, Wei, & Marcus, 2022). Meanwhile, execution-tracing-based feature location techniques analyze the runtime behavior of a program to identify program elements activated when a concern is exercised (Hammoudi, Mayr-Dorn, Mashkoor, & Egyed, 2021). Finally, the static analysis aims to use the outputs of DFLT and TFLT to find the near-complete set of FL by locating all dependencies in the identified segments of the source code.

1.3 Monitoring the Software System at Runtime

Dynamic analysis involves gathering data from a system while it's running. In the context of DFLT, dynamic analysis is employed to pinpoint feature implementations. The process of feature location using dynamic analysis necessitates the inclusion of three components for all types of DFLT. These components are defined as follows:

- **Scenarios or Test Cases:** A scenario for a software feature refers to a series of actions that activate the feature within the system. These actions can be a sequence of user inputs that initiate system activities with observable outcomes, which are referred to as scenarios.

- **Instrumentation or Profiler Technique:** This refers to a method or solution that can be implemented on the software system to track and record all data invoked during the execution of the software feature by the scenario.
- **Execution Trace:** This refers to a finite sequence of events that occur during runtime. The logged data are represented as a text file when the software feature is exercised by the scenario. Therefore, the logged events are relied on both the scenarios and the instrumentation techniques used.

Figure 1.2 depicts an illustration of the necessary components to perform DFLT. The targeted software system should be instrumented to generate execution traces when a scenario exercises the software feature. Hence, the scenario refers to the required input to exercise the instrumented software feature. Note, each software feature can be exhibited by more than one scenario.

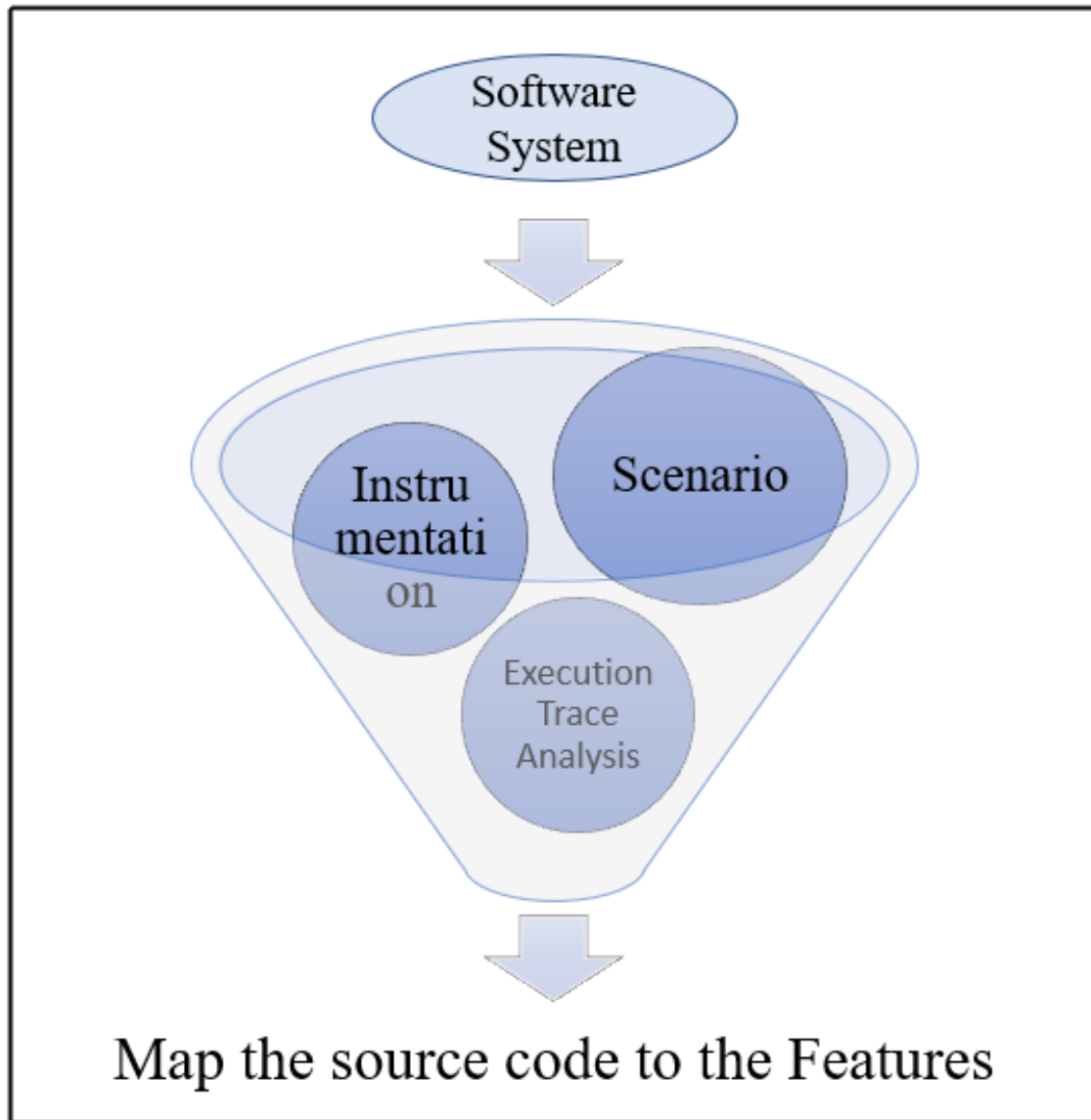


Figure 1.2: The main required elements to implement DFLT to map the source code to the relevant feature.

1.3.1 Formulating the Scenarios in Feature Location

Formulating a scenario is a pivotal element in the DFLT in software engineering. This process involves creating a specific use case or scenario that triggers the software feature of interest. By executing this scenario, we can monitor the system's behavior and record which parts of the code

are active when the feature is in use. The quality of the scenario directly influences the effectiveness of DFLT. A well-crafted scenario allows for a comprehensive and accurate identification of the code implementing the feature, thereby enhancing the overall quality of the DFLT. Therefore, the formulation of a precise and detailed scenario is an essential prerequisite for effective dynamic feature location.

For example, the developers are required to devise appropriate scenarios to exercise the functional software feature. Typically, one or more feature-specific scenarios are created to invoke the feature of interest (Martinez et al., 2018). In several DFLT, software programmers manually formulate two sets of scenarios (one of these scenarios exercises the software feature of interest and another one exercises an irrelevant feature). These scenarios are executed, and the relevant code is identified by comparing the differences between the set of executed units (Wilde & Scully, 1995; Koschke & Quante, 2005; Huh & Kim, 2021).

1.3.2 Instrumentation in Feature Location

Instrumentation in the context of computer programming refers to the measure of a product's performance to diagnose errors and write trace information. It can be of two types: source instrumentation and binary instrumentation. Instrumentation can be used for various purposes, such as profiling, inserting timers into functions, and logging major events like crashes.

Profiling is a technique that measures dynamic program behaviors during a training run with a representative input. It helps analyze properties of a program that cannot be analyzed statically with sufficient precision, such as alias analysis. Profiling can involve inserting timers into functions and logging major events like crashes.

Generally, in DFLT, instrumentation refers to the practice where developers insert logging code into the source code to record runtime information

before executing the scenario. As a result, when a scenario exercises the instrumented software feature, an execution trace is generated that includes all invoked events (Bogatinovski & Kao, 2023).

All DFLTs necessitate some form of instrumentation technique to profile the invoked methods, classes, and other granularities. However, the effectiveness of the profiled events hinges on the quality of the instrumented logging code. Log messages generated by high-quality logging code can significantly facilitate various log analysis tasks, such as event monitoring (Chen & Jiang, 2021).

Therefore, developers can configure the instrumentation technique to profile events at a specific level of granularity, such as classes or methods.

1.3.3 Execution Traces in Feature Location

In software engineering, an execution trace is a record of the sequence of events or operations that occur in a system during runtime. It is generated when a software program is executed and provides a detailed account of the activities within the system, such as method calls, variable changes, or system states. Execution traces are particularly useful in debugging and performance optimization, as they allow developers to track the flow of execution and identify any anomalies or inefficiencies. They are also used in various software analysis tasks, including feature location, program comprehension, and reverse engineering.

In reverse software engineering, execution traces play a crucial role in DFLT. Execution traces, which are records of the sequence of events that occur during the runtime of a system, provide valuable insights into the dynamic behavior of exercised features by a scenario.

When a feature is activated, an associated execution trace is produced, recording all the events triggered by that feature. Through analyzing these traces, developers can link the executed events back to their respective source

code, efficiently pinpointing the feature’s implementation. This methodology proves especially valuable in extensive and intricate software systems, where manual feature identification may be both time-intensive and prone to errors.

Moreover, execution traces can be used in conjunction with static and textual analysis techniques to improve the accuracy and comprehensiveness of DFLT’s. This combination forms what is known as HFLT’s, which leverage the strengths of multiple analysis methods to provide a more effective approach to feature location in reverse software engineering (Abbasli & Ganiz, 2021; Whitworth, 2022).

1.4 The Research Gap

The examination of three existing papers (Cornelissen, Zaidman, Van Deursen, Moonen, & Koschke, 2009; Dit, Revelle, Gethers, & Poshyvanyk, 2013; Rubin & Chechik, 2013) that describe FLT’s, along with a subsequent literature review, indicated that DFLT’s have been somewhat utilized in FLT’s. Specifically, among the DFLT’s papers identified by (Dit, Revelle, Gethers, & Poshyvanyk, 2013), 26 out of 60 papers employed dynamic analysis to implement FLT’s.

An additional literature review conducted by (Buzaid & Hammad, 2021b) identified 32 more DFLT’s papers published between January 2012 and December 2020 (refer to Section 2.4). Out of these 32, three papers exclusively used DFLT’s, leveraging only execution traces to apply FL. Among the remaining 29 papers, four unique approaches were discovered that combined textual analysis with dynamic analysis to apply FL. The literature search was then broadened to include relevant papers on static analysis, revealing two more approaches that utilized execution traces for FL.

Despite the significant advancements in the use of software execution

traces for feature location in software systems, there remains a notable research gap. The primary challenge lies in the complexity and volume of execution traces, especially in large-scale software systems. These traces rely on the scenario or test case input that are used to exercise the software features.

Furthermore, the effectiveness of using execution traces for feature location heavily depends on the quality of test suites or scenarios used to generate these traces. Inadequate or poorly designed test suites may not invoke all aspects of a feature, leading to incomplete or misleading execution traces. DFLT requires suitable test suites or the creation of appropriate scenarios to invoke only the traces related to a specific feature (Michelon, Linsbauer, et al., 2021).

In addition, while DFLTs are effective, the quality of their outcomes is contingent on analysts preparing adequate scenarios to exercise the features (Hayashi et al., 2016). However, the literature has not sufficiently described how to formulate accurate scenarios or test cases in DFLTs. Specifically, existing approaches appear to use inappropriate or randomly arbitrary scenarios, often based entirely on the developer's familiarity with the system, to exercise the feature of interest (Zhao, Zhang, Liu, Sun, & Yang, 2006; Kazato, Hayashi, Kobayashi, et al., 2013). In other words, analysts prepare initial features, scenarios, and a mapping between them based on their current understanding of the system under discussion.

Addressing these challenges could greatly enhance the effectiveness and accuracy of feature location in software systems.

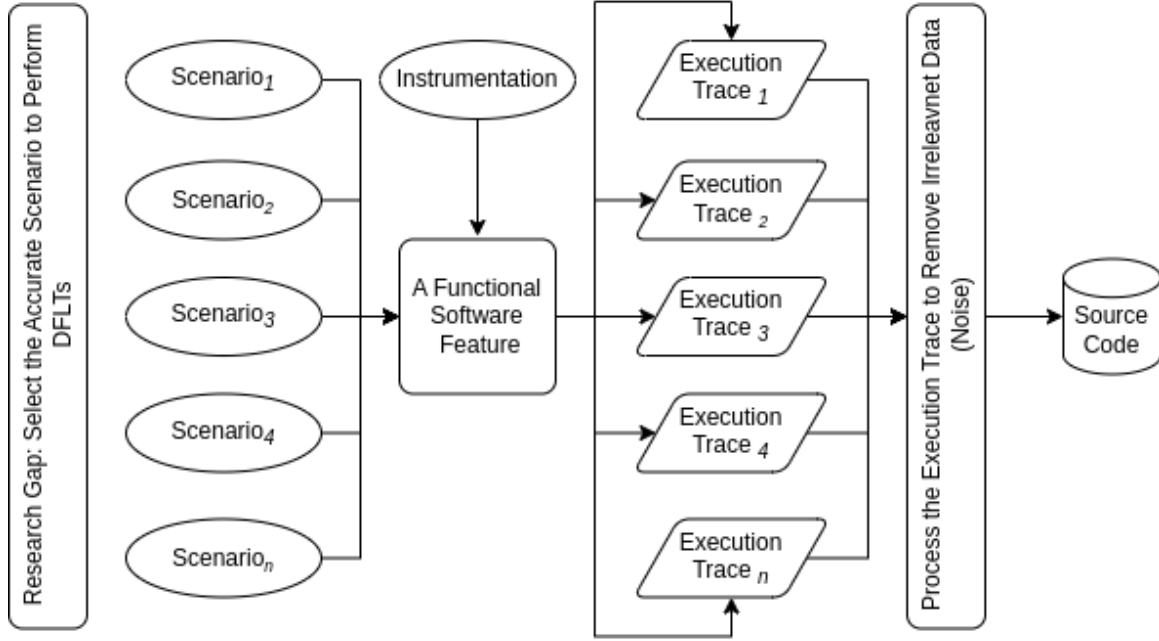


Figure 1.3: The focus of this research.

In conclusion, as depicted in Figure 1.3, there is a significant research gap at a high level in formulating the accurate scenario to exercise the software feature. An instrumented functional software feature can be demonstrated by several possible scenarios, and each scenario execution generates a relevant execution trace. The content of the execution trace depends on the scenario input. The accuracy of the output of using DFLT to map the source code to the exhibited feature relies on the content of the execution trace, which in turn depends on the scenario input.

1.5 The Research Objectives

In this dissertation, we introduce an innovative technique to aid programmers in the creation, formulation, or reformulation of the necessary scenarios or test cases for applying DFLT. Our method is grounded in the analysis of execution traces, a process we have termed Aggregation of Execution Traces to Formulate a Scenario (AETFS). This process involves the collec-

tion, aggregation, and analysis of execution traces from a software system using a set of scenarios / test cases. The execution traces gathered comprise millions of detailed low-level execution events, which are obtained by instrumenting real-world applications and documenting updates to the internal state and method calls. To utilize the information stored in the program trace database, we present several high-level query interfaces. These interfaces enable a software developer using our tool to express their programming intent and acquire the necessary scenario or test case for conducting DFLTs.

To achieve the main objective, the following sub-goals were established:

- The goal is to develop a method to standardize the implementation of DFLTs to address the scarcity issue in formulating accurate inputs to exercise the software feature. The resulting approach is known as AETFS.
- To conduct an empirical evaluation and fine-tune AETFS by determining the most effective configuration of the method.
- To compare a set of optimized configuration AETFS against state-of-the-art baseline DFLTs, TDFLTs, and SDFLTs.

The comparison of the best-performing configuration of the AETFS to the set of baselines identified a contradictory result: half of the subject system in the experiment responded better to AETFS, and the other half responded better without using AETFS. Two post-hoc objectives were formulated based on this finding:

- To characterize the two sets of the subject systems to determine any differences related to their response to AETFS and existing solutions in DFLTs, DFLTS, and SDFLTs.

- To evaluate the potential for hybridization between the set of configurations in AETFS.

1.6 The Research Questions

The research question section of this thesis is divided into two integral subsections, both of which are focused on addressing a key challenge in DFLT: the formulation of the required scenarios. The first subsection presents a series of research questions that delve into this problem, exploring the intricacies of scenario formulation and its impact on the application of DFLTs. These questions lay the groundwork for our investigation, highlighting the areas that need attention and improvement. The second subsection builds on this foundation by proposing potential solutions to each research question. These proposals outline various methodologies, metrics, and strategies that could be employed to enhance the accuracy and efficiency of scenario formulation in DFLTs. Together, these subsections provide a comprehensive approach to tackling the central challenge of scenario formulation in DFLTs.

1.6.1 Questions

The Goal-Question-Metric (GQM) approach is a structured method for defining and interpreting software metrics, ensuring that measurement activities are aligned with specific goals. Developed in the early 1980s and piloted at NASA Goddard Space Flight Center, GQM involves establishing goals, formulating questions to refine these goals into information needs, and identifying metrics to answer these questions (Caldiera & Rombach, 1994). This approach provides traceability between measurements and organizational goals, fostering a culture of continuous improvement (Caldiera & Rombach, 1994). In my research, I employ the GQM approach to systematically ad-

dress the following research question, ensuring that the metrics used are directly tied to the overarching goals of the study.

The following research question encompasses the direction of this research:

Main Goal: Optimize scenarios or test cases to effectively exhibit software features to assist DFLTs.

Main Research Question How can we measure the effectiveness of a scenario or test case in exhibiting a software feature?

- **Metric 1** Number of software features accurately exhibited by the scenario or test case.
- **Metric 2** The size of the collected execution traces.

The more detailed sub-questions answered in this work include:

Goal 1: Effectively formulate scenarios or test cases to exercise specific software features in DFLTs.

RQ1 How can we measure the effectiveness of a formulated scenario or test case in exercising a specific software feature?

- **Metric 1.1** Number of software features accurately exercised by the scenario or test case.
- **Metric 1.2** The size of the collected execution traces.

Goal 2: Understand the impact of the accuracy of scenario formulation on the performance of DFLTs.

RQ2 How can we measure the accuracy of a formulated scenario or test case?

- **Metric 2.1** Number of retrieved relevant methods / classes that are invoked when the targeted software feature is exercised by the scenario or test case.

- **Metric 2.2** The size of the irrelevant retrieved data on the collected execution traces.

Goal 3: Optimize the utilization of execution traces in the process of feature location when using accurately formulated scenarios or test cases.

RQ3 How can we measure the optimal utilization of execution traces in the process of feature location?

- **Metric 3.1** The size of the collected execution traces comparing with the dataset.
- **Metric 3.2** the size of the located foothold in the execution trace comparing to the near-full-set feature in the execution trace.

The results from research question 3, suggested that some systems react well to AETFS and others do not. This suggests a characterization of systems in terms of the DFLT's that be appropriately applied to them and that dynamic approaches might not be beneficial. Hence, the following research questions assess:

Goal 4: Identify the challenges and potential solutions in formulating scenarios or test cases for DFLT's in large and complex software systems.

RQ4 What are the challenges in formulating scenarios or test cases for DFLT's in large and complex software systems and what are the potential solutions to these challenges?

- **Metric 4.1** Number of identified challenges, and solution.
- **Metric 4.2** Severity of each identified challenge, and solution.
- **Metric 4.3** Frequency of occurrence of each identified challenge and the effectiveness of each identify solution.

Goal 5: Improve the effectiveness and efficiency of scenario formulation strategies in DFLT's.

RQ5 How can we measure the effectiveness and efficiency of different scenario formulation strategies in DFLTs?

- **Metric 5.1** Scenario accuracy, the degree to which the scenarios ability to retrieve the required methods/classes in that related the exhibited software feature.
- **Metric 5.2** Scenario coverage, the extent of possible data that are covered by the formulated scenarios.

Goal 6: Develop tools or techniques to assist developers in formulating accurate scenarios or test cases for DFLTs.

RQ6 What tools or techniques can be developed to assist developers in formulating accurate scenarios or test cases for DFLTs?

- **Metric 6.1** Tool/technique effectiveness, this could be measured by the increase in accuracy of the scenarios or test cases formulated using the developed tools or techniques.
- **Metric 6.2** Ease of use, this could be measured by conducting user studies to understand how easy it is for developers to use the developed tools or techniques.

Goal 7: Ensure the proposed solution to formulate the scenario test case is being able to cover all aspects of the exercised feature.

RQ7 How can we ensure that the formulated scenarios or test cases cover all aspects of the exercised feature?

- **Metric 7.1** Coverage metric: this could be measured by the percentage of feature aspects that are covered by the formulated scenarios or test cases.
- **Metric 7.2** Quality metric: this could be measured by the number of defects found during testing that were not covered by the initial scenarios or test case.

RQ8: What role does developer expertise play in scenario formulation and how can this be incorporated into the DFLT process?

Goal 8: Incorporate developer expertise into the DFLT process to enhance scenario formulation.

RQ8 What role does developer expertise play in scenario formulation and how can this be incorporated into the DFLT process?

- **Metric 8.1** Expertise level: measure the level of expertise of the developers involved in the scenario formulation.
- **Metric 8.2** Quality of scenarios: assess the quality of scenarios formulated by developers with varying levels of expertise. This could be measured by the accuracy of the scenarios in predicting outcomes (dataset benchmark).
- **Metric 8.3** Coverage of scenarios: evaluate the extent to which the scenarios formulated by developers with varying levels of expertise cover all aspects of the exhibited feature.

Goal 9: Leverage ML or artificial intelligence to automate or enhance the scenario formulation process in DFLTs.

RQ9 How can ML or Artificial Intelligence (AI) be leveraged to automate or enhance the scenario formulation process in DFLTs?

- **Metric 9.1** Automation level: measure the extent to which the scenario formulation process is automated.
- **Metric 9.2** Enhancement level: assess the improvement in the quality of scenarios formulated using ML or AI.
- **Metric 9.3** Coverage: evaluate the extent to which the scenarios formulated using ML or AI cover all aspects of the targeted software feature.

Goal 10: Standardize best practices for scenario formulation in DFLT's across different projects or teams.

RQ10 What are the best practices for scenario formulation in DFLT's and how can these be standardized across different projects or teams?

- **Metric 10.1** Adoption rate: measure the percentage of AETFS solution that adopt the standardized best practices.
- **Metric 10.2** Consistency: Assess the degree of consistency in scenario formulation across different projects or software systems.
- **Metric 10.3** Quality of scenarios: evaluate the quality of scenarios formulated using the standardized best practices in retrieving the related method/classes for the exercised software feature.

1.6.2 Initial Proposed Solution to Answer Each Research Question

This subsection addresses each research question in the thesis, focusing on DFLT's. We propose initial solutions for scenario formulation and explore how this can improve the accuracy of mapping source code to features.

Proposal RQ1: To address Goal 1 and answer RQ1, we propose the following approach:

1. **Formulation of Scenarios/Test Cases:** We will develop a set of scenarios or test cases specifically designed to exercise the targeted software features in DFLT's. These scenarios will be created based on the functional requirements and expected behavior of the software features.
2. **Execution and Data Collection:** The formulated scenarios or test cases will be executed, and the resulting execution traces will be collected. These traces will provide valuable data about the interaction

of the software features during the execution of the scenarios or test cases.

3. **Evaluation of Effectiveness:** The effectiveness of the scenarios or test cases will be evaluated using the following metrics:

- **Metric 1.1:** The number of software features accurately exercised by the scenario or test case. This will provide a quantitative measure of how well the scenario or test case is able to exercise the targeted features.
- **Metric 1.2:** The size of the collected execution traces. This will give an indication of the complexity of the interactions among the software features during the execution of the scenario or test case.

By following this approach, we aim to provide a systematic and measurable way to assess the effectiveness of formulated scenarios or test cases in exercising specific software features in DFLTs. This will contribute to the overall goal of improving the accuracy and efficiency of feature location in source code.

Proposal RQ2: To address Goal 2 and answer RQ2, we propose the following approach:

1. **Formulation of Scenarios/Test Cases:** We will create scenarios or test cases designed to exercise specific software features in DFLTs. These will be based on the functional requirements and expected behavior of the software features.
2. **Execution and Data Collection:** We will execute the formulated scenarios or test cases and collect the resulting execution traces. These traces will provide data about the interaction of the software features during the execution.

3. **Evaluation of Accuracy:** The accuracy of the scenarios or test cases will be evaluated using the following metrics:

- **Metric 2.1:** The number of retrieved relevant methods/classes invoked when the targeted software feature is exercised by the scenario or test case. This will provide a quantitative measure of the accuracy of the scenario or test case in exercising the targeted features.
- **Metric 2.2:** The size of the irrelevant retrieved data on the collected execution traces. This will give an indication of the amount of noise in the data collected during the execution of the scenario or test case.

By following this approach, we aim to understand the impact of the accuracy of scenario formulation on the performance of DFLT. This will contribute to the overall goal of improving the accuracy and efficiency of feature location in source code.

Proposal RQ3: To address Goal 3 and answer RQ3, we propose the following approach:

1. **Formulation of Scenarios/Test Case:** We will create scenarios or test cases designed to exercise specific software features in DFLT. These will be based on the functional requirements and expected behavior of the software features.
2. **Execution and Data Collection:** We will execute the formulated scenarios or test cases and collect the resulting execution traces. These traces will provide data about the interaction of the software features during the execution.
3. **Evaluation of Optimal Utilization:** The optimal utilization of execution traces in the process of feature location will be evaluated using

the following metrics:

- **Metric 3.1:** The size of the collected execution traces compared with the dataset. This will provide a quantitative measure of the amount of data collected during the execution of the scenario or test case relative to the total available data.
- **Metric 3.2:** The size of the located foothold in the execution trace compared to the near-full-set feature in the execution trace. This will give an indication of the precision of the feature location process.

By following this approach, we aim to optimize the utilization of execution traces in the process of feature location when using accurately formulated scenarios or test cases. This will contribute to the overall goal of improving the accuracy and efficiency of feature location in source code.

Proposal RQ4: To address Goal 4 and answer RQ4, we propose the following approach:

1. **Identification of Challenges:** We will conduct a comprehensive review of existing literature and perform an in-depth analysis of large and complex software systems to identify the challenges in formulating scenarios or test cases for DFLTs.
2. **Proposing Potential Solutions:** For each identified challenge, we will propose potential solutions based on best practices, expert opinions, and innovative ideas.
3. **Evaluation of Challenges and Solutions:** The challenges and solutions will be evaluated using the following metrics:
 - **Metric 4.1:** The number of identified challenges and solutions. This will provide a quantitative measure of the breadth of the issues addressed.

- **Metric 4.2:** The severity of each identified challenge and the effectiveness of each solution. This will provide an indication of the impact of the challenges and the potential benefits of the solutions.
- **Metric 4.3:** The frequency of occurrence of each identified challenge and the effectiveness of each identified solution. This will give an indication of the prevalence of the challenges and the practicality of the solutions.

By following this approach, we aim to identify the challenges and potential solutions in formulating scenarios or test cases for DFLTs in large and complex software systems. This will contribute to the overall goal of improving the accuracy and efficiency of feature location in source code.

Proposal RQ5: To address Goal 5 and answer RQ5, we propose the following approach:

1. **Development of Scenario Formulation Strategies:** We will develop a variety of scenario formulation strategies for DFLTs. These strategies will be designed to effectively and efficiently exercise specific software features.
2. **Application of Strategies and Data Collection:** We will apply these strategies to formulate scenarios or test cases, execute them, and collect the resulting execution traces. These traces will provide data about the interaction of the software features during the execution.
3. **Evaluation of Effectiveness and Efficiency:** The effectiveness and efficiency of the scenario formulation strategies will be evaluated using the following metrics:
 - **Metric 5.1:** Scenario accuracy, which measures the degree to which the scenarios are able to retrieve the required methods/-

classes related to the exhibited software feature. This will provide a quantitative measure of the accuracy of the scenario formulation strategies.

- **Metric 5.2:** Scenario coverage, which measures the extent of possible data that are covered by the formulated scenarios. This will give an indication of the comprehensiveness of the scenario formulation strategies.

By following this approach, we aim to improve the effectiveness and efficiency of scenario formulation strategies in DFLTs. This will contribute to the overall goal of improving the accuracy and efficiency of feature location in source code.

Proposal RQ6: To address Goal 6 and answer RQ6, we propose the following approach:

1. **Development of Tools/Techniques:** We will design and develop tools or techniques that can assist developers in formulating accurate scenarios or test cases for DFLTs. These tools or techniques could include automated scenario generators, scenario validation tools, or advanced feature mapping algorithms.
2. **Application and Evaluation of Tools/Techniques:** We will apply these tools or techniques in the process of scenario formulation and evaluate their effectiveness and ease of use. This will involve formulating scenarios or test cases using the developed tools or techniques, executing them, and assessing the accuracy of the resulting scenarios or test cases.
3. **Evaluation of Tools/Techniques:** The effectiveness and ease of use of the developed tools or techniques will be evaluated using the following metrics:

- **Metric 6.1:** Tool/technique effectiveness, which could be measured by the increase in accuracy of the scenarios or test cases formulated using the developed tools or techniques. This will provide a quantitative measure of the effectiveness of the tools or techniques.
- **Metric 6.2:** Ease of use, which could be measured by conducting user studies to understand how easy it is for developers to use the developed tools or techniques.

By following this approach, we aim to develop tools or techniques that can assist developers in formulating accurate scenarios or test cases for DFLTs. This will contribute to the overall goal of improving the accuracy and efficiency of feature location in source code.

Proposal RQ7: To address Goal 7 and answer RQ7, we propose the following approach:

1. **Formulation of Scenarios/Test Cases:** We will develop scenarios or test cases for DFLTs that aim to cover all aspects of the exercised feature. These scenarios or test cases will be based on a comprehensive understanding of the feature's functionality, behavior, and interactions with other features.
2. **Execution and Data Collection:** We will execute these scenarios or test cases and collect the resulting execution traces. These traces will provide data about the interaction of the software features during the execution.
3. **Evaluation of Coverage and Quality:** The coverage and quality of the scenarios or test cases will be evaluated using the following metrics:
 - **Metric 7.1:** Coverage metric, which could be measured by the

percentage of feature aspects that are covered by the formulated scenarios or test cases. This will provide a quantitative measure of the comprehensiveness of the scenarios or test cases.

- **Metric 7.2:** Quality metric, which could be measured by the number of defects found during testing that were not covered by the initial scenarios or test case. This will provide an indication of the quality of the scenarios or test cases and their ability to detect defects.

By following this approach, we aim to ensure that the formulated scenarios or test cases cover all aspects of the exercised feature. This will contribute to the overall goal of improving the accuracy and efficiency of feature location in source code.

Proposal RQ8: To address Goal 8 and answer RQ8, we propose the following approach:

1. **Incorporation of Developer Expertise:** We will incorporate developer expertise into the DFLT's process. Developers with varying levels of expertise will be involved in the formulation of scenarios or test cases. Their insights, knowledge, and experience will be leveraged to enhance the quality and coverage of the scenarios or test cases.
2. **Evaluation of Developer Expertise and Scenario Quality:** The role of developer expertise in scenario formulation will be evaluated using the following metrics:
 - **Metric 8.1:** Expertise level, which measures the level of expertise of the developers involved in the scenario formulation. This will provide a quantitative measure of the developers' knowledge and experience.

- **Metric 8.2:** Quality of scenarios, which assesses the quality of scenarios formulated by developers with varying levels of expertise. This could be measured by the accuracy of the scenarios in predicting outcomes (dataset benchmark). This will provide an indication of the effectiveness of incorporating developer expertise into the DFLT's process.
- **Metric 8.3:** Coverage of scenarios, which evaluates the extent to which the scenarios formulated by developers with varying levels of expertise cover all aspects of the exhibited feature. This will provide a measure of the comprehensiveness of the scenarios or test cases.

By following this approach, we aim to incorporate developer expertise into the DFLT's process to enhance scenario formulation. This will contribute to the overall goal of improving the accuracy and efficiency of feature location in source code.

Proposal RQ9: To address Goal 9 and answer RQ9, we propose the following approach:

1. **Leveraging ML/AI** We will leverage ML or AI techniques to automate or enhance the scenario formulation process in DFLT's. This could involve using ML algorithms to predict effective scenarios, or AI techniques to generate scenarios that cover a wide range of feature aspects.
2. **Application and Evaluation of ML/AI:** We will apply these ML/AI techniques in the process of scenario formulation and evaluate their effectiveness in terms of automation, enhancement, and coverage. This will involve formulating scenarios using the developed ML/AI techniques, executing them, and assessing the quality and coverage of the resulting scenarios.

3. **Evaluation of ML/AI Techniques:** The effectiveness of the ML/AI techniques will be evaluated using the following metrics:

- **Metric 9.1:** Automation level, which measures the extent to which the scenario formulation process is automated using ML/AI. This will provide a quantitative measure of the level of automation achieved.
- **Metric 9.2:** Enhancement level, which assesses the improvement in the quality of scenarios formulated using ML/AI. This will provide an indication of the effectiveness of the ML/AI techniques in enhancing the scenario formulation process.
- **Metric 9.3:** Coverage, which evaluates the extent to which the scenarios formulated using ML/AI cover all aspects of the targeted software feature. This will provide a measure of the comprehensiveness of the scenarios formulated using ML/AI.

By following this approach, we aim to leverage ML or AI to automate or enhance the scenario formulation process in DFLT's. This will contribute to the overall goal of improving the accuracy and efficiency of feature location in source code.

Proposal RQ10: To address Goal 10 and answer RQ10, we propose the following approach:

1. **Identification of Best Practices:** We will conduct a comprehensive review of existing literature, case studies, and expert opinions to identify the best practices for scenario formulation in DFLT's.
2. **Standardization of Best Practices:** We will develop a standardized set of best practices that can be applied across different projects or

teams. This could involve creating guidelines, checklists, or templates for scenario formulation that incorporate the identified best practices.

3. **Implementation and Evaluation of Best Practices:** We will implement these standardized best practices in different projects or teams and evaluate their effectiveness in terms of adoption rate, consistency, and quality of scenarios. This will involve formulating scenarios using the standardized best practices, executing them, and assessing the quality and consistency of the resulting scenarios.

4. **Evaluation of Standardized Best Practices:** The effectiveness of the standardized best practices will be evaluated using the following metrics:

- **Metric 10.1:** Adoption rate, which measures the percentage of AETFS solutions that adopt the standardized best practices. This will provide a quantitative measure of the acceptance and implementation of the best practices.
- **Metric 10.2:** Consistency, which assesses the degree of consistency in scenario formulation across different projects or software systems. This will provide an indication of the uniformity and standardization achieved through the best practices.
- **Metric 10.3:** Quality of scenarios, which evaluates the quality of scenarios formulated using the standardized best practices in retrieving the related methods/classes for the exercised software feature. This will provide a measure of the effectiveness of the best practices in improving the quality of scenarios.

By following this approach, we aim to standardize best practices for scenario formulation in DFLTs across different projects or teams. This will contribute to the overall goal of improving the accuracy and efficiency of

feature location in source code.

1.7 Contributions

The original contributions of this work to the existing knowledge include:

- A novel approach to select the required input to apply DFLTs - AETFS.
- Empirically determining the best configuration of AETFS with respect to the selected instrumentation technique, and execution trace analysis.
- Partially characterizing its applicability in term of system characterises.
- The implementation of AETFS, its configuration, the hybrid approach, and accumulating existing work in DFLTs, TDFLTs, and SDFLTs as open-source tools, available to other researchers.

The specific findings of this work suggest that certain configurations of AETFS result in a better performance. A broader implication (and contribution) for DFLTs research is in need to optimize the DFLTs' input (scenarios or test cases), because, as was the case in this work, selected the optimal scenario or test cases to exhibit the software feature and generate the required execution trace may significantly enhance the performance of the DFLT. Currently, and in contrast, the research direction of DFLT has adjusted towards combinations of approaches and new approaches like query expansion. This work suggests that when implementing such approaches, practitioners should also concentrate on determining the most optimal scenario to perform the task DFLT.

Likewise, it was also found that characteristics of the scenarios or test cases have a significant impact on the effectiveness of the new approach. Currently, DFLTs employ "one-approach-fits-all" strategy when selecting the required scenario in the DFLT. The finding of this work implies, that it could be valuable for future research to characterize software systems toward picking more optimal DFLT for the system (or the most optimal instrument configuration of such DFLT) to efficiently work with that particular type of the software.

1.8 Thesis Outline

This work is organized in the following structure:

- In Chapter 2 summarizes the related work and identifies the research gap. The chapter begins with an introduction to the literature review research gap. The reviewed material covers the definitions and background of FL, the input of DFLT, instrumentation techniques in DFLT, and the type of analysis used in DFLT. The review section includes a systematic literature review protocol that covers publications in DFLT between 2013 and 2020. Additionally, a taxonomy of DFLT is presented with examples for each category, focusing on TD-FLT and SDFLT in particular. A discussion of the trends in DFLT going forward and how they are evaluated is also included. The chapter concludes with a discussion of more specifically related work: the input paucity problem in DFLT and how standardizing the process of DFLT has been a problem in past research work.
- Chapter 3 of this thesis introduces a new method that uses execution traces to aggregate data from the instrumented system. The method is called AETFS and is described in detail. The chapter begins by pre-

senting the significant steps of the approach, followed by an in-depth description of each step. The chapter then describes the difficulties and peculiarities of filtering execution traces using advanced IR techniques such as LSI, LDA, and VSM. Finally, the chapter concludes with an elaboration of implementation details.

- In Chapter 4, it explores the experiments and case studies conducted for AETFS. The first experiment and case study focuses on using text queries to retrieve information related to the exercised feature. The second case study and experiment demonstrate how execution traces can be divided into segments. It provides details on how developers can exclude irrelevant segments to reformulate the required scenario for optimal DFLT. Lastly, the third experiment and case study investigates the effectiveness of using UML diagrams in the AETFS approach. Each experiment and case study is accompanied by details that illustrate how practical work is implemented, along with a guideline for replicating the work in further research investigations.
- In Chapter 5, the results of Chapter 4 are discussed to demonstrate the most effective configuration of AETFS. The best-performing configuration (subsection 4.2) was compared against TDFLTs (subsection 4.1) and SDFLTs (subsection 4.3). The analysis revealed a dichotomy in the results, with some systems responding better to AETFS and others to the baseline systems. Consequently, the subject systems of the experiment were characterized based on this dichotomy. The discovery of this dichotomy also prompted the proposal to combine the three approaches: AETFS, TDFLTs, and SDFLTs. The best-performing configuration of AETFS is used in conjunction with the TDFLTs approach to evaluate their combined effectiveness. This chapter presents the results, their analysis, and a detailed discussion.

- Chapter 6 is a crucial part of any research work. It serves as a summary of the entire study, highlighting the key contributions and findings. In this chapter, the results are generalized to draw broader conclusions and implications for academic and industrial applications. Additionally, it provides directions for future work and identifies potential threats to the validity of the findings.

Chapter 2

Literature Review

2.1 Literature Review

Many works have been published to illustrate the recent research activities in Feature Location (FL). This chapter provides the readers with a comprehensive overview of the literature review of the well-known papers in Feature Location Techniques (FLT). Mainly focus on dynamic analysis. The search scope included articles produced up to, and including, 2020. The literature review was categorized into several sections, starting from the background of the research and ending with a conclusion about the finding in Dynamic Feature Location Techniques (DFLTs).

The background of the research was introduced in the first section. The section starts with the definition of FL. It defined each category of FL, such as DFLT, Static Feature Location Techniques (SFLT), Textual Feature Location Techniques (TFLT), and Hybrid Feature Location Techniques (HFLT). Next, the section includes a brief background of DFLT com-

ponents.

The second section funnels to more specifically related work: the vocabulary paucity problem in DFLT. It described how the existing materials had been leveraged to address the DFLT problem.

The identified materials were reviewed in the third section. The section starts with a taxonomy of FLT with concrete examples for each category. Next, the DFLT and their underlying analysis were focused on. This is followed by a section to conclude with a discussion of how DFLT were evaluated and highlights the trends in type techniques from now on.

The conclusion was presented in the last section of the literature review chapter. It showed the reader a summary of the research finding in DFLT. Mainly focused on the current problem in the research domain and presented a possible solution.

2.2 Dynamic Feature Location Background

To present the reader with the area of DFLT, we have to define FL first and show the main categories of FL. Figure 2.1 illustrates the architecture of the DFL background. The Full Dynamic Feature Location Techniques (FDFLT) are shown in Figure 2.1(a), using test cases or scenarios to generate execution traces, then map the source code to the relevant feature based on the execution trace.

However, Figure 2.1(b) shows the architecture of the Hybrid Dynamic Feature Location Techniques (HDFLT) background. It uses both execution trace and static source code to perform FL. Query input is used on the static approach to search in the source code, and scenarios or test cases are used to exhibit the instrumented software feature on the dynamic side.

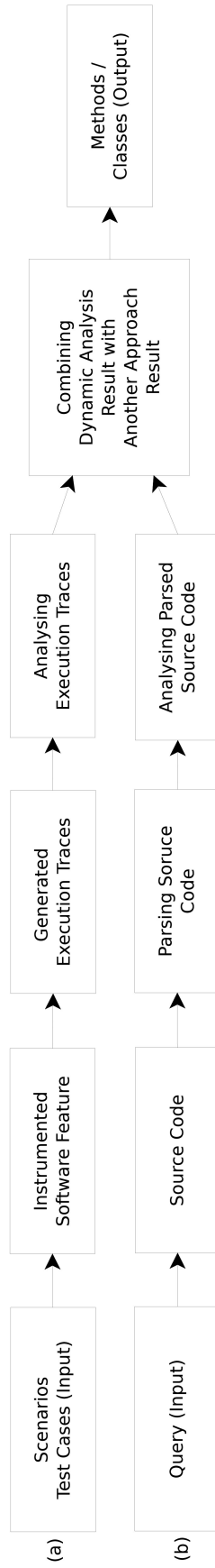


Figure 2.1: General steps to implement DFLT.

2.2.1 Definitions of Feature Location

When the developers maintain existing software, they intuitively map its features to the relevant source code to process the maintenance. Formal definitions have to define mapping features to the source code. Dit *et al.* (Dit, Revelle, Gethers, & Poshyvanyk, 2013) defined the FL as locating the initial point in the source code used to implement a functional feature in the software system. Razzaq *et al.* (Razzaq et al., 2020) locate two aims of FLT, a foothold to locate an initial point of the feature in the source code and a near-full feature to locate all the source code required to update or construct the required feature. FL is classified into three major types: Static Feature Location Techniques (SFLT), Textual Feature Location (TFLT), and DFLT.

SFLT aims to map source code to the software feature based on the structural dependencies of the source code component. For example, a point in the source code, such as a method, can be selected as a foothold in the source code to represent a feature. Static analysis inspects all outgoing calls and all incoming calls for that selected method to construct a near-full set of elements in the source code.

TFLT utilizes the meaningful textual information in the source code to introduce software features in the source code. For example, meaningful methods identifier names, meaningful comments in the source code, and meaningful folder names can be used by lexical analysis to provide a feature concept in the source code based on the textual description.

DFLTs use a scenario or test case to execute the desired feature of the software system to generate an execution trace. It utilizes the profiled data on execution trace to perform FL. For example, the execution trace can be analyzed to locate all invoked methods when exhibiting a feature to determine all appropriate techniques required to construct the exercised feature.

2.2.2 Input of DFLTs

Developers can use DFLTs during the software maintenance phase. There are several forms of input in DFLTs. Some standard input types can be a test case, a scenario, a query, a source code artifact, or any combination of these. The scenario is a set of steps that are carried out to exhibit a desired functional feature in the instrumented software system to generate an execution trace. The query in DFLTs can be used as a set of words in natural language to retrieve events from the execution trace or the source code based on the execution trace. A source code artifact can be any code, such as methods or classes. For instance, any level of the code element is used as an input in DFLTs to locate the relevant source code or the appropriate feature.

However, in the context of DFLTs, Behavior Driven Development (BDD) can be utilized to formulate scenarios for exercising software features. Leveraging the principles from the paper “Predicting Co-Changes between Functionality Specifications and Source Code in Behavior Driven Development” (A. Z. Yang, da Costa, & Zou, 2019), we can create a bridge between feature specifications and source code (Savage et al., 2010). The Gherkin language, a cornerstone of BDD, can be dynamically used to write human-readable scenarios that describe the behavior of the software feature (Koschke & Quante, 2005). These scenarios, written in Given-When-Then format, serve as both documentation and automated tests. For instance, a developer unfamiliar with the software system could write a Gherkin scenario to describe a feature’s behavior, then use DFL techniques to map this high-level description to specific source code (Revelle & Poshyvanyk, 2009). This approach enhances traceability and understanding of the system, particularly for developers new to the project.

2.2.3 Instrumentation Techniques in DFLTs

All types of DFLTs require profiling or instrumentation techniques to generate execution traces when features are exhibited by scenarios or test cases. Instrumentation in DFLTs refers to using class load to trace the data on the byte code level. Java Platform Debugger Architecture (JPDA)¹ can be used as an instrumentation tool to collect traces when a scenario exhibits a feature. Marked traces can be generated by using the JPDA profiler. Java Virtual Machine (JVM) is launched when JPDA is initiated to trace the Java program. Another Java-based instrumentation technique is called Java Agent². It allows tracing the invoked methods at the byte-code level for the existing compiled classes.

Profiler in DFLTs refers to using a source code-level probe to generate an execution trace. Markup tags-based instrumentation called srcML (Collard & Maletic, 2016) can be used to create execution traces. The srcML layout is an XML characterization for software source code, where the markup tags link feature of the abstract syntax for the programming language. The srcML program is defined as a command-line application for turning source code to srcML, an interface for examining, inspecting, and controlling source code in this form and altering srcML back to source code. The contemporary srcML supports C/C++, C#, and Java.

2.2.4 Type of Analysis in DFLTs

DFLTs are based on dynamic analysis relying on the execution trace to locate the feature on the source code. It is performed by analyzing the invoked events on the execution trace when a scenario or test case exercises the feature. The execution traces are generated by instrumenting the source code or

¹<https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/>

²<https://www.baeldung.com/java-instrumentation>

using a profiling tool. Generated execution traces can be compared against each other to filter irrelevant events to locate the relevant event to the feature (Wilde & Scully, 1995).

Textual analysis can be combined with the DFLTs to perform Textual Dynamic Feature Location Techniques (TDFLTs) to locate the software feature in the source code. The idea of the combination is to use the invoked events on the execution trace as a foothold to analyze the source code to locate a near-full set of features in a further process (Sulír & Porubán, 2017b). For example, a method profiled on the execution trace, an appropriate method for the feature, can be used as an input query to find the proper method on the source code. The output could be a list of source code elements related to the exhibited feature.

The dynamic part gathers the invoked events when a scenario or test case exercises the desired feature. Then, a filtering technique is implemented to select the relevant events from the execution trace for further analysis. The selected data from the execution trace can be used as input into the source code to find their relevant data based on the events' dependencies. For example, a static dependency graph can allocate the dependencies for each input event in the source code. Static analysis can be combined with DFLTs to present Static Dynamic Feature Location Techniques (SDFLTs) for showing the dependencies on the feature source code based on the execution trace input.

Textual, static, and dynamic analysis are combined to formulate a technique called Textual Static Dynamic Feature Location Techniques (TSDFLTs). The invoked methods or classes profiled in the execution traces can be used as input in either static or textual analysis to locate the feature in the source code. A static dependency graph can be combined with textual analysis techniques to improve the accuracy of FL.

2.3 Review of Publications

Herein, an overview of the related work to comprehend source code is represented in Subsection 2.3.1. Subsection 2.3.2 introduces a full dynamic approach of FL. Subsection 2.3.3 outlines several research studies about the hybrid approach of FLT.

2.3.1 Footstep to Comprehend Source Code

One direction in the software engineering research venues is to assist software practitioners, and developers in comprehending source code before processing software maintenance. Sulir (Sulír, n.d.) introduced a short literature review to summarize the research field of program comprehension theories. A brief introduction of DFLT, tools, and results was presented to understand the concept behind mapping the source code to the relevant feature.

Using FLT is the main key to comprehending the source code and identifying an initial point in the source code used to construct a function feature in a software system. Dit *et al.* (Dit, Revelle, Gethers, & Poshyvanyk, 2013) excelled in details about FLT. He studied eighty-nine articles about FL and classified FLT into SFLT, Textual Feature Location Techniques (TFLT), DFLT, and Hybrid Feature Location Techniques (HFLT). At the same time, Rubin *et al.* (Rubin & Chechik, 2013) explained in more detail the classification of FL and classified the output of FLT into guided and unguided output. In contrast, another study proposed a guideline for practitioners in the industry to select FLT based on their needs. Foothold or near-full of FL was the key to the selection to opt FLT by the practitioners.

A prominent work in the dynamic analysis was presented in a systemic survey to provide an overview of a large research body in DFLT. Cornelissen *et al.* (Cornelissen et al., 2009) reported a survey paper in dynamic

analysis to show the common DFLT's trending in research between 1999 and 2008.

2.3.2 Full Dynamic Approach of Feature Location

Analyzing the execution trace is the main principle of conducting a DFLT's. Muller *et al.* (Müller & Eisenecker, 2019) used Formal Concept Analysis (FCA) to analyze the generated execution trace to map the source code to the relevant feature. The proposed solution identified the relationship between the source code and the software feature based on extracting the software properties from the execution trace. When the relation was allocated, the FCA used a set of theories to identify whether the feature was a single, combination, or negation feature in the targeted analyzed system. Another use for FCA was proposed by Nakano *et al.* (Nakano, Noda, Hayashi, & Kobayashi, 2017). He used the FCA to filter the execution trace to increase the accuracy of performing DFL. The odds ratio and similarity coefficient concepts represent the relationship between the desired feature and the relevant source code.

Generated execution traces mainly rely on inputting scenarios or test cases to exhibit a desired software feature. Eisenberg *et al.* (Eisenberg & De Volder, 2005) proposed using test cases created under Test Driven Development (TDD) to exercise the desired feature. The generated execution trace contained all the methods invoked when executing test cases. Ranking heuristics techniques were implemented to identify code artifacts relevant to the exercised feature. Another solution was proposed by CAI *et al.* (Cai, Yang, Wang, & Kavs, 2012) to use a single scenario or test case to generate an execution trace. He used a control flow analysis to extend the execution trace and allocate all invoked classes. However, Ziftci *et al.* (Ziftci & Krüger, 2012) proposed using association rule learning on a set of labeled test-cases, i.e., test-cases labeled with the features they exercise in the

existing test-suite of the system, to locate features in source code.

2.3.3 Hybrid Approach in Dynamic Feature Location

Combining the DFLT's approach with SFLT's presents the correlation between using two techniques to enhance the accuracy of carrying FLT's. Rohatgi *et al.* (Rohatgi, Hamou-Lhadj, & Rilling, 2009) proposed HDFLT's solution by combining the SFL and DFLT's approaches. DFLT's approach generated execution traces by exercising the desired feature. The component dependency graph was the main static approach to rank the software artifact based on the similarity between components on the execution trace and the source code. The ranking system measured the impact of the software's granularity that appeared on the execution trace against the source code in the rest of the system. A dependency graph was created to illustrate the dependencies between classes profiled on the execution trace when the feature was exhibited by scenario or test case. Each node in the constructed graph represented a class.

Textual analysis can be combined with dynamic analysis to cover the limitation of each study in performing FL. Sulir *et al.* (Sulir & Porubän, 2017b) created a tool called RuntimeSearch: Ctrl+F for a Running Program based on dynamic and textual analysis. The tool was used as an instrumentation solution at Java Virtual Machine (JVM) level to find the User Interface (UI) text string occurrence at runtime. The located text string in the execution trace was used as a foothold to find a similar event of the located string in the source code to develop a near-full feature in a further stage. The tool principle was based on treating the running program as a series of expression evaluations. Every captured string on the system was compared against the invoked string at the runtime. The result of the dynamic analysis mapped to the textual research in the source code to locate all possible software artifacts related to the feature.

Static, textual, and dynamic analysis techniques can be combined to perform FL. Eaddy *et al.* (Eaddy, Aho, Antoniol, & Guéhéneuc, 2008) created a prune dependency analysis technique that combined information retrieval, execution trace, and prunes dependency analysis. The textual analysis approach used the IR technique to parse source code and extracts comment and identifiers, including the names of methods, to create a semantic analysis. Names were split into separate names based on standard naming conventions. In the dynamic analysis, the desired feature was exercised by a scenario to generate an execution trace. Each method in the execution trace was scored by using three dynamic feature location metrics, Software Reconnaissance (SR) (Wilde & Scully, 1995), Dynamic Feature Trace (DFT) (Antoniol & Guéhéneuc, 2006), and Scenario-Based Probabilistic Ranking (SPR) (Eisenberg & De Volder, 2005). The output was a list of ranked methods related to the exercised feature. The solution used a directed program dependency graph to illustrate the dependencies of the elements based on the initial seed point, which was a result of applying a threshold to identify highly relevant elements based on the result of the textual and dynamic analysis.

2.4 Systematic Literature Review

The literature search protocol was adjusted to locate the recent publications in software maintenance and how the FL approach is used as the primary solution to comprehend the existing software system. Gracia's guideline for performing literature review (García Holgado, Marcos Pablos, García Peñalvo, et al., 2020) was applied to locate a list of publications, and only part of them are selected in this review paper based on several essential principles that are illustrated in the coming sections.

2.4.1 General Setting for Search Techniques

The target was defined to analyse the current and recent publications that are relevant to FL. Scopus³ is the digital library that is used to search for the following search strings:

- Feature location in source code
- Dynamic feature location in source code

Table 2.1 outlines the search parameters used for analyzing current and recent publications relevant to Feature Location (FL). The search was conducted using the digital library Scopus. The search strings used were “Feature location in source code” and “Dynamic feature location in source code”. The search was limited to publications in the subject of Computer Science and in the English language. The variability in the search criteria was based on the search query terms and the year of the publications. All other settings remained on Scopus’s default setting. This systematic approach ensures a comprehensive and relevant collection of literature for the analysis.

Parameter	Description
Digital Library	Scopus ⁴
Search Strings	”Feature location in source code”, ”Dynamic feature location in source code”
Subject	Computer Science
Language	English
Variability	Based on search queries terms and year of publications
Other Settings	Scopus’s default setting

Table 2.1: Table of Search Parameters for Feature Location Publications

³<https://www.scopus.com/>

2.4.2 General Setting for Reading Process

Figure 2.2 reveals the number of scientific publications among two topics, FLT_s, and DFLT_s, in the period 2012-2020. It gives an impression of how DFLT_s are widely investigated compared with the general topic in FLT_s. Gray bars indicate the number of publications in DFLT_s, which is a subset of FLT_s. Black bars show the number of publications in FLT_s, which includes all types of FLT_s. For instance, the number of publications in 2020 for FLT_s is 477, while the number of publications in the same year for DFLT_s is 174. FLT_s bars indicate all types of FLT_s, including DFLT_s, TSDFLT_s, TDFLT_s, and another unclassified venue under the FLT_s domain. DFLT_s bars show only the number of publications in a certain field in FL.

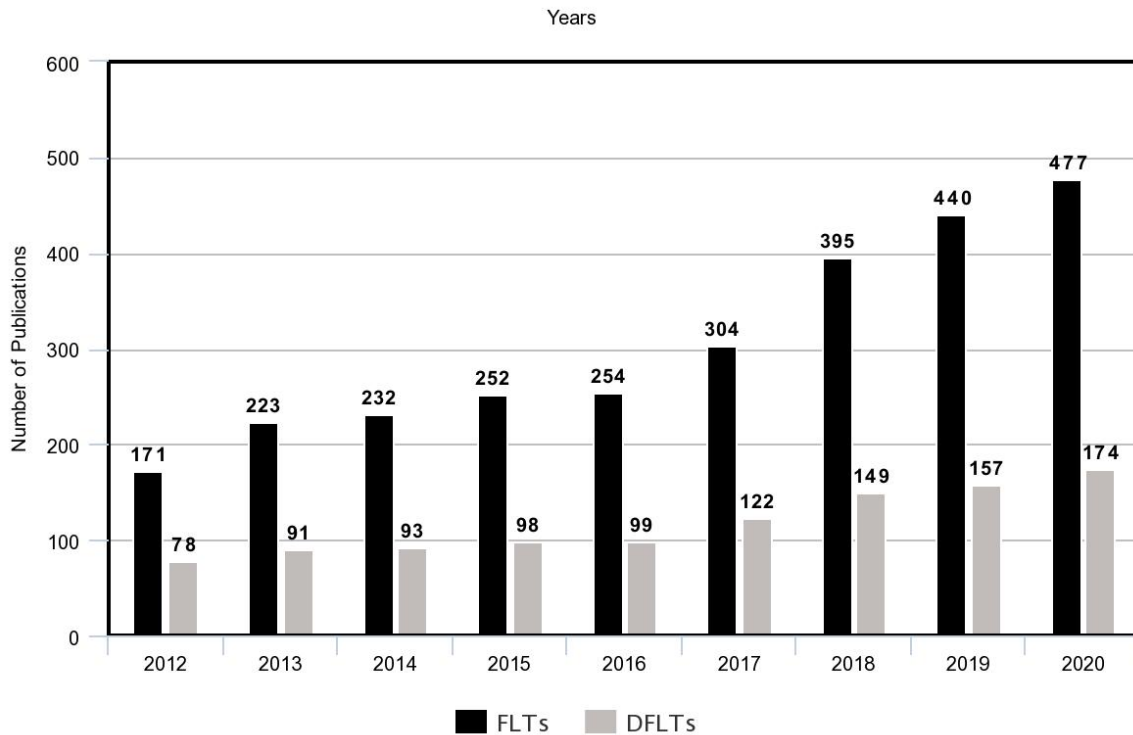


Figure 2.2: The number of scientific publications referring to the search of FLT_s and DFLT_s in the period 2012-2020. Source: Scopus (accessed on 31 March 2021).

Comparison of DFLT_s with FLT_s, in general, announced that DFLT_s have been less investigated in software maintenance than the general FLT_s

venue due to the challenges of understating the system at runtime. Figure 2.2 illustrates the changes of the number of publications in FLT and DFLT between 2012 and 2020. The result for all years shows the number of publications in DFLT is 1061 which is less than the number of publications in FLT which is 2448.

2.5 Analysing the Finding of the Systematic Review

2.5.1 Common FL Techniques Used in Publications

DFLTs utilize the information from a running system to map a software feature to the relevant source code. Figure 2.3 classifies DFLT into five sections based on a pilot study in DFLT between 2012 and 2020. The suggested classification aims to categorize the work into five sections, FDFLTs, TDFLTs, SDFLTs, TSDFLT, and others. The initial search query on Scopus is embedded by the type of the analysis such as Static, Dynamic, Textual, or a combination between more than one type to obtain the statistics result. TSDFLT merges three techniques, textual, static, and dynamic analysis. It has the lowest number of research work in DFLT with 68 publications between 2012 and 2020. TDFLT combines the dynamic analysis with textual analysis to perform FL has 122 publications. SDFLT propose to combine static analysis and dynamic analysis to map features to the source code. It has 402 publications between 2012 and 2020. FDFLT has 254 publications that use only execution trace to implement FL. Lastly, others show the number of papers that have fallen into the DFLT domain, yet have not been classified into either FDFLTs, SDFLTs, TSDFLT, or TDFLTs.

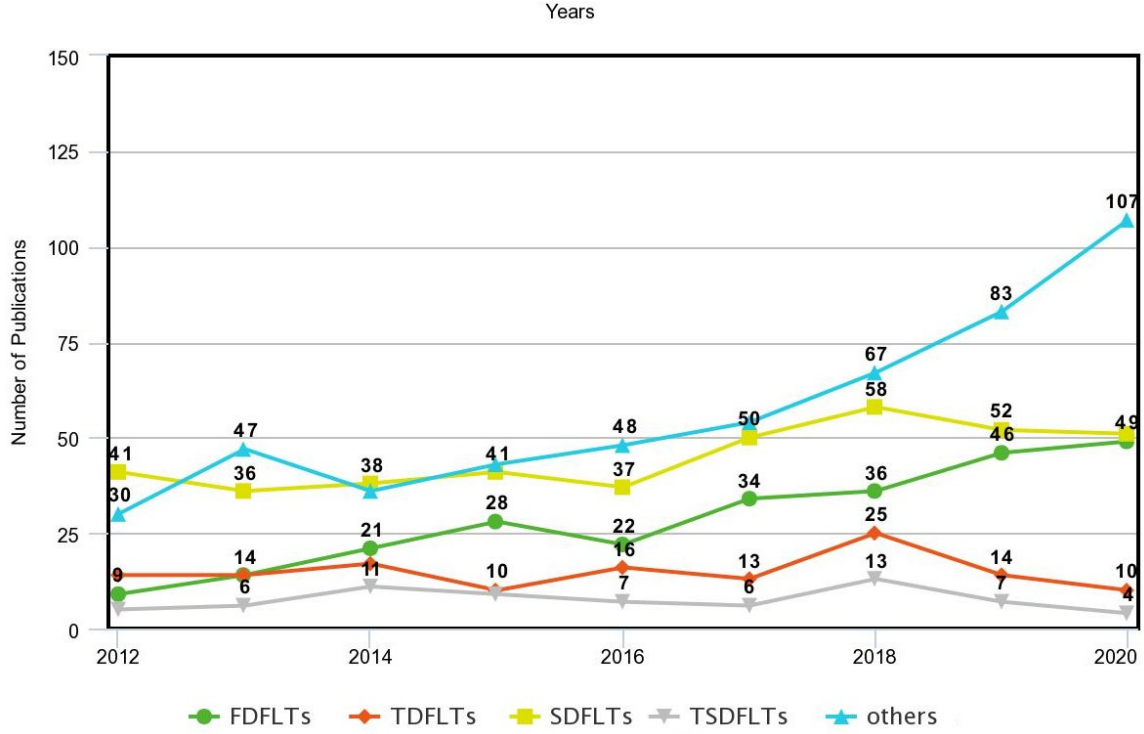


Figure 2.3: Number of techniques used.

2.5.2 Classification of DFLTs

DFLTs consist of a large variety of models in continuous evaluation, providing uncertain borderlines and cross relationships. In addition, other aspects and applications could guide different classifications. Thus, it is not possible to select or create one universal taxonomy for all problems and solutions from DFLT literature. Yet the depicted taxonomy can be presented based on an original view to identifying the diversity among the multiple techniques that are implemented in DFLTs, as drawn in Figure 2.4. It is adapted to DFLTs and prevent representing any FLT that do not include a dynamic solution. The first unique evidence in Figure 2.4 is to classify the DFLTs into four branches, each one has a dynamic approach. FDFLTs, TDFLTs, SDFLTs, and TSDFLTs can be further characterized into several sub-branches, and execution trace is the common sharing approach between

all branches in the taxonomy. The corpus-based analysis is applied in two techniques, TSDFLT and TDFLT. Dependencies analysis is covered in two FL solutions, TSDFLT and SDFLT.

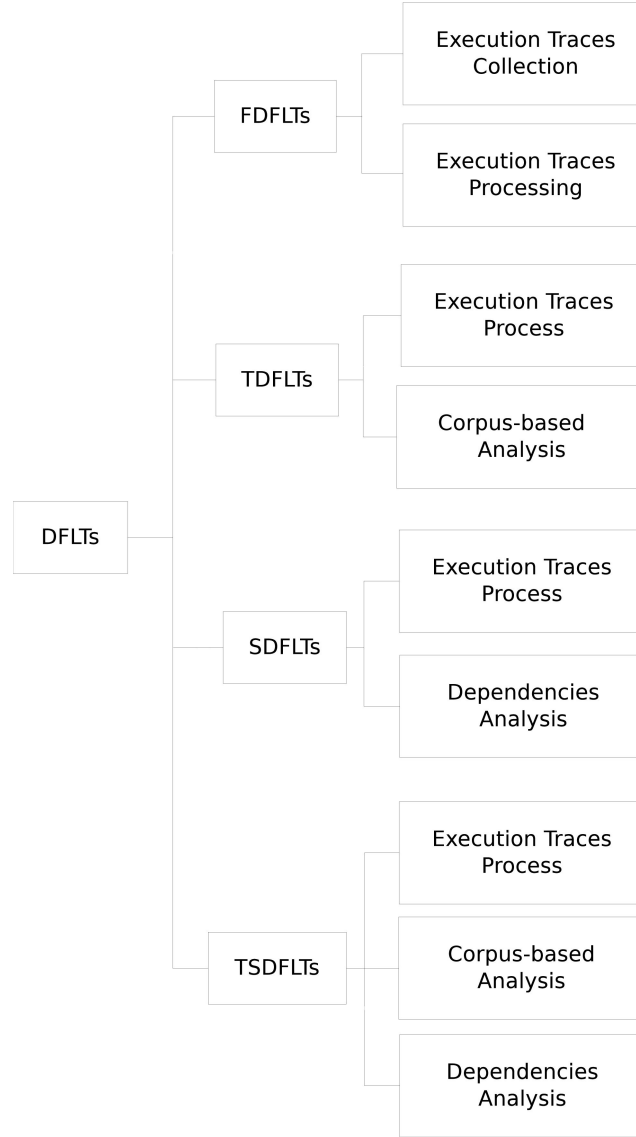


Figure 2.4: The main classification of DFLT Taxonomy.

2.5.3 A Common Shared Applications in DFLT

Table 2.2 reports the main applications which are proposed to spotlight DFLT that were located in earlier research. Rows indicate the reference

of publications that described DFLT. Columns represent the certain application that is used as a part of the DFLT solution. Each cell indicates whether the technique is implemented or not. When the cell is empty, it points that, to the best of our knowledge, the technique is not used in the presented references. This table shows that three techniques are applied to all considered references in DFLT. The corpus-based solution is categorized into three techniques, LSI, LDA, and VSM. Execution trace is classified into two sections. Profiler when the solution uses an external tool to profile all invoked events, and bytecode instrumentation if the DFLT instrument the source code to profile invoked events. Dependencies are classified into control flow or call graph diagrams to show the relation between the source code or the execution traces.

The Table 2.2 you provided is a classification view of the most common techniques used in DFLT. It lists various research references along the left column and different techniques along the top row. The techniques are divided into three categories: Corpus-based, Execution Traces, and Dependencies.

Each row in the table corresponds to a specific reference or set of references. The checkmarks in the cells indicate which techniques are used in the corresponding reference. For example, the first row indicates that the references (Cruz, Figueiredo, & Martinez, 2019; Razzaq et al., 2020) use all the listed techniques.

The Corpus-based category includes techniques such as LSI, LDA, and VSM. The Execution Traces category includes techniques such as Profiler and Bytecode Instrumentation. The Dependencies category includes techniques such as Control Flow and Call Graph.

This table provides a comprehensive overview of the techniques used in various studies on DFLT, allowing for easy comparison and understanding of the common methods in this field.

Table 2.2: A CLASSIFICATION VIEW AMONG THE MOST COMMON TECHNIQUES THAT ARE USED
IN DFLT's.

References	Corpus-based				Execution Traces		Dependencies	
	LS	LD	VSM	Profile	Bytecode	Inst.	Control Flow	Call Graph
(Cruz et al., 2019; Razzaq et al., 2020)	✓	✓	✓	✓	✓	✓	✓	✓
(T.-D. B. Le, Oentaryo, & Lo, 2015; Takahashi, Sae-Lim, Hayashi, & Saeki, 2021; Belmonte & Dugerdil, 2018)	✓	✓	✓		✓	✓		
(Michelon, Linsbauer, et al., 2021)	✓	✓	✓				✓	✓
(Romano, Scanniello, Antoniol, & Marchetto, 2018)	✓		✓	✓		✓	✓	✓
(Eyal-Salman, Seriai, & Hammad, 2016)	✓	✓	✓		✓	✓		
(Kuang et al., 2015)	✓	✓	✓				✓	✓
(DeLozier, Decker, Newman, & Maletic, 2018; ANDAM, 2018)	✓			✓				
(Fujioka & Nitta, 2017; Kazato, Hayashi, Oshima, et al., 2013)	✓					✓		✓
(Chochlov, English, & Buckley, 2017)			✓		✓	✓		✓
(Santos & Albuquerque, 2016; Henderson & Podgurski, 2018)					✓	✓		✓
(?, ?, Braun, Amyot, & Lethbridge, 2015; Sisman, Akbar, & Kak, 2017)				✓		✓	✓	✓
(Nakano et al., 2017; Hayashi et al., 2016)				✓	✓			✓
(Dhamija & Sikka, 2019)				✓	✓	✓	✓	✓

The references listed in the Table 2.2 are those that have employed the techniques categorized under Corpus-based, Execution Traces, and Dependencies in the field of DFLT. They are listed because they provide a diverse range of methodologies and approaches within this field. The selection of these specific references helps to illustrate the breadth and depth of techniques used in DFLT, and serves as a representative sample for researchers and practitioners interested in this area. However, it's important to note that while these references are indicative of the work done in this field, they do not encompass all the research in DFLT. There may be other relevant studies not included in this table.

2.5.4 Challenges

Being able to create a software system and having a confidence level in coding is a prerequisite for program comprehension activities, particularly for DFLT approaches that connect runtime information with the source code. A comprehension study in DFLT shows several difficulties that can decrease the usability of mapping source code to the relevant software feature via analyzing the system's actual behavior. Project size and age are the main vectors to create the most frequent error related to software dependencies. For instance, relying on the existing tools in DFLT to solve FL problems in Java projects from GitHub ended with a failure in 38 % of cases (Sulír, n.d.).

DFLTs, like other feature location methods, encounter a variety of issues and challenges. Many of these challenges are associated with different IR techniques such as LSI, LDA, or VSM (Oliveto, Gethers, Poshyvanyk, & De Lucia, 2010). The most common and significant challenges lie in the precision and recall of the data retrieved. The accuracy of precision and recall in DFLT depends on the execution trace generated, as well as the quality of the query. For example, when a feature is activated, the resulting

execution trace may contain numerous irrelevant events, increasing the likelihood of lower precision and recall accuracy. As a result, it's essential to filter execution traces before applying any IR technique to construct DFLT's.

DFLT's issues and challenges are reviewed from several perspectives. They can be categorized into several construction-related problems and internal and external issues. The following subsections explain the DFLT's issues and challenges for each.

2.5.5 Benchmark Creation

One of the challenging in the area of DFLT's is the evaluations. The number of benchmark datasets available for DFLT's is notably less than those available for SFLT's or TFLT's (Rohatgi, Hamou-Lhadj, & Rilling, 2008; Buzaid & Hammad, 2021b). This deficiency leads to a gap in our understanding and capability to effectively design a comprehensive dynamic framework for DFLT's. Moreover, even the existing DFLT's benchmark has a lack in a real implementation to find a complete set. For instance, the different results of the mapping between feature and source code can be generated with the same case study by different developers. Therefore, having the right benchmark to evaluate the DFLT's is one of the most important challenges that should be considered in DFLT's.

2.5.6 Data Collection

Most of the DFLT's include IR techniques that are used by developers to retrieve the relevant data from the execution trace. However, assigning a vector to reflect the size of the selected source code compared to the near-full feature source code is not applicable in many DFLT's. For example, using any IR technique to rank and order the retrieved data does not guarantee the real importance of the sorted data (Nakano et al., 2017). The retrieved or-

dered data can be indicating whether the retrieved data is a foothold feature or not, which is far from indicating a full set of feature.

2.5.6.1 Obtaining the Execution Trace Configurations

Automating the process of saving and updating an execution trace file streamlines is hard. The main challenges remain in capturing and storage of trace data, ensuring continuous and accurate profiling of required invoked data when the feature is executed by the scenario. The proposed approach seeks to obtain several execution traces for every implemented scenario as each scenario would be partitioned into several scenarios. Each partitioned scenario generates its relevant execution trace. Hence, an aggregation solution is needed to collect the entire parts of the execution traces for the relevant scenario for a further process in AETFS.

Several steps are required to be applied to automate the process of aggregating the entire execution traces for a running scenario. The first step in automating the process of saving and updating an execution trace file is to define the specific points in the code where trace data should be collected. These points can include method entry and exit points, specific events, or checkpoints that are relevant to the monitoring and analysis of the application's execution flow. By carefully selecting these collection points, developers can ensure that the trace data captures key information about the feature's behavior and profiling the invoked data. This step lays the foundation for effectively monitoring and analyzing the execution trace, enabling a better understanding of the scenario's behavior.

Once the collection points for trace data have been identified, the next step in automating the process is to implement the capture of the trace data. This involves writing code to extract the relevant information at the specified collection points. Depending on the requirements, this may include capturing method names, parameters, timestamps, variable values, or any

other data that provides insights into the program's execution. By accurately capturing the trace data at these designated points, developers can obtain a comprehensive view of the executed scenario's behavior and performance, enabling effective analysis of the scenario behavior against the feature. Implementing the trace data capture ensures that the necessary information is collected consistently and reliably, forming the basis for further automation and analysis in the trace file.

Moreover, depending on the number of partitioned scenarios, the solution can be selected to update the trace file. This step plays a crucial role in determining how frequently the trace data should be saved, ensuring an up-to-date and accurate representation of the application's execution flow.

Then, it is required to set up proper file handling for the trace file. This step is crucial to ensure reliable and secure operations on the file. It involves creating or opening the trace file in the appropriate mode, such as append or overwrite, based on the specific requirements. Additionally, it is essential to implement error-handling mechanisms to gracefully handle any exceptions or issues that may occur during file operations. Proper file handling ensures that the trace data is accurately and consistently saved without data loss or corruption. It also includes considerations for resource management and potential file locking mechanisms to handle concurrent access scenarios. By addressing file-handling requirements, developers can establish a robust foundation for storing and updating the trace data effectively.

However, modern Version Control Systems (VCSs) can be used to support aggregation of the execution trace to some extent. Aggregation support broadly refers to any functionality in VCSs that allows for identification and grouping of the execution traces that are generated by exercising the software feature by a scenario. Table ?? provides a brief comparison of Git, SVN, and CVS in terms of their version control capabilities for saving and updating execution traces, considering factors such as distribution,

branching, commit granularity, performance, conflict resolution, scalability, repository size, integration, and popularity/support.

Table 2.3: Comparison of Version Control Systems for Execution Trace Management (Bousse et al., 2019)

Git	SVN	CVS
Distributed	Centralized	Centralized
Efficient and easy	Supported but complex	Supported but complex
Granular	File-level	File-level
Fast	Slower compared to Git	Slower compared to Git
Efficient	Supported but challenging	Supported but challenging
Compact	Centralized	Centralized
Strong integration	Good integration	Limited integration
Compact	Centralized	Centralized

Using VCS to save and manage execution traces generated from exercising features through a set of scenarios proves highly effective in AETFS. VCS allows for systematic organization and storage of the execution traces, ensuring traceability and easy retrieval. By capturing the changes made during scenario execution, VCS enables developers to track the evolution of the traces partitioned scenarios execution, facilitating analysis and mining the execution traces. The ability to create branches in VCS enables parallel development of different scenarios, allowing developers to work simultaneously without conflicts. Furthermore, VCS supports merging, enabling the consolidation of changes from multiple scenarios into a single aggregated trace. This enhances collaboration and ensures that the execution traces accurately represent the full range of scenarios. With the ability to roll back to previous versions, VCS provides a safety net to restore or compare earlier versions of the traces. Overall, the use of VCS in saving and managing execution traces resulting from scenario-based feature testing enhances trace-

ability, and collaboration, and facilitates efficient scenario management in AETFS.

2.5.6.2 Aggregating Execution Traces by Granularity and Based on the Partitioned Scenarios

The existing research tried to address the problem of identifying the required parts of the scenario that apply the required action to perform DFLT. Several solutions have been proposed to address the dependency on test cases or usage scenarios in applying DFLT (Cornelissen et al., 2008; Hojaji, Mayerhofer, Zamani, Hamou-Lhadj, & Bousse, 2019; Zangeneh, 2021; Al-Sharif, Abdalrahman, & Jeffery, 2021; Almulla & Gay, 2022). Here are a few existing research solutions:

- The authors (Cornelissen et al., 2008) used a tool called Extravis⁵, which implements the proposed visualization techniques of the circular bundle view and the massive sequence view for execution trace analysis.
- Another example is the Papyrus-RT⁶ framework, which is an extension of the Eclipse Papyrus tool specifically designed for real-time and embedded systems. Papyrus-RT offers a simulation and execution environment where models can be executed, and execution traces can be captured for analysis of the formulated test case (Hojaji et al., 2019).
- Another solution is Test2Vec approach (Zangeneh, 2021). It utilizes execution traces, including input values, to capture behavioral patterns of test cases and outperforms traditional code coverage metrics in distinguishing failed test cases.

⁵<https://www.primordion.com/Xholon/wiki/ExTraVis/>

⁶<https://github.com/kjahed/umlrtonline-papyrusrt>

- Another solution utilizes Text2Vec tool⁷ to monitor the framework as a lightweight architecture for capturing execution traces during software testing (Al-Sharif et al., 2021). Text2Vec is to propose a test case encoding approach using execution traces to minimize redundant test cases and enhance the efficiency of software testing processes.

In AETFS, given or formulating a random scenario by developers who are unfamiliar with the system, the Java Agent Bytecode Instrumentation, allows AETFS to generate execution traces without modifying the source code. It allows the developers to comprehend the software code and provide details of the system during runtime by logging method invocations, parameter values, and other relevant information. In this subsection, the generated execution traces were illustrated based on using mouse clicking to decompose the formulated scenario into several steps.

Generating execution traces based on Java agent bytecode instrumentation by mouse-clicking would require a combination of techniques and tools. Here's a high-level outline of the process:

- Identify the target application: Determine the Java application or GUI framework that you want to generate execution traces for by mouse clicking.
- Set up a Java agent: Develop a Java agent using bytecode instrumentation techniques, as discussed earlier. The agent should be capable of intercepting mouse click events and capturing relevant information.
- Instrumentation for mouse click events: Within the Java agent, identify the classes or methods responsible for handling mouse click events in the target application. Use bytecode manipulation techniques to modify these classes or methods to capture execution traces whenever a mouse click event occurs.

⁷<https://github.com/dselivanov/text2vec>

- Capture and store execution traces: Define a mechanism within the Java agent to capture the execution traces when a mouse click event is intercepted. This could involve logging the traces to a file, storing them in a database, or sending them to a centralized repository for further analysis.
- Attach the Java agent to the JVM: Start the target application with the Java agent attached to the JVM using the `-javaagent` command-line option, specifying the path to the agent JAR file.
- Interact with the application: As you interact with the target application by clicking the mouse, the Java agent will intercept the mouse click events, capture the associated execution traces, and store them according to your defined mechanism.
- Analyze the execution traces: After interacting with the application and generating a set of execution traces, you can analyze the captured traces to gain insights into the behavior, user interactions, and any issues or patterns that may arise.

It is important to note that implementing Java agent bytecode instrumentation for capturing mouse click events requires knowledge of the target application's GUI framework, event handling mechanisms, and the appropriate bytecode manipulation techniques for intercepting and capturing the events (Soueidi, Falcone, & Hallé, 2023). Additionally, the instrumentation should be designed to handle different types of mouse click events and provide the necessary context for effective analysis.

Figure 2.5 shows how can be defined a Java agent named `MouseClicked-Agent` that uses bytecode instrumentation to intercept mouse click events in Swing's `JComponent` class. Note, in a Java application, the `premain` method acts as the entry point for Java agents, allowing dynamic bytecode modifica-

tion by registering a `ClassFileTransformer` to modify loaded classes during runtime (Binder, Hulaas, & Moret, 2007).



```
5 public class MouseClickAgent {
6     @
7     public static void premain(String agentArgs, Instrumentation inst) {
8         inst.addTransformer(new ClassFileTransformer() {
9             @Override
10             public byte[] transform(
11                 ClassLoader loader,
12                 String className,
13                 Class<?> classBeingRedefined,
14                 ProtectionDomain protectionDomain,
15                 byte[] classfileBuffer
16             ) throws IllegalClassFormatException {
17                 if (className.equals("javax.swing.JComponent")) {
18                     return transformJComponent(classfileBuffer);
19                 }
20                 return classfileBuffer;
21             }
22         });
23     }
```

Figure 2.5: Java code snippet that demonstrates how you can use Java agent bytecode instrumentation to capture mouse click events and generate execution traces.

The `transform` method is called for every class loaded by the JVM. In this example, we specifically target the `JComponent` class and use the ASM library to transform its bytecode. We override the `visitMethod` method to intercept the `processMouseEvent` method and replace the invocation of `processMouseClicked` with our custom code for capturing the execution trace.

The `captureMouseClicked` method is the custom code that gets inserted into the transformed bytecode. Here, we simply print the coordinates of the mouse click event, but you can modify this method to capture the necessary information and store it as per your requirements.

Remember to compile this code into a separate JAR file and specify it as a Java agent using the `-javaagent` command-line option when running your

target Java application.

Please note that this code snippet is a simplified example to demonstrate the concept of Java agent bytecode instrumentation for capturing mouse click events. In practice, more considerations such as handling different event types, handling multiple classes, and error handling should be taken into account.

Java Agent Instrumentation is utilized in AETFS to facilitate the generation of execution traces corresponding to a set of scenarios (Arcega, Font, Haugen, & Cetina, 2017; Wu et al., 2022). These gathered execution traces can be stored in an array collection, where each array holds an execution file that represents a portion of the partitioned scenario (Arcega et al., 2017; Wu et al., 2022), or in a multipartite graph (Hendriks et al., 2017; Müller & Eisenecker, 2019). For instance, if the method outlined by Muller et al. is applied to the method in the running example, it would yield traces that correspond to the scenarios, as depicted in Figure 2.6 (Müller & Eisenecker, 2019). In this figure, the circles represent the partitioned scenarios. A circle containing the letter “T” indicates that the partitioned scenario is included to demonstrate the software feature, while “E” signifies that the scenario is excluded. An empty circle suggests the potential application of an additional random scenario. The horizontal lines between the circles provide a visual guide to trace the position of the partitioned scenarios. If a line does not have edges leading to the next partitioned scenarios, it implies that the partitioned scenario was removed in the subsequent feature demonstration.

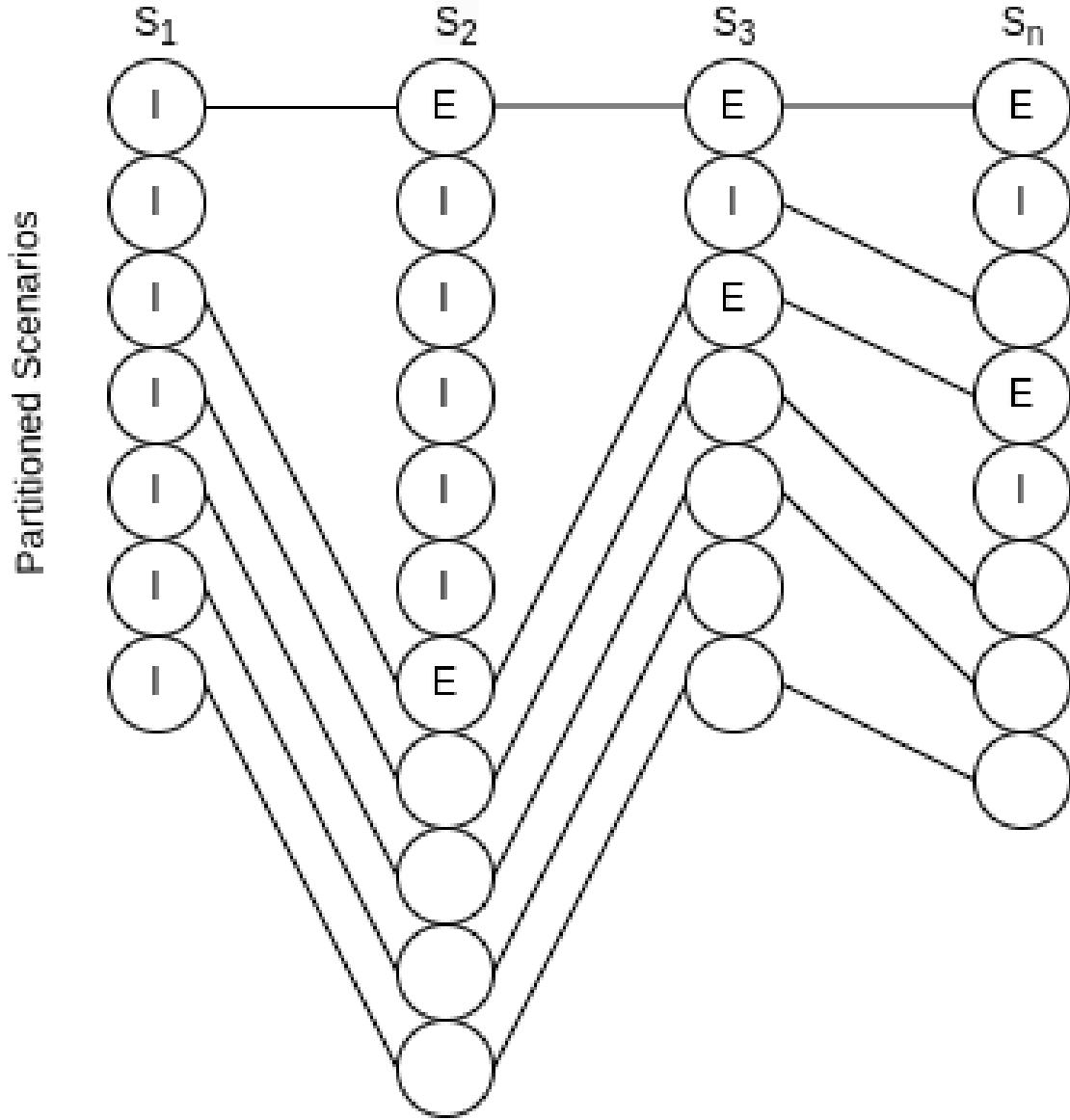


Figure 2.6: Aggregation of Execution Traces for partitioned scenarios for the running example.

(Ali et al., 2019) method can be adapted to create a universal aggregation model that accommodates all AETFS configurations, as depicted in Figure 2.7. In this model, each scenario, taken from a specific version, is represented by a list of N sub-scenarios. Each sub-scenario (each list member) points to a linked list of execution traces that capture the evolution (capability to invoke required data) of this sub-scenario. Consequently, each node in

the linked list refers to a sub-scenario with the relevant execution traces that exhibited the desired feature. Each sub-scenario with a relevant execution trace reference points to a hash table of M (the sum of the arbitrarily sized linked lists, over the S number of sub-scenarios) for quick entry lookup. This model enables efficient storage of only the necessary sub-scenarios to exercise the feature (only sub-scenarios capable of generating a relevant execution trace to the exhibited feature are considered) and allows for versatile configurations of several sub-scenarios (including scenarios formulated by the Gherkin standard).

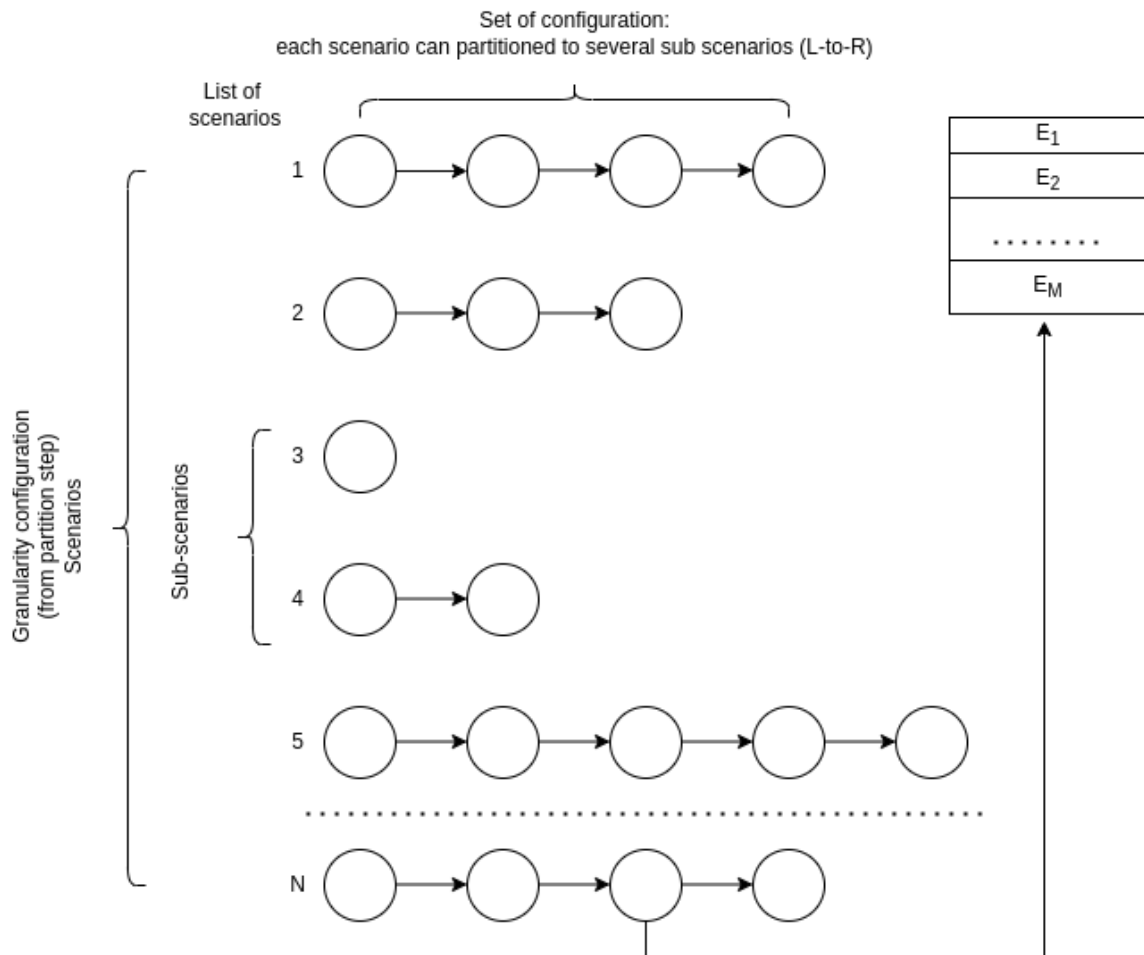


Figure 2.7: Generic sub-scenarios aggregation approach.

However, the following examples demonstrate how this model can be

employed to aggregate the execution traces with their relevant scenarios for the various configurations, as outlined in Table ??:

1. Setup and Integration:

- Initialize a Git repository for your software project.
- Ensure that your project is properly configured to generate and capture execution traces during runtime.
- Configure your VCS (e.g., Git) to track and commit the execution trace files along with the source code.

2. Capturing Execution Traces:

- Run your software or system while the execution trace-capturing mechanism is active.
- The captured execution traces could include logs, debug information, or any other relevant data.
- Ensure that the execution trace files are generated and saved within the project directory.

3. Committing Execution Traces:

- Use Git commands (e.g., `git add`, `git commit`) to stage and commit the execution trace files to the repository.
- Include descriptive commit messages that provide context about the captured execution traces.
- Commit the trace files alongside the source code changes that are associated with the specific execution trace.

4. Analyzing Execution Traces:

- Utilize Git’s history and diff capabilities to inspect and analyze the execution trace files.
- Use commands like `git log`, `git diff`, or graphical tools to review the changes in the execution traces across commits.
- Identify patterns, anomalies, or trends in the execution trace data through comparison and visualization.
- Leverage Git branches and tags to isolate specific execution trace versions for focused analysis or comparison.

5. Collaboration and Knowledge Sharing:

- Share the execution trace commits with the development team or relevant stakeholders.
- Use Git’s collaboration features (e.g., pull requests, and code reviews) to discuss and provide feedback on the execution traces.
- Integrate with issue tracking systems (e.g., JIRA, GitHub Issues) to link execution traces to specific bugs or feature requests.
- Collaborate on analyzing the execution traces and use Git’s commenting features to document observations or insights.

2.5.7 Indexing the Content of the Collected Execution Traces

In the final phase, the gathered execution traces must be refined to form the document corpus D , which requires storage and retrieval capabilities. The AETFS employs VSM for indexing and querying these text-based documents. The choice of VSM is attributed to its well-researched nature and proven efficacy in applying FL (Razzaq et al., 2018).

Other IR models, such as LSI and LDA, have been utilized in DFLT’s and related areas (Biggers et al., 2014; Lee & Ostwald, 2024). However, mul-

multiple researchers have noted their inferior performance compared to VSM. This lesser effectiveness is attributed to the limited range of concepts derived from execution traces. Consequently, the efficacy of LSI and LDA in this domain remains unverified, with VSM continuing to be the favored method (Silva, Galster, & Gilson, 2021). Nonetheless, this study employs VSM, LDA, and LSI in conjunction with a TF-IDF function for document analysis and cosine similarity for document comparison.

2.6 Future Directions

The DFLT's are not investigated as much as SFLT's in program comprehension. Many software researchers and practitioners face difficulties to select the most proper DFLT's to enhance their process to comprehend the software system. A lack of standardizing the process to compare DFLT's would exacerbate the evaluation steps. There is a need to standardise DFLT's comparability to address the advantage and disadvantages of the possible selected DFLT's to support the developers to understand the existing software systems.

Many researchers have provided a contribution in DFLT's, to increase the level of program comprehension for the existing software programs. Understanding the boundaries of the proposed software feature is a key element for the developer to opt for the right DFLT's. Analysing existing DFLT's empirical studies show that the system program language may affect the performance of the selected DFLT's. There is a necessity to understand the interaction between programming language and DFLT's, to perform FL. There is a vast demand in software engineering to show how DFLT's can interact with modern high-level program language such as Julia⁸. The domain of DFLT's calls for finding to regulate the process of using execution trace to

⁸<https://julialang.org/>

perform FL based on using any programming language.

To increase the level of accuracy of mapping source code to the relevant feature in using DFLT, we need to consider existing issues and challenges, the limitation, and possible solutions to the challenges. As the number of software programs is increased year by year, more sophisticated DFLT are needed to overcome the comprehension challenges for maintaining existing software. Consequently, more efficient and sophisticated DFLT should be designed to address a problem such as maintaining existing software that has no documentation or maintaining a system by unfamiliar developers with the system. Moreover, the solution provided by DFLT should not rely on program structure as the existing systems that are needed to comprehend can be poorly structured with no meaningful lexicon. The purpose of using DFLT should be to map features to the source code for any existing software system with the optimal time requested with the minimum required training is needed to use DFLT by developers.

2.7 Conclusion

Although the number of solutions in FL is increased nowadays, the challenges in dynamic feature location are getting more tangled. This literature reviewed the current prominent dynamic feature location solution. A large number of the current solutions contain steps where corpus-based analysis, execution trace collection, and dependencies analysis are used. A comparison of several DFLT is implemented to categorize the list of techniques into three prominent techniques. The results suggest that the selected techniques in corpus-based analysis, such as using LSI, and the selected technique in execution trace collection, such as using bytecode instrumentation, play essential roles to provide the most current solution in mapping features to the source code in DFLT.

As DFLTs gain prominence in modern software maintenance, identifying the most suitable DFLT for feature location in a given software system is crucial. The key metric for evaluating DFLT is their ability to accurately relate a software's functional feature to its relevant artifact. Numerous studies have introduced novel methods to enhance DFLT's efficacy and sophistication, as well as to address associated challenges. This chapter provides a literature review from various perspectives, offering a taxonomy and analysis of past research to present a comprehensive view of existing techniques. Emphasizing the common approach of mapping features to software artifacts using DFLT, potential weaknesses and gaps in DFLT are highlighted. Lastly, future directions are outlined to guide software practitioners and researchers in further investigations to address DFLT-related issues and challenges.

Chapter 3

Proposed Methodology

3.1 Overview

As described above, formulating an accurate scenario to apply DFLT is a noted problem. In the search for a solution to increase the accuracy of the formulated scenario, enhancing the process of using DFLT is proposed in AETFS. The solution leverages the collected execution traces to measure the accuracy of the applied scenario to perform DFLT.

There are several different configuration options available to AETFS:

- The AETFS technique is employed to assess the accuracy of a scenario, which is devised by software developers who are not familiar with the given software system. These developers, relying on their own understanding, intuitively formulate the scenario. A single execution trace is produced in this process. To gauge the accuracy of the executed scenario, these developers utilize a text query, leveraging IR

Techniques such as TF-IDF

- The AETFS technique applies to measure the accuracy of the scenarios that are formulated based on Behavior-Driven Development (BDD) (A. Z. Yang et al., 2019) by using Gherkin¹ syntax structures plain text. Several execution traces are generated, and the developers use advanced topic modelling information retrieval techniques to measure the accuracy of the executed scenarios.
- The AETFS technique applies to measure the accuracy of the scenarios that are formulated based on using Unified Modeling Language (UML), such as using Use Case Diagram (Buzaid, Albalooshi, & Elmedany, 2022).

This chapter begins by introducing the AETFS approach, supplemented by a practical example to aid in its explanation. Subsequently, it outlines the baseline approach, which serves as a comparison point for AETFS and delves into the implementation specifics of both AETFS and the baseline approach.

3.2 The AETFS Approach

A provisional description of the AETFS approach was first presented in (Buzaid & Hammad, 2021a). It was later extended and evaluated in (Buzaid, Albalooshi, & Elmedany, 2022; Buzaid, Albalooshi, & Hammad, 2023), and is presented in detail in Figure 3.1. The AETFS approach allows for aggregation of collected execution traces to indicate the ability of the executed scenario to locate the source code entity that has been touched when a software feature is exercised by the scenario to apply DFLT's. The process

¹<https://cucumber.io/docs/gherkin/>

includes combining the resultant, textual, scenarios, and the feature's query descriptions with IR, it allows for indexing, and searching in the collected execution traces, given a search feature query or a given scenario.

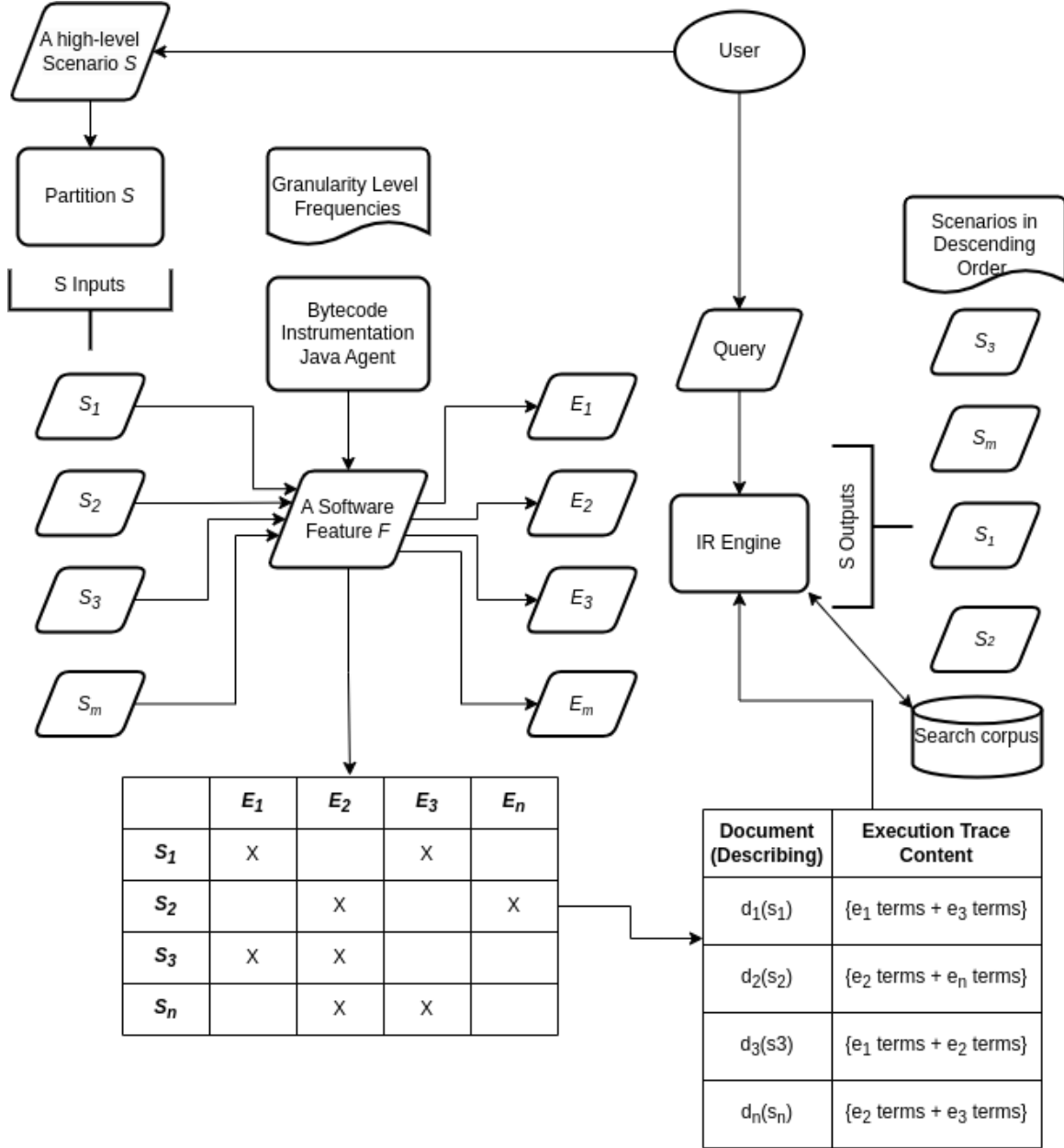


Figure 3.1: The AETFS approach.

The input to AETFS is a scenario / or a set of scenarios, and an instrumentation agent that generates an execution trace or a set of execution traces $E = \{e_1, \dots, e_M\}$ on the targeted executed feature F , where M is the number

of such collected execution traces. AETFS relies on the descriptions in E to generate the textual content of documents D for IR, where each document represents a source code entity. There are three major steps in the AETFS approach, as shown in Figure 3.1:

1. Partitioning scenarios into a set of scenarios segments $S = \{a, \dots, a_N\}$, where N is the number of segments
2. Annotating scenarios that segments based on the related execution traces. In this step, AETFS matches all the scenario segments in A with the execution traces in E (that are generated when the targeted F is executed). AETFS then aggregates textual data from the set of execution traces (that are mapped to the scenarios segments) to add content to the documents $D = \{d_1, \dots, d_N | \forall d_i, \exists E_i \subset E\}$, where D describes the source code entities the related to the F when it is exercised by S .

This step is divided into two sub-steps:

- (a) Aggregate the set of execution traces for every scenario segment (that exercised the software feature). AETFS allows for aggregation of the set of execution traces according to the granularity of the Instrumentation Agent, profiling the invoked methods, classes, and objects, or profiling the recurrent of the invocations with respect to the configuration of the instrumentation agent (see Section 2.2.3). Depending on the choices made, this results in the formation of the collected execution traces $E_i \subset E$ for every scenario segment in the set of scenarios $s \in S$.
- (b) Represent the accuracy of the scenario segments with the related execution traces (where each execution trace was formed based on the content of the relevant scenario segment, according to the

selected configuration in the previous step). For each content of the extracted execution trace, textual and process text analysis is applied resulting in a set of textual documents D , each element of which measures the accuracy of the executing scenario in applying DFLTs.

3. Use the IR engine to index the content of the execution trace to the relevant scenarios entities (their textual representation: the set of documents D) and to later use these outputs to apply DFLTs.

3.2.1 Preliminaries and Notations

In the AETFS approach, several key concepts and notations are fundamental for understanding its functionality and processes. These notations are crucial for articulating the methodology's intricacies and computational operations efficiently.

Notation	Description
$E = \{e_1, \dots, e_M\}$	Set of execution traces generated by the instrumentation agent
F	Targeted executed feature
M	Number of collected execution traces
D	Set of textual documents for Information Retrieval (IR)
$S = \{a, \dots, a_N\}$	Set of scenario segments
N	Number of segments in the scenario
d_i	Textual document representing a source code entity
$E_i \subset E$	Subset of execution traces for a scenario segment

3.2.2 Major Steps in the AETFS Approach

1. Partitioning Scenarios:

- Dividing scenarios into segments for further analysis.

2. Annotating Scenario Segments:

- Matching scenario segments with related execution traces to aggregate textual data for IR.

(a) Aggregation of Execution Traces:

- Profiling method invocations based on the configuration of the instrumentation agent.

(b) Accuracy Assessment:

- Analyzing textual content of extracted execution traces to measure scenario accuracy.

3. Information Retrieval (IR) Engine:

- Indexing execution trace content to scenario entities for DFLT's application.

In the subsequent Chapters, these notations and processes in the AETFS approach will be further elaborated and contextualized within the research framework.

3.3 AETFS Technique in Details

In this section, the steps of the AETFS technique are presented in more detail, describing the transformation of the two inputs to AETFS, a high-level scenario to exercise the software feature, and a text query that describes the software feature, into the textual documents that represent the input scenario with accuracy value in applying DFLT's. Hence, the output is a set of scenarios that are presented in descending order based on the accuracy level of the scenario to apply DFLT's.

3.3.1 The Running Example

To assist the reader in following the text, a running example is presented in this part. ArgoUML² will be utilized as an example to illustrate the proposed methodology solution. Specifically, the Activity Diagram feature of ArgoUML will be employed to demonstrate the workings and implementation of the solution. This approach allows for a clear illustration of the required process and an understanding of the intricacies of the proposed solution in a real-world context. The Activity Diagram serves as a step-by-step guide, effectively showcasing how the feature can be exercised by a scenario or a set of scenarios to apply DFLTs.

The Activity Diagram in ArgoUML can be exhibited by several scenarios. A random scenario can be formulated by developers who are unfamiliar with the ArgoUML software as shown in Figure 3.2. It shows the possible steps that are implemented to exercise the activity diagram feature. The users can formulate several scenarios without following a certain template to exercise the targeted feature.

```
Scenario 1: Implementing a Work flow Using the Activity Diagram Feature
1- Start ArgoUML: Launch the ArgoUML application.
2- Create a New Project: From the "File" menu, select "New Project".
3- Open Activity Diagram: Navigate to the "Diagrams" menu and select "Add Diagram",
then choose "Activity Diagram".
4- Add Activity Nodes: Use the toolbar on the right to add activity nodes to the
diagram. This can include initial nodes, activity nodes, and final nodes.
5- Connect Nodes: Use the "Transition" tool to connect the nodes in the order of the
activity flow.
6- Add Decision Nodes: If there are decision points in the activity, use the
"Decision" tool to add decision nodes and connect them appropriately.
7- Label Nodes and Transitions: Click on nodes and transitions to add labels
describing the activities and conditions.
8- Save the Diagram: Once the diagram is complete, save the project.
```

Figure 3.2: Random Scenario to exercise Activity Diagram in ArgoUML.

²<https://argouml-tigris-org.github.io/tigris/argouml/>

However, the developers can enhance the process of formulating the random scenarios by using Gherkin³. Figure 3.3 shows five possible scenarios to exercise the Activity Diagram. All these five scenarios exercise the Activity Diagram feature, but they are formulated to implement different steps.

```
Scenario 2: Creating a New Activity Diagram
Given the user has launched ArgoUML
When the user selects "New Project" from the "File" menu
And the user selects "Add Diagram" from the "Diagrams" menu
And the user chooses "Activity Diagram"
Then a new Activity Diagram should be created

Scenario 3: Adding an Activity Node
Given the user has an open Activity Diagram
When the user selects the "Activity Node" tool
And the user clicks on the diagram
Then a new Activity Node should be added to the diagram

Scenario 4: Connecting Activity Nodes
Given the user has an open Activity Diagram with at least two Activity Nodes
When the user selects the "Transition" tool
And the user clicks and drags from one node to another
Then a Transition should be created connecting the two nodes

Scenario 5: Adding a Decision Node
Given the user has an open Activity Diagram
When the user selects the "Decision" tool
And the user clicks on the diagram
Then a new Decision Node should be added to the diagram

Scenario 6: Saving the Activity Diagram
Given the user has an open Activity Diagram
When the user selects "Save Project" from the "File" menu
Then the Activity Diagram should be saved
```

Figure 3.3: Random Scenarios that are formulated by using Gherkin framework to exercise Activity Diagram in ArgoUML.

There are six versions of scenarios in Figures 3.2 and 3.3. The first

³<https://cucumber.io/docs/gherkin/>

version (Scenario 1) in Figure 3.2 shows the possible steps to be implemented to exhibit the Activity Diagram Feature. In Figure 3.3, a possible improvement applied to formulate scenarios, where each scenario is classified into three steps, Given, When, and Then. The second version (Scenario 2) shows the required steps to create a new activity diagram only. Several steps were eliminated from (Scenario 1). Scenario 2 has some of the steps from (Scenario 1), and others a new steps only implemented in Scenario 2. Similar processes were implemented for every scenario in the running example, where each scenario has slightly different steps than the other.

Each running scenario generates an execution trace. The similarity between each collected execution trace from running scenario can be calculated by using the Cosine Similarity function. The Cosine Similarity is a widely used measure in NLP and IR for quantifying the similarity between two vectors. In the context of execution traces, we can represent each trace as a vector, where each dimension corresponds to the frequency of a specific feature or event within the trace. By treating the execution traces as vectors, we can calculate their Cosine Similarity. The Cosine Similarity is defined as the cosine of the angle between two vectors and ranges from 0 to 1, where 1 signifies complete similarity and 0 indicates no similarity. In the code, the execution traces are read from text files, and the words within the traces are processed to construct the vectors. The program then calculates the dot product and magnitudes of the vectors using the word frequencies. Finally, it applies the Cosine Similarity formula.

The Cosine Similarity formula, represented in LaTeX format, can be written as:

$$\text{Cosine Similarity} = \frac{\sum_{i=1}^n A_i \cdot B_i}{\sqrt{\sum_{i=1}^n A_i^2} \cdot \sqrt{\sum_{i=1}^n B_i^2}}$$

Here, A_i and B_i represent the components of the two vectors, and n denotes the dimensionality of the vectors. The resulting Cosine Similarity

value indicates the extent of similarity between the two execution traces, scaled from 0 to 100.

Table 3.1: Comparison between the collected execution trace that are generated by using different scenarios to exercise Activity Diagram Feature in the running examples

Scenario ID	Feature Name	The Similarity of the Execution Trace Content
1	Activity Diagram	100%
2	Activity Diagram	48%
3	Activity Diagram	52%
4	Activity Diagram	61%
5	Activity Diagram	49%
6	Activity Diagram	63%

In the running example, each scenario generates an execution trace when exercising the instrumented Activity Diagram feature. To illustrate the running example, any changes in the scenario steps, would effect the content of the collected execution traces. In this case, to show the reader how every scenario generates different execution traces, relying on the scenario input, all the generated execution traces were compared with the output of Scenario ID 1. Table 3.1 shows how the output of each scenario can be different than each other by comparing the collected execution traces. For instance, Scenario ID 1 results in 100 % similarity as the execution trace compared to itself in this running example. For the rest of the scenarios, the similarity of the collected execution traces is compared against the collected execution trace for Scenario ID 1.

The similarity results in Table 3.1 were obtained by using the details described in Section 1.8. Particularly, for this running example:

- Generating execution trace: Bytecode instrumentation solution was used.

- Preprocessing: Clean and normalize the texts.
- Vectorization: Convert the texts into numerical vectors.
- Similarity Calculation: Calculate the similarity between the vectors.
- Representation: Multiply the cosine similarity by 100 to get a percentage representation of the text similarity.

3.3.2 Partitioning of the formulated Scenarios

Due to the large size of the generated execution trace when the instrumented software feature is exercised by a scenario, the first step in AETFS is to partition the running scenario into a set of chunks of smaller sizes. Each part of the partitioning scenario aims to generate its related execution trace. Each collected execution trace has varied content of the invoked data granularity. The contents of each execution trace rely on the content of the scenario input, and instrumentation configuration in the exhibited feature. For example, the scenario formulated in Figures 3.2 can be partitioned into eight steps, where each step generates its execution trace. Each scenario in Figure 3.3 can be partitioned into three steps, Given, When, and Then. Each part generates its execution trace.

The granularity of the generated execution trace can be at various levels (See Section 2.2.3), based on the needs/preferences of the user. This can range from the most fine to the most coarse: line-level, class-level, method-level package level.

Certain levels of granularity, like utility data invoked in execution traces, are straightforward. File-level granularity is similarly simple: each scenario part would call upon this data and integrate it into the execution traces within the software system. However, content entities at other granularity levels, such as class, method, and object levels, necessitate a basic instrumentation

method to profile these invoked data. Java bytecode instrumentation, particularly employing a Java agent, is a potent technique for profiling and logging Java software system behavior (Soueidi, Monnier, & Falcone, 2023). This method entails injecting additional bytecode into the loaded classes' methods by the JVM. As the application operates, these instrumented methods are triggered, enabling the Java agent to monitor and record method invocations, class interactions, and object states, with the collected data logged for further analysis in the AETFS approach.

Hence, in the AETFS approach, in this first step, the input is a scenario or a set of scenarios. The input scenarios that aim to execute the targeted software feature are then partitioned into a set of partitioned scenarios. The output of this step) $S = \{s_1, \dots, s_M\}$, where the number of partitioned M depends on the initial formatted Random Scenario Figure 3.2 or Gherkin platform Figure 3.3 of the input scenario, see Figure 3.1 for more details.

3.3.3 Annotating Partitioned Scenarios with the Relevant Execution Traces

This section describes how the collected execution traces is aggregated for partitioned scenarios and how the content of the execution traces are then used to textually represent the accuracy of the formulated scenario in applying DFLTs.

3.3.3.1 AETFS Configuration: Grouping and Classification of the Content of the Execution Traces based on Using Bytecode Instrumentation

The process of grouping and classifying the content of execution traces using bytecode instrumentation is a sophisticated approach in software engineering. Bytecode Instrumentation (BCI) is a method that allows for the insertion of additional bytecode into an application's method. This technique

is used to profile every method, class invoked when a partitioned scenario exercises the software feature (Bézières la Fosse, Tisi, & Mottu, 2018).

In the context of execution traces, BCI can be used to generate detailed information about a program's execution. This information can then be analyzed and grouped based on various criteria, such as the execution path taken, the methods invoked, or the data accessed.

The content of execution traces aggregation groups the collected traces with each part of the partitioned scenarios that exercised the software feature. This grouping depends on the selected level of granularity of the instrumentation technique to profile the invoked data. AETFS can be configured to profile the execution trace content for each running part of the scenario according to the frequency of methods, class, and objects invocation and to ignore the utilities data (de Almeida Maia & Lafetá, 2013).

Finally, AETFS allows for ignoring certain types of collected execution traces. The types of configurations are based on three heuristics methods to determine whether the collected data in the execution traces are aggregated for further analysis or is omitted from the execution trace aggregation.

Moreover, AETFS mainly relies in IR concepts, though the threshold value, which can significantly vary, is often empirically determined based on the specific application, dataset, and similarity measure. Studies have introduced general thresholds for similarity, such as in word embedding for IR, and proposed automatic and online estimations of the similarity threshold for content-based visual IR systems (Rekabsaz, Lupu, & Hanbury, 2017). However, without full access to these studies, specific threshold values remain unspecified. Setting the threshold is a complex task involving a trade-off between precision and recall, typically set based on application requirements and an acceptable balance between false positives and false negatives.

3.3.3.2 Defining Accuracy in the Context of AETFS

In the context of this research, accuracy in the AETFS methodology pertains to the solution's capability to construct accurate scenarios. An accurate scenario denotes the proficiency of generating execution traces essential for applying DFLT's with a high degree of locating the required source code for the exercised software feature by the formulated scenario. The evaluation of accuracy in this study is facilitated through the utilization of three key software metrics: Precision (P), Recall (R), and F-Measure.

The effectiveness of a scenario in the context of DFLT's is fundamental for accurately mapping source code to exhibited software features. Achieving a high level of accuracy in scenario formulation is crucial, as it directly impacts the P, R, F-Measure and overall effectiveness of DFLT's applications. In the following points, we delve into how a high accurate scenario plays a pivotal role in enhancing DFLT performance by facilitating the accurate identification of source code corresponding to exercised features.

A comprehensive explanation of these metrics is elaborated upon in Chapter 4, providing detailed insights into the assessment of accuracy within the AETFS framework.

3.4 The Baseline Approach: DFLT's baseline

A number of existing methods were evaluated as potential baselines for DFLT's. However, they were all ultimately dismissed due to the following reasons:

- The current research in DFLT's (Buzaid & Hammad, 2021b; Razzaq et al., 2020) is focused on developing methods to associate source code with its corresponding F . In contrast, the work presented in this thesis is dedicated to facilitating a singular aspect of DFLT's, specifically

the creation of precise scenarios that demonstrate the desired F . Consequently, the chosen baseline for DFLT is evaluated against those DFLT that implement the AETFS.

- A potential baseline for DFLT was the utilization of grep, akin to the method in (Vale & de Almeida, 2019). Nevertheless, this strategy is unsuitable due to the significant differences between pattern matching and VSM, LDA, and LSI, leading to incompatible results from these two methods.
- The methodology proposed by (Hayashi et al., 2016), including their foundational approach, was not accessible to the public.

The FLAT3 method, as described by (Savage et al., 2010) work, exhibited unsuitable granularity levels, combining elements of sub-program and individual code line granularity while lacking granularity at the file level.

- The adoption rate of TraceLab, as referenced in (Keenan et al., 2012), remains low, with merely three out of twenty-nine papers between 2013 and 2018 documenting the use of this framework.

The innovative aspect of AETFS lies in the analysis of the content within the gathered execution traces. As a result, the decision was made to adopt (Marcus & Haiduc, 2013) methodology for TFLT, while also updating these procedures to align with the latest advancements in TFLT. Specifically, modifications were made to address the unique requirements of AETFS.

- LSI is utilized in AETFS to discern the conceptual correlations between terms in collected execution traces and associated scenarios. By incorporating additional context, such as stereotypes, LSI transcends mere keyword matching, employing statistical techniques to

uncover the latent structure of data, thereby enhancing the precision of selected scenario (Poshyvanyk, Marcus, Rajlich, Gueheneuc, & Antoniol, 2006)

- LDA is applied in AETFS to index execution trace corpora and retrieve scenarios, with various configurations impacting its accuracy and effectiveness (Biggers et al., 2014).
- VSM is utilized in the AETFS approach to map scenario to execution trace by representing both as vectors in a high-dimensional space, enabling the calculation of similarity scores that help identify the most relevant code segments (Zamani, Lee, Shokripour, & Anvik, 2016).
- Execution trace entities were characterized using identifiers, comments, and literals. (Biggers et al., 2014) noted that this particular amalgamation of text data enhances effectiveness. (Alhindawi, Dragan, Collard, & Maletic, 2013) also employed this blend in their research on expanding vocabulary through stereotypes. Conversely, (Marcus, Sergeyev, Rajlich, & Maletic, 2004) limited their description to identifiers and comments alone. Note, that comments are not considered in AETFS as they are not included in the collected execution traces.
- Enhancements in text preprocessing have been made. (Dit, Guerrouj, Poshyvanyk, & Antoniol, 2011) proposed that advanced splitting methods could enhance the performance of TFLTs. Therefore, the TFLTs now include “dot” splitting in addition to the “camel case” and “underscore” splitting initially introduced by (Marcus & Haiduc, 2013). Moreover, when employing file-level granularity, any license information is excluded from the analysis.

The baseline for AETFS adheres to the following procedures to assem-

ble the search corpus for DFLT's to formulate the scenario: (see Figure 3.4):

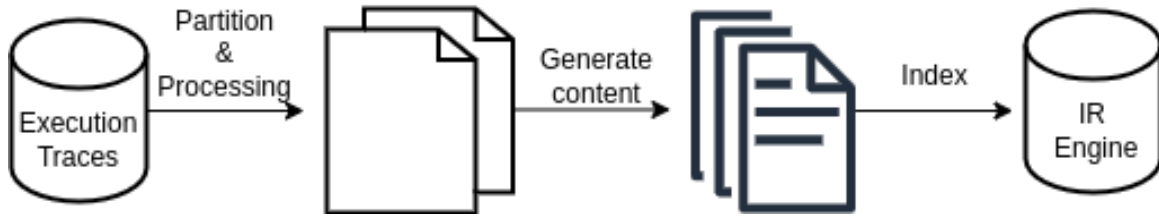


Figure 3.4: The baseline DFLT's approach to index the content of the execution traces

1. **Collected Execution Traces.** Execution traces can be collected to record every method, object, class, and variable that is invoked during the runtime of the software, providing a detailed account of the program's dynamic behavior.
2. **Partition & Processing.** It involves segmenting the collected data into meaningful units and analyzing them to extract insights about the software's runtime behavior (Hendriks et al., 2017).
3. **Generate content.** Each document that details an execution trace entity consists of terms extracted from the entity's identifiers, and literals. The text preprocessing follows the method outlined for AETFS (refer to Section 2.5.7), with the additional removal of programming language keywords. Subsequently, every document within the search corpus is depicted as a collection of words.
4. **Index documents.** The document collection is subsequently organized and preserved utilizing the LSI, LDA, and VSM as an IR framework.

3.5 Implementation details

The AETFS and DFLT's execution relies on external libraries for tracing, parsing, and document indexing through the IR engine. Further details on key operations and supporting libraries are outlined below.

- Bytecode instrumentation in Java, using a Java agent within the JVM, is a potent method for monitoring runtime behavior. It modifies class bytecodes upon loading, enabling comprehensive logging of method invocations and data interactions. This tool, the Brutusin Instrumentation Framework⁴, facilitates this process by capturing method events and notifying custom listeners.
- The text parser tool⁵ streamlines the analysis of execution traces by segmenting content based on specified detail levels. In method-level system partitioning, abstract methods, constructors, and anonymous class methods are excluded from trace entities. Abstract methods lack functionality, constructors set initial states, and anonymous methods are skipped to avoid redundancies within other methods.

To facilitate the research methodology, a specialized tool was necessary for interfacing with the dataset repository, which includes benchmarks (Martinez et al., 2018) for evaluating the effectiveness of AETFS in retrieving pertinent source code. The IntelliJ IDEA Community Edition served as the primary Integrated Development Environment (IDE) for this purpose. Additionally, tools like the Text Parser were employed to process and incorporate the necessary outcomes into AETFS. The application, along with the data from the empirical studies presented in this thesis, can be accessed at the provided GitHub link⁶.

⁴<https://github.com/brutusin/instrumentation>

⁵<https://www.make.com/en/help/tools/text-parser>

⁶<https://github.com/buzaidf/AETFS>

Chapter 4

Experiment and Case Studies

This chapter presents a thorough investigation of the proposed methodology, AETFS, through a sequence of three experiments. These experiments are designed to delve into the efficacy of AETFS in optimizing the process of DFLTs by linking features to source code through execution traces (Buzaid & Hammad, 2021a), sifting out irrelevant inputs via the Scenario Sifter, and harnessing the power of UML diagrams to enrich DFLTs (Buzaid, Albalooshi, & Elmedany, 2022). In doing so, this chapter addresses the research questions concerning the practical application of AETFS and its capacity to enhance the precision of scenario selection in DFLTs, resonating with the core objectives of the study.

The forthcoming trio of experiments has a clear mission: to provide definitive answers to the research questions at hand. Through a systematic approach, each of these experiments is poised to delve into specific facets of the study's objectives, offering insights and conclusions that collectively address the core inquiries driving this research endeavor. By meticulously

examining the outcomes of these experiments, the study aims to not only shed light on the efficacy of the AETFS methodology but also to offer valuable contributions to the DFLTs.

The primary objective of the first experiment, "Mapping features to source code based on execution trace," is to investigate the effectiveness of using reformulated queries to assess the accuracy of the DFLT's input scenarios in applying FL. By thoroughly analyzing the collected execution traces, this experiment aims to establish a direct correlation between features and their respective source code sections. The accurate identification of the source code sections associated with specific features offers valuable insights into improving the feature-to-code mapping process. Furthermore, by leveraging reformulated queries, this experiment seeks to evaluate how well the DFLT's input scenarios align with the accurate application of FL, contributing to the assessment of their effectiveness in ensuring precise and targeted testing coverage.

The second experiment, "Scenario Sifter to Filter Irrelevant Inputs in DFLT's," investigates the application of the Scenario Sifter, a component of the AETFS methodology, to filter out irrelevant inputs during DFLT's. The Scenario Sifter aims to optimize the testing process by identifying and excluding test scenarios that are unlikely to contribute to the effectiveness and coverage of DFLT's, improving the overall efficiency of the testing process.

The third experiment, "The Use of UML Diagrams to Enhance DFLT's," explores the utilization of UML diagrams as a tool to enhance DFLT's. By incorporating UML diagrams into the testing process, this experiment investigates their potential to provide a visual representation of the system's structure and behavior, aiding in the identification of relevant test scenarios and improving the accuracy of DFLT's.

Through these three experiments, this chapter seeks to evaluate the effectiveness of the AETFS methodology in enhancing the process of DFLT's.

The results obtained from these experiments provide insights into the practical application of AETFS and its potential to improve the selection of accurate scenarios for DFLT, ultimately contributing to the overall quality and reliability of software systems.

4.1 Mapping features to source code based on execution trace

The ultimate aim of any DFLT is to map software feature to the relevant source code. Allocating only relevant certain methods from execution trace to the related feature is the main difficulties in DFLT due to the size of the collected execution trace. In this experiment, we present a DFLT (experiment case study called MappFL) to map the relevant methods to the software feature based on analysing the actual behaviour of the system. The MappFL refers to use auto-reformulating query technique to extract the relevant methods from the execution trace which was generated by executing one scenario that exhibits a desired feature. Distance software metrics are proposed to measure the distance between the extracted methods by DFL techniques. A case study was implemented to map software feature to the relevant methods in the source code.

4.1.1 Overview

First configuration of AETFS is to use one single scenario to generate execution trace and use reformulating text query to specify the input of DFLT. In order to increase the effectiveness of feature location, both static and dynamic techniques have to be implemented to solve the problem of mapping the source code to the relevant software feature. In this experiment, a dynamic technique generates a huge size of the execution trace that contains a lot of noise, which refers to irrelevant or extraneous information that is cap-

tured during the execution of the software system (Maalej, Tiarks, Roehm, & Koschke, 2014). This noise can manifest as data points, events, or behaviors that are not directly related to the specific features being analyzed or the desired software feature (Dit, Revelle, & Poshyvanyk, 2013).

There was a need to utilize static techniques to filter the execution trace events to eliminate the noise. In addition, a query was constructed that describe feature in natural language to retrieve the data from the execution trace. Information retrieval techniques, such as LSI is used to index the query to the relevant data (feature's source code) only and only if the query was formulated correctly.

In this experiment, a technical approach was implemented to map a software feature to its relevant source code elements (that is, classes, methods or functions). The technical experiment, called feature localization, incorporates static and dynamic feature location analyses. The idea of the dynamic analysis is based on instrumenting the source code (Hammad & Al-Hawawreh, 2018) to profile all methods that are invoked when developers execute a scenario that exhibits a certain software feature. Whereas, the idea of the static analysis is based on filtering the logged execution trace to extract the relevant pieces of source code by using textual analysis (Liu, Marcus, Poshyvanyk, & Rajlich, 2007). Furthermore, developers input a natural language query to describe the feature of interest, aiding in the extraction of data from the execution trace. The entered query is reformulated based on using Refoqus (Haiduc et al., 2013), which is an automatic query reformulation technique that is used to support the query to filter the execution trace to retrieve list of ranked methods. The obtained result as ranked list of the methods is verified by using distance metrics (Wong & Gokhale, 2005; Wong, Gokhale, & Horgan, 2000) to measure the distance between these methods to confirm that all retrieve methods are close the each other and they are relevant to the software feature.

4.1.2 Design The experiment

The main steps of mapping software feature to the relevant methods are shown in Figure 4.1. The depicted figure has three processes. Each one of them receives at least one file as an input, and outputs one file as a result of completing process. There are two main initial inputs into the proposed solution, a scenario, and a source code. The source should be compiled and run when a scenario is executed. The developer has to generate a scenario that exhibits a feature in the software system, and the results are used as input in Generating Execution Trace Process. The scenario is a set of events that are generated by the developer to exercise a desired software feature. For instance, a list of events have to be created to construct a scenario that exhibit a certain feature. Next, the Generating Execution Trace Process profiles all the methods and method's content that are invoked during executing a scenario to output a file called execution trace. The execution trace file contains on both relevant and irrelevant methods including method's statement that are profiled during running a scenario. Next, both files, query and execution trace are used as an input to the next process. Query reformulation and filter execution process outputs a list of ranked methods that are used as an input to the final process where to measure the distance between these ranked methods on the list. When the process of measuring distance between methods are completed, the system outputs a file contains a list of methods that are used to construct feature that where described by developers in the entered query. The following subsections provide details in depth for each process.

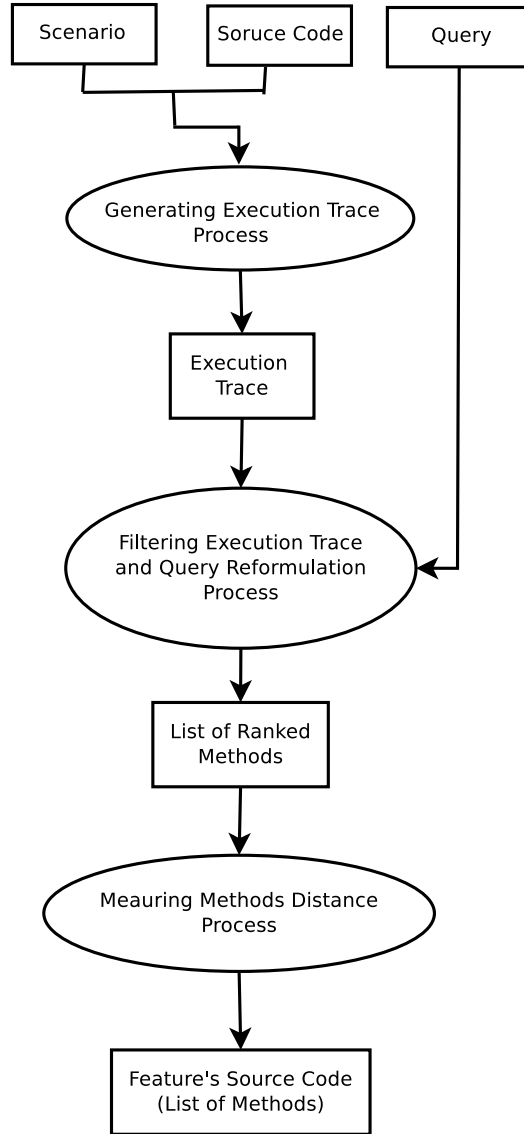


Figure 4.1: The main steps of designing the experiment to map feature to the source code by using execution trace.

4.1.2.1 Generating Execution Trace Process

Instrumenting source code is required to generate and profile the execution trace for a running program. We propose to use a lightweight toolkit called srcML (Collard, Decker, & Maletic, 2013) to profile every methods that are invoked, including methods statement, when a scenario exhibits the desired feature. SrcML tool is selected to profile the execution trace, because it is

easy to use, and provides flexibility in profiling files individually or entire the system. Figure 4.2 depicts the main processes that are needed to generate the execution trace which are presented as follow:

Convert the source code to srcML format process is received a source code in an original format to output the source code in XML format. This translation is done for all the original source code for for a running system to have the source code presented in XML format. For example, all the files with the extension Java are translated to files with the extension *XML*. The resulting of completing this process is an XML files which are utilized in next process. For instance, a file called *app.java* is converted to *app.xml* by using the command: *src2srcml app.java -o app.java.xml*.

XML transformation is required to perform instrumentation. The converted XML is analysed by using XPath. Any element in the XML format can be selected and extracted by using XPath. Insert profiler code process uses a basic *XSLT* file that contains instruction of instrumenting source code to collect information from the XML files. For instance, the *profiler.xslt* has the required script to manipulate the data on XML files which are xml files. The output of this process is an instrumented XML files that has all required information to hook all the methods and and methods' content in the system. Hint, *profiler.xslt* manipulate the *xml files* to profile all methods including the content between the method's entry and method's exit, i.e entry and return for each method.

Convert instrumented XML format to original source code format is the last process to generate execution trace. The instrumented XML code are translated to the original source code format with additional instrumentation code to profile a running scenario's events. For instance, the instrumented XML file called *app.java.xml* is converted back to *appV1.java* by using the command: *src2srcml app.java.xml -o appV1.java*. The file *app.java* is identical to *appV1.java*, except the *appV1.java* is an instrumented file to monitor

the the program at runtime. At this process, we have an instrumented running program and the content of the execution trace is a result of exercising a feature bases on the entered scenario. For instance, a scenario is used as an input to exercise a software feature, and all the methods and statements that are invoked during the execution are profiled on a file. This file is called execution trace, and this is the output of the convert srcML format to source code.

Hint, the main output results of the generated execution trace is the execution trace file. It a text file which contains on the list of all methods that are invoked when execute a scenario to exhibit a desired software feature.

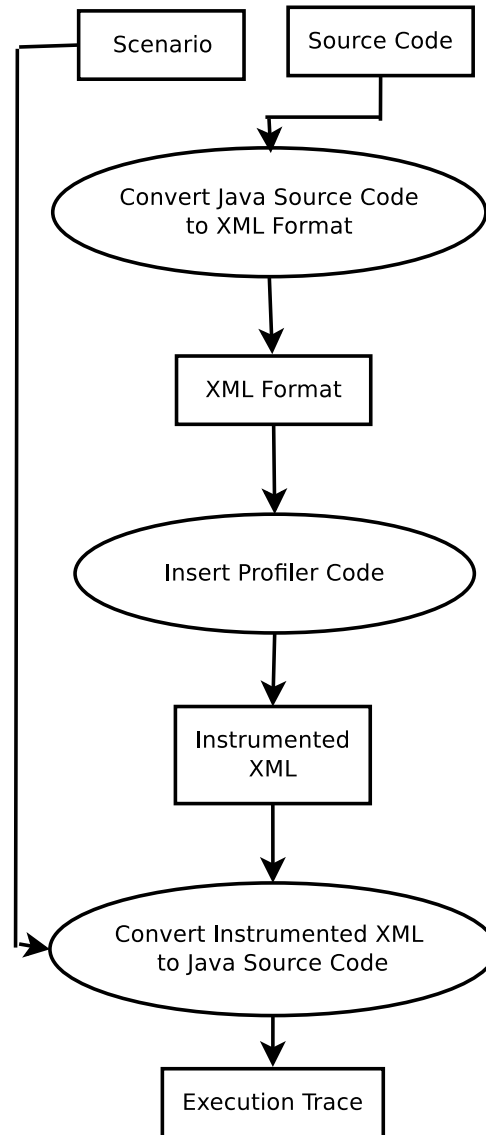


Figure 4.2: Steps of generating execution trace process.

4.1.2.2 Filtering Execution Trace by Query Reformulation

Analyzing the generated execution trace relies on several steps. Static analysis dominates the the analysis techniques. The size of the execution trace is a huge as the events that profiled such as methods and statements would include many irrelevant data which called noise, and only part of them are considered to be relevant to the feature of interest. Filtering the execution trace is based on many factors such as the quality of the entered query, and

how to they can be used to extract information based on the IR techniques. The following subsections illustrate how creation corpus, entered query, reformulate query and uses IR techniques such as LSI, results to output a list of ranked methods that are used to construct the software feature.

Corpus creation is the initial step to categorize the execution trace. In this step, all methods are extracted from the execution trace, and represented as documents. Each document indexes to the content of the document. For instance, a method A is assigned as a document A. The content of the document A is indexed with the content that were profiled from the method's entry to the method's exit. Figure 4.3, shows each method corresponds to a document, and each of a document is indexing to terms which are a content of the method. In other words, the created corpus contains list of methods in the execution trace, where each method corresponds to a document. Each document includes method name, name of variable, function calls, identifier names, and statements. For instance, when srcML profiles n number of methods, that results the corpus to have n number of documents. Each document indexes the content of the methods from the method declaration to method return.

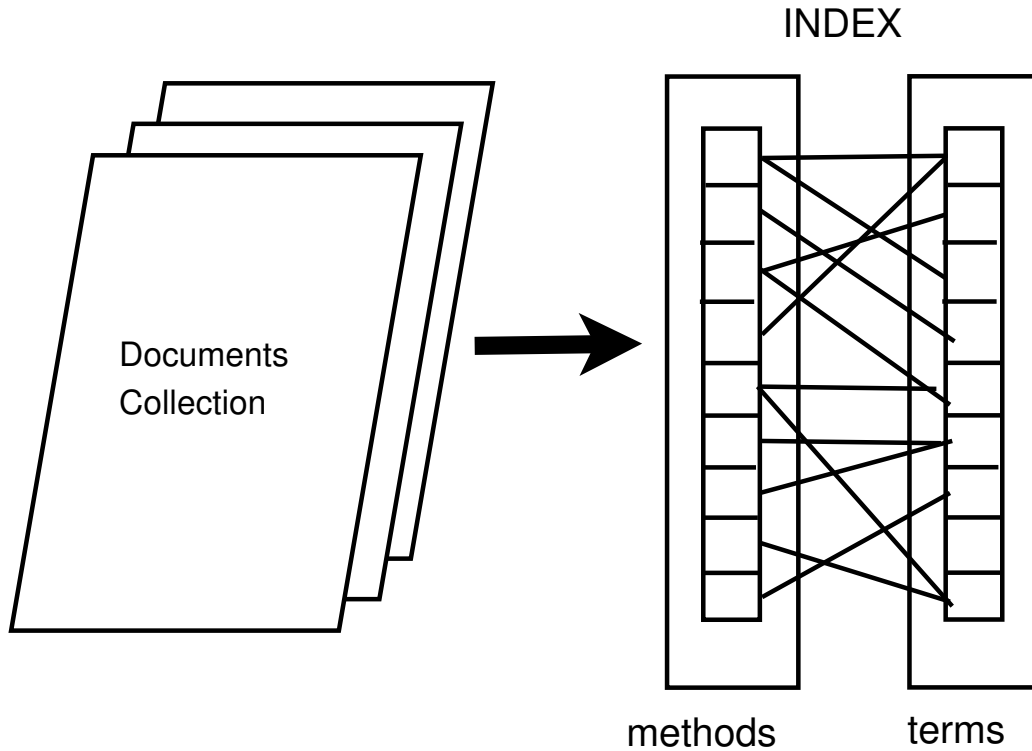


Figure 4.3: Corpus indexing process.

Query reformulation techniques are used to search from the created corpus. For instance, the developers write a query to locate relevant methods from the created corpus by using (LSI) to retrieve a document that has a matching string between the document and the entered query. If the query does not have a string matching in the created corpus, the retrieve result would be effected. Hint, the quality of the entered query plays an essential role to determine the quality of the retrieved list of ranked methods. Semi-auto technique in (Lu & Keefer, 1995) are used to expand or reduce the initial query.

Query expansion refers to use addition terms for short query. For instance, retrieved information is effected when the important terms are not included in the search query. Three strategies are used to expand query based in (Haiduc et al., 2013), such as Similarity-based (Rafiei & Mendelzon, 1997), Rocchio's method (Rocchio, 1971), and Robertson Selection

Value (RSV) (Robertson, Walker, Beaulieu, Gatford, & Payne, 1996).

Similarity-based is the first strategy (as shown in Equation 4.1) is used to rank the top M documents and order the their terms based on the Dice similarity between them based on the individual query words. The equation can be presented as follow:

$$Dice = \frac{2df_{x \wedge y}}{df_x + df_y} \quad (4.1)$$

where x denotes a term of the query, y donates a term of the top M documents, and df illustrates the number of documents in the corpus, that containing x , y , or both x and y . The Equation 1 results the two terms are relevant only and only if the two terms present in the same document.

Rocchio's method is the second strategy (as shown in Equation 4.2) is used to locate a sum of the *Tf-Idf* scores as a metric for each term in the top M document. Finding number of times the term is appeared in document is an indicator to show the important of terms based on the document. Term Frequency (*Tf*) refers to the number of times the term shows in a document, and the Inverse Document Frequency (*Idf*), refers to the inverse of the number of documents in the corpus containing that term and illustrates the specificity of that term for a document that having it. The equation can be presented as follow:

$$Rocchio = \sum_{d \in Z} TfIdf(w, d) \quad (4.2)$$

where Z indicates the set of top M related documents in the list of retrieved result, while d indicates a document in Z , and the indication of term in d is referred by w .

Robertson Selection Value (RSV), is the third strategy (as shown in Equation 4.3) is used as a function to order the terms in the top M docu-

ment. The equation can be presented as follow:

$$RSV = \sum_{d \in Z} Tfidf(w, d)[p(w|Z) - p(w|O)] \quad (4.3)$$

where the collection of documents in the corpus is indicated by O , the set of top M related document in the list of invoked result is indicated by Z , d is used to indicate a document in Z , and a term in d is indicated by w .

Query reduction refers to eliminate some data from the query to increase the invocation quality based on better matching of the textual retrieval engine. The reduction strategy in this approach suggests to delete the terms that show in more than 25% of the document in the terms corpus based on non-discriminating (Gay, Haiduc, Marcus, & Menzies, 2009).

Query and corpus keyword synonyms refers to add the synonyms of the terms to enrich the created corpus in to enhance the the process of feature location. The created corpus in Section 4.1.2.2 is linked to WordNet (Miller, 1995), to obtain the synonym for the entered query by the developer to have a bigger search area in lexical database.

4.1.2.3 Measuring Methods Distance

The result of filtering execution trace is a list of ranked methods that are relevant to the feature of interest. A further analysis takes place in this process to find the distance between the ranked methods. The hypothesis in this section proposes that the close distance between methods, re-orders the methods to be on the top ranked list. In other words, the top ranked methods are re-measured by using distance metric to confirm the accuracy of the ranked method result in Section 4.1.2.2. The result of using distance metrics may re-order the list of ranked methods that obtained by using query reformulation technique. Static metrics and dynamic metrics are used to measure the distance between ranked methods.

Static Metrics refers to provide a distance between two methods based on the methods implementation point of view. Such as finding the similarity between methods implementation based on the methods lexicon.

Static analysis using similarity distance, the proposed technique involves several steps to find similarity between methods and rank them (Desnos, 2012). First, each method is represented using a suitable representation (such as abstract syntax trees or control flow graphs), and relevant features are extracted. Next, a customized similarity distance is computed between pairs of methods, considering both syntactic and semantic similarities. Higher similarity scores indicate greater similarity. Finally, methods are ranked based on their similarity scores. This approach can be applied to identify methods similarity, enhancing static analysis and improving the list of ranked methods.

Dynamic Metrics refers to provide a distance between two methods based on the execution point of view. Such as using the number of time the method is invoked and the execution time for methods to be executed.

To apply the dynamic analysis, the ranking and determining similarities between methods, one can follow a JIVE approach (Lessa, Czyz, & Jayaraman, 2010). Initially, compile a comprehensive dataset of Java programs with varying complexities. Subsequently, utilize the JIVE tool to visually trace the execution flow of these programs, observing how methods interact and execute within the codebase. By analyzing the visualizations produced by JIVE, patterns in method execution can be identified, aiding in the ranking of methods based on their frequency of interaction, depth of invocation, or execution paths. Additionally, similarities between methods can be inferred by comparing their visual representations, highlighting commonalities in how they interact with one another during program execution.

4.2 Scenario Sifter to Filter Irrelevant Inputs in DFLTs

Formulating accurate scenarios to exercise the software feature is essential in carrying DFLTt. The properties of the formulating scenarios play a vital role in all dynamic analysis. The size of the execution traces depends on the quality of the DFLTt' input. In this second experiment, a practical work is called a Scenario Sifter was designed and implemented. It aims to support the software programmer in selecting the accurate scenario to exercise the exhibited software feature. The experiment is based on the Online Synchronous Instrumentation Approach (OSIA) to process the scenario's input as soon as the input is executed to exercise the software feature. The Scenario Sifter excludes the irrelevant parts of the executed scenarios using IR techniques. The accuracy of the proposed approach is evaluated against three open-source Java projects, Sudoku, Notepad, and ArgoUML-SPL. The evaluation is based on comparative analysis to evaluate the effectiveness of using Scenario Sifter with DFLTt.

4.2.1 Overview

The second configuration of the AETFS is to divide the single scenario several sub-scenarios to specify the input of DFLTt. Nowadays, all software systems are subject to updates and maintains during the software's life cycle. Developers use supportive techniques to comprehend the software systems before implementing the required update. Dynamic Feature Location Techniques (DFLTt) are used as supportive techniques to map the software features to the relevant source code based on dynamic analysis. One difficulty of using DFLTt is to allocate the required input of the required scenario to exercise the software feature. It is time-consuming and error-prone for the software developers to manually formulate all possible DFLTt' input

to exercise the feature of interest in a large software system. The software developers must carefully prepare the DFLT's input as the generated execution traces rely on the formulating scenario. Currently, most of the DFLT's use offline instrumentation techniques to collect execution traces rather than using OSIA (Zhang, Gebhard, & Sokolsky, 2016). The offline instrumentation approach forces the developers to investigate the entire execution traces when the scenario is fully implemented. Unfamiliar developers with the system may apply inaccurate or unneeded scenario input, resulting in erroneous execution traces. Hence, the accuracy of performing DFLT's is sensitive to the input data on the system (Bittner et al., 2022). Any change in the scenario's input of the system affects the execution traces output on the DFLT's. Therefore, the developers must formulate the exact scenario to exercise the targeted software feature.

In a complex software system, the software maintenance teams seek a solution to enhance the mapping process between the software features and the source code. DFLT's accomplish the mapping, although preparing the accurate scenario is a difficult task in a large system. Therefore, proposing a solution to locate the appropriate input in DFLT's is essential to optimize the generated execution traces. Solving the data input problem eases the difficulties in developing the accurate required execution traces in DFLT's. There is a need for DFLT's to find a solution to filter the input data automatically to increase the assurance of having an exact scenario. Exercising a feature using a correct input can result in more precise execution traces with fewer false positives and more true positives and enable a more effective and efficient mapping between the features and the source code. In addition, using OSIA to profile a software program would help the software developers to ease the difficulties in processing DFLT's. It limits the cost of generating a large execution trace since the running input is partitioned into several sub-inputs. Early detecting whether the parts of the DFLT's input are related to

the feature or not reduces the burden of analyzing a large execution trace (de Carvalho, 2019).

This subsection shows a technique called Scenario Sifter to enhance the process of DFLT's. It supports DFLT's input to automatically determine the required parts of the scenario's input to exercise the feature of interest. It automatically discards the irrelevant parts of the scenario's input, such as deleting unrelated factors in the scenario that may be used to exercise the target software feature. The proposed technique is developed based on OSIA (Zhang et al., 2016), dynamic, and textual analysis. The OSIA (Zhang et al., 2016) generates the execution traces synchronously based on independently executing each part of the DFLT's input. It results in an execution trace for each part of the scenario's input. In addition, textual analysis is applied for each collected execution trace to determine whether or not the implemented running part of the scenario's input is relevant to the exercised feature. According to the characteristic of the Scenario Sifter, the implemented scenarios' input in the DFLT's presumes to be through the system's GUI in this work. A possible extension can execute Scenario Sifter input through keyboards, text queries, test cases, or any other data input format in DFLT's.

4.2.2 Experiment Design

The aim of this experiment is to create a prototype tool called a Scenario Sifter. Figure 4.4 shows the main process to design the tool. It aims to delete irrelevant parts of the scenario to increase the quality of the DFLT's input. The high-level requirements to use Scenario Sifter are a compiled program and data input (e.g., scenario and text query) to exercise the software's functional features. The proposed approach is based on the OSIA (Gosain & Sharma, 2015) to collect execution traces independently for each part of the running scenario. In addition, three IR techniques, including a

query enhancement solution, are implemented in the Scenario Sifter to provide the final output for selecting the relevant part in the scenario input.

Generally, the Scenario Sifter decomposes the scenario input into a series of sub-scenarios based on the number of mouse clicking to generate a series of execution traces. Only one sub-scenario (implemented by two mouse clicks) is executed simultaneously. The remaining sub-scenarios stay blocked unexecuted until monitoring the currently executed sub-scenario is completed. Suppose the executed sub-scenario is related to the exhibited feature. Then, the executed sub-scenario is stored in a specific component on the Scenario Sifter system to be used in a further stage to implement the right scenario to exhibit the software feature. Otherwise, the currently investigated sub-scenario gets dropped from the system, and another sub-scenario from the main scenario gets processed.

As shown in Figure 4.4, Scenario Sifter is a component-based solution (Tiwari & Kumar, 2020), whereby each component refers to a module of classes that represents independent system or subsystems with the ability to interface with the rest of the system. The Scenario Sifter system is based on the OSIA approach to synchronously profile independently each sub-scenario which is a sub-set from the main scenario to determine its relevancy to the exhibited feature. In the following subsections, we provide details in-depth for each component in the system. The symbol notations we use are listed in Table 4.1.

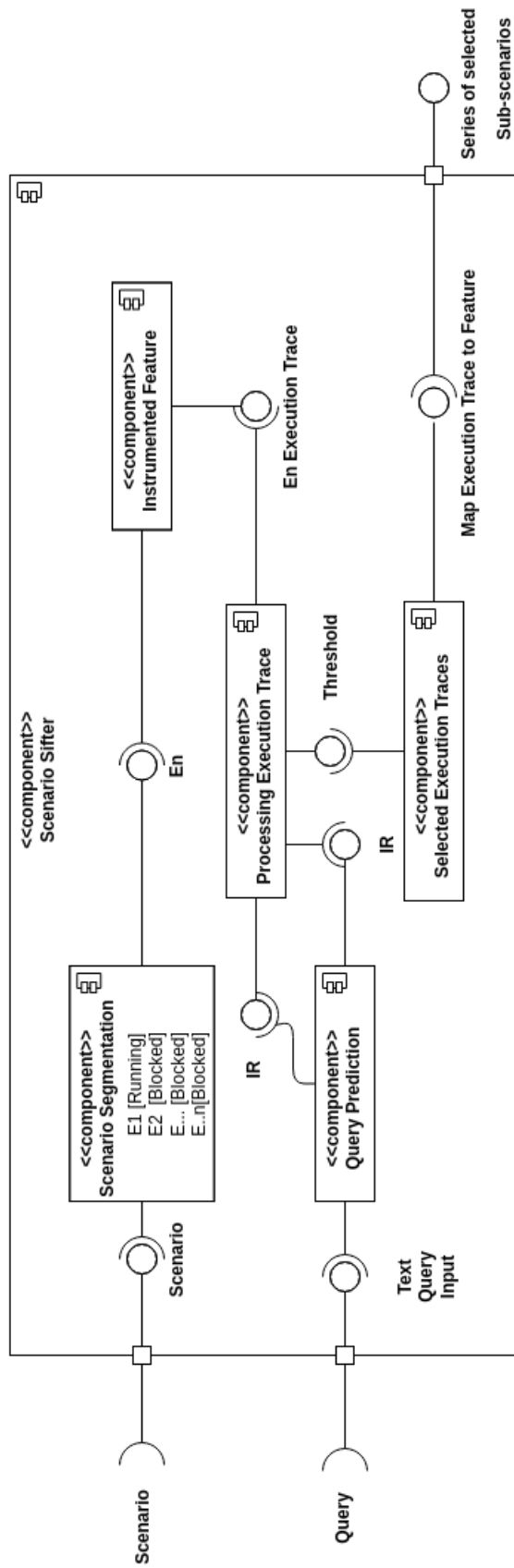


Figure 4.4: Scenario Sifter Proposed Methodology Diagram to Filter Irrelevant Scenario's Inputs.

Table 4.1: The basic symbols and definitions used in Scenario Sifter

Symbol	Description
S	Refers to a Scenario, representing the total actions needed to trigger a feature on the system using the GUI.
F	Indicates a Feature of the software system, a functional requirement exercised by a scenario S .
T	Represents the execution trace generated when a feature F is exercised by a scenario S .
Q	Denotes a query formulated with terms describing feature F using natural language.
MC	Represents the required number of mouse clicks to exercise feature F .
f_1, f_n	Subset of all features F that may be exhibited by a subset of scenarios S . $(f_1, f_2, \dots, f_n) \subseteq F$.
s_1, s_n	Subset of all scenarios S used to exercise feature F or parts of feature F . $(s_1, s_2, \dots, s_n) \subseteq S$.
t_1, t_n	Subset of all execution traces T . $(t_1, t_2, \dots, t_n) \subseteq T$.
mc_1, mc_n	Subset of all mouse clicks MC used to implement a scenario S to exercise feature F . $(mc_1, mc_2, \dots, mc_n) \subseteq MC$.

The final output of the Scenario Sifter is a set of sub-scenarios mapped to the relevant execution trace to illustrate the critical part of the scenario. Figure 4.5 shows the final output of the Scenario Sifter to filter the initial scenario input.

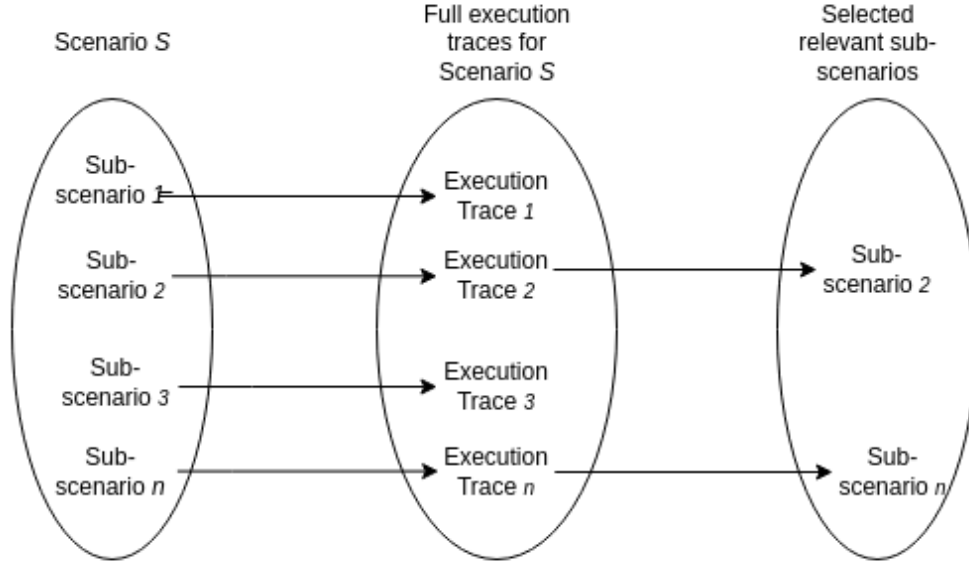


Figure 4.5: The final output of the Scenario Sifter to show important part of the initial Scenario.

Hence, each sub-scenario in the Figure 4.5 indicates two mouse clicks. In the following subsections, full details are provided in-depth for each component in the system.

4.2.2.1 Data input in Scenario Sifter system

Figure 4.4 shows, there are two required inputs to perform the proposed methodology, scenario, and query.

Scenario S is a set of actions that can be executed to exercise the targeted software feature. Due to space limitation, S is defined to be a series of action that can be implemented through mouse-clicking to exercise the feature of interest via using GUI. Generally, the S input is decomposed into a series of sub-scenarios (*e.g.*, $s_1, s_2, \dots, s_n$). Each s_n invokes an instrumentation technique (Gosain & Sharma, 2015) to instrument a certain part of the feature f_n independently than the rest of the entire targeted feature F , generating an execution trace T_n that related to the executed s_n . Hence, s_n

is executed when f_n is exhibited by mouse clicking MC_n which refers to s_n . Additional mouse click MC_{n+1} executes the second sub-scenario s_{n+1} to exercise f_{n+1} , and it terminates the current process of the generating T_{n+1} , to start generating T_{n+2} process. Hence, the S can be terminated at anytime by right click on the mouse. The developer can navigate through GUI to implement S by using a MC , or a touchscreen in case of using touch panel GUI.

Query Q comprises a collection of terms crafted using a Natural Language (NL) (Moreno et al., 2013) methodology to depict the targeted feature. Its primary objective is to determine the relevance of the executed segment of scenario s_n to the specified feature F . The constructed Q employs Textual Retrieval (TR) (Khatiwada, Kelly, & Mahmoud, 2016) strategies to ascertain the pertinence of the executed s_n to F following an examination of the associated execution trace T_n . Moreover, the Query Quality Prediction (Q2P) (Mills et al., 2017) technique is leveraged to enhance the effectiveness of the IR methodologies utilized.

The purpose of Q2P in the context of Scenario Sifter tasks involving TR is to automatically assess the quality of queries before analyzing the results. By leveraging machine learning techniques and a set of query properties, Q2P predicts whether a query will retrieve relevant documents effectively. This prediction aids software engineers in optimizing their formulating queries that used to describe the software feature, distinguishing between high and low-quality queries to streamline their workflow. Q2P's application extends to concept location and traceability link recovery tasks, where it guides developers on inspecting or reformulating queries based on their quality, ultimately enhancing the efficiency and accuracy of TR-based software engineering practices.

4.2.2.2 Scenario segmentation

The input of the formulated scenario is based on using the system's GUI. The scenario input is decomposed into several of sub-scenarios. Each sub-scenario exposes a particular part of the targeted feature. For instance, the scenario usually consists of a series of sub-scenarios, *e.g.* a feature in Microsoft Word called "Save As", it can be exercised by a scenario, "*Start Application, click on File, click on Save As, choose distention to save the file, click on Save*". This scenario is formulated based on five sub-scenarios correlated with the mouse clicking. Hence, some sub-scenarios may consist of others iterative sub-scenarios, such as choosing a file destination, which needs more than one mouse click to proceed with it. Each time the developer acts, *i.e.*, mouse-clicking, a session is initiated to start profiling, and the following mouse-clicking triggers the end of the current profiling session to form a new profiling session if exists. For illustration, the first mouse-click sends a signal to launch the instrumentation technique to indicate the event's start. The second mouse-clicking sends a signal to stop the instrumentation technique and relaunch it for the second event if the scenario is not terminated by right mouse clicking.

The output of the scenario segmentation component is a series of signals to control the instrumentation technique. Each signal includes a unique identification number used to map the current executed sub-scenario when the feature is exhibited to generate an execution trace. The following Sub-sections 4.2.3.3 illustrates how the output of the scenario segmentation component controls the instrumentation technique to exercise the software feature based on the OSIA as shown in Figure 4.4.

4.2.2.3 Instrumented Feature

Instrumentation technique can be used to trace the software application behavior at runtime. Several techniques are available for tracing user interface

events (Di Martino, Fasolino, Starace, & Tramontana, 2021). An instrumentation approach is required to specify the locations in the targeted application to capture the invoked information during execution time based on using mouse clicking. Mouse clicking can be used to control the instrumentation techniques by creating a Mouse Listener (*MouseEvent.getButton() method in Java*) to manage when the instrumentation starts and ends generating execution trace. For instance, an execution trace T_n is generated for every two mouse clicks MC . Hence, the number of the generated Execution Traces T by implementing the entire S is around $50\% \leq$ the total number of the Mouse Clicks MC when F is exhibited by a series of S_n . The following equation illustrates the number of the execution traces that expected for mouse clicks:

$$T_n = \frac{MC_n}{2} \quad (4.4)$$

where T_n indicate the number of execution trace, and MC_n refers to the total number of mouse clicks.

The targeted feature should be instrumented so the result of every mouse clicking is captured to start profiling all invoked data between two mouse clicks. There are several open source instrumentation techniques can be used to generate execution trace based on tracing program at bytecode level (Horváth et al., 2019; Soueidi, Monnier, Kassem, & Falcone, 2021). However, Scenario Sifter aims to provide a solution for Java¹ programming language on this experiment. Therefore, instrumentation approach is performed at the bytecode level, in stack-based virtual machines like the JVM. Each invoked data is represented by an instruction pushing a value on the operand stack, which can be used with an advantage. Technically, instrumentation at the bytecode option has the advantage of not requiring the system to be under analysis before its launching. Currently, the invoked methods, classes,

¹<https://docs.oracle.com/javase>

and variables are profiled to make the most scenes to create the required data that can be used in the later phase on Scenario Sifter system. Methods execution events are recorded when a method body is executed, while method Call event refers to provide method dependencies (Hilsdale & Hugunin, 2004). For instance, the invocation of the methods and the classes are profiled in the instrumentation models, without monitoring the dependencies at this case. For the variable, the profiler monitors the details information of all variables defined or used at statement. Each variable is indexed with its definition or use, in addition to object address of array and any indexing information. Moreover, variable values are computed at runtime, and indexed in the execution trace to enhance the information when IR technique used in later stage.

Moreover, the instrumented feature by using Java Implemented VERGE Editor (JIVE)² produces an execution trace that corresponding to exhibited part of the feature. Another recommended approach that can be used in this case is called Soot framework (Vallée-Rai et al., 2010). It profiles a unique identifier of each execution thread (utilizing `Thread.getId()`), and for each statements within the thread. In addition, it provides a low-level bytecode tracing, and intermediate IR manipulation. The collected execution trace monitors methods, classes, and variable-level that corresponding to every independent T_n for each S_n .

the Soot framework plays a significant role in VERGE by offering advanced profiling capabilities and facilitating detailed analysis of software execution traces. Soot is integrated into the workflow to enhance the instrumentation process alongside the Java Implemented VERGE Editor (JIVE). By leveraging Soot, unique identifiers for each execution thread are profiled using `Thread.getId()`, enabling precise tracking of statements within

²JIVE is an open-source plug-in that can be integrated with Integrated Development Environment (IDE) such as Eclipse to provide the programs execution trace with several other execution trace feature.

threads. Soot's functionality extends to providing low-level bytecode tracing and intermediate IR manipulation, thereby enhancing the granularity of the collected execution traces. This comprehensive monitoring allows for the identification and analysis of methods, classes, and variables at different levels, aligning closely with the generation of independent execution traces T_n for each S_n within the software feature under examination.

The proposed instrumentation technique should be packaged into executable file, aggregating many Java class files and associated metadata and resources into one file for distribution. The Java instrumentation at bytecode level can be implemented as a Java agent working inside JVM. At the runtime, the Java agent monitors class loading events, and logging instrumentation is injected into the loaded classes using a Java bytecode manipulation framework, such as ASM library³. The injected logging instructions in the Java agent are executed as a part of the target program. Note that, the standard library classes (*e.g.*, java and javax package) or any other libraries can be executed from injection to prevent overhead instrumentation.

The opted instrumentation technique must be safe, efficient, and preventing runtime overhead. The instrumentation tool is converted to executable JAR file to instrument Java classes when they are loaded by JVM. The instrumented executable JAR file is controlled to start and end the instrumentation to start generating execution trace by using mouse click (*e.g.*, Mouse Listener (MouseEvent.getButton()) method) in Java. The instrumentation process should not result to alter the functionality behavior of the target application. In addition, the instrumentation technique must generate an execution trace to show all invoked data. The output of this component is T_n which is corresponded to S_n . Hence, the collected execution trace is based on OSIA environment. Each T_n begins under analysis as soon as s_n executed.

³<https://asm.ow2.io/>

4.2.2.4 Processing Execution Trace

Analysing the collected execution trace is an essential step in DFLTs. The first step in processing execution trace component is to load the execution trace file for analysis. The execution trace contains low-level information such as methods calls, classes, variables, or any other events. The collected execution trace is parsed to obtain the required data to preform TR analysis to determine its relevancy to the input Query (Q) that describes the feature of interest. The execution trace is defined as the collection of the software artifact that retrieves important information regarding to the software execution. The file contains the lines executed from the source code when the scenario exhibits the feature of interest. Each scenario consists of a series sub-scenarios. The output of each executing sub-scenario consists of text files named by the package and class executed. Each execution trace contains the number of lines executed within the source code of the individual running sub-scenario.

The execution trace usually contains utility methods and repetitive methods that are invoked during executing a scenario, *i.e.* a sub-scenarios in this case. The distribution of frequencies is applied to filter all the utilities methods, such as the method invoked when clicking on the mouse, that imported from packages and libraries. Moreover, Run Length Encoding (RLE) algorithm is implemented to reduce the size of the execution trace by removing the replication between the methods in the execution trace (Khoury, Shi, & Hamou-Lhadj, 2016).

The following subsections describe how the collected execution traces are processed to select the relevant sub-scenarios to exercise the feature.

Term Corpus

A term corpus is required to be created to analyse the execution trace, and extract the required information to perform Scenario Sifter. The text content on the execution trace is extracted, where each document corre-

sponds to a method or class. The text corpus is required to create a term-by-document co-occurrence matrix to retrieve the data based on the queries in NL that describe the feature of interest (Arcega et al., 2017).

Several preprocessing steps are usually required to be performed before the term corpus phase in contextual representation. The steps commonly used are (Binkley, Lawrie, Uehlinger, & Heinz, 2015; Wang, Peng, Xing, Fu, & Zhao, 2017):

- **Splitting:** discrete tokens into basic words based on ordinary coding style conventions such as CamelCase and Naming conventions split
- **Normalizing:** change each upper case letter with the analogous lower case letter
- **Filtering:** delete stop words such as articles, or short words. In this case, delete all the profiled java packages such as *java.lang*.

After the extraction, a document is obtained for each method of product variant in raw text format. The obtained corpora are used to evaluate whether the generated execution trace is relevant to the feature or not based on the input query that is enhanced by using a prediction system (Mills et al., 2017) to indicate the quality of the query in retrieving the relevant information from the created term corpus when IR techniques used.

Query Prediction

Scenario Sifter requires to use a query as an input when the feature is exercised. The developers enter text query that describe the feature of interest. The query is formulated manually by using selected NL inspired approach. The term query quality refers to the ability of the query to retrieve the relevant software artifact, and the retrieval data can be easily located by the developers. The query achieves this is considered a high-quality query, as opposed to a low quality query, which either fails to even select the document that may contain the relevant data to the query. The output of the

Query Prediction component is a binary classification of two choices, high or low quality to help the developers to make the decision clearly whether pursue the query or reformulate it. The successful retrieval, leads to a result of a set of artifact that in the scenario are useful for solving the current Scenario Sifter task at hand by assuming the used query is the best possible approach.

The proposed query system is supported by using a prediction called Q2P (Mills et al., 2017) to predict the quality of the entered query. The main purpose of using Q2P is to give the developers a clear indication of whether the query that is used to describe the feature of interest is worth pursuing it or should be reformulated for a better performance. Hence, relying on the basic NL process is not enough to determine the quality of the query in retrieving the data, while the semantic relationship between words in natural language documents differ than in execution trace or source code.

Q2P is categorised into pre-retrieval properties and post-retrieval properties, depending on the moment when they are implemented. Pre-retrieval properties are captures before the query is executed, while post-retrieval properties are computed after query execution. The query predictor approach uses Machine Learning (ML) techniques and a set of 21 pre-retrieval techniques that are used to measure the query before executing it. In addition, 7 post-retrieval techniques that are used to measure the query after the execution. For example, during Scenario Sifter process, the developers can use the quality information as an indicator of the correctness of terms that are used to describe the feature of interest (in the case when the quality of the query is high) or reformulate the entered query (in the case when low quality is indicated). The developer can rely on highest quality of the query if multiple queries that are formulated marked as low quality.

The prediction of the quality of the textual queries in the context of the Scenario Sifter task, and making this information available to the developer

before his or her analysis of the collected execution trace begin. The entered query is considered a good, when the retrieval code elements is likely to contain code relevant to the feature of interest that describe by the entered query. Unless, the investigated part of the execution trace is not related to the targeted feature. The developers have an instance feedback about the quality of the formulated query. The Q2P system indicates whether the current formulated query is in high-quality or not in the retrieval document in the current execution trace artifact.

Retrieval Phase

IR techniques are required to be used in Scenario Sifter to select the relevant part of the executed sub-scenarios. Three IR techniques that are commonly used in FL are selected indicate whether the processing execution trace is relevant to the target feature or not. In addition to the use of the input text query that enhanced based on using Q2P system. These techniques are namely, Paragraph Vectors (DVs), Latent Dirichlet Allocation (LDA), and Latent Semantic Indexing (LSI). The accumulated average result of three IR techniques is calculated to compare it against the threshold value to decide whether the allocated execution trace T_n is opted to be related to the feature of interest or not. These techniques are described as follow:

LSI is a Natural Language Processing (NLP) technique as well as unsupervised learning technique. The aim of LSI is to create representation of the text data (execution trace) in term of topics or hidden feature. The approach obtains an underlying latent semantic structure from data composed by words. The main idea of the proposed LSI is to map each method or class into some concept, and map each term into some concept. The concept is defined as a set of terms, with weight. For instance, the input query that describes feature of interest, it uses the retrieval process first make use of the first matrix. The term concept matrix is used to locate the the right concept

that related to the input query. Then, the method concept matrix is used to assign a weight for the result of the first matrix.

LSI process is automated to retrieve the document beyond than exact match between the input query and the retrieved methods or classes. The idea is the text methods or classes can be presented as points in the Euclidean space. These points are called vectors. Table 1 illustrates an example of a Method-term Matrix. Each row in Table 4.2 represents one of the method by using method's identifier. Each column in Table 4.2 represents a term that appear in method's content, such as statements, variable, and etc. These terms should be appeared at least one in the methods as shown in rows of Table 4.2. For instance, (0, 1, 0, 1) is the value vector for the Method₁.

Table 4.2: Method-Term matrix based on LSI concept

	term ₁	term ₂	term ₃	term _n
Method ₁	0	1	0	1
Method ₂	1	1	0	1
Method ₃	0	0	0	0
Method _n	1	0	1	1

Moreover, LSI requires to use Singular Value Decomposition (SVD) on Method-term Matrix. The SVD reduces the Dimensionality of the original dataset by encoding it using hidden features. Hence, LSI represents the topic in the original text data.

The output of LSI process is a list of ranked methods or classes, assigned with vector values. The the average of the top highest values are selected for a further analysis in Processing Execution Trace component.

DVs is a ML process. It is known as Deep Learning (DL), that learns a feature representation for a given piece of text like sentences, paragraphs, and documents (Q. Le & Mikolov, 2014) to produce word embedding. Any

given term is associated with a numbers. For instance, the input query that is used to describe the feature of interest is replaced by array of numbers. Mathematical calculation is applied to find the similarity between the input query and the retrieved data. Euclidean distance is calculated to find the similarity between two terms.

In other words, DV solves the problem in finding the similarity between the input query and analyzed text. Each term is displayed as a unique vector represented by a size space dimensions. The size of the dimension is defined by a hyper-parameter K , and the learning process consists of a supervised task, where the set of methods or classes is used to construct these representations. Each vector is indexed by the position of the entire terms in the space. The concatenation of sum of vectors is subsequently used as a features for predicting for the next term in the sentence. The prediction task changes the word vectors and paragraph vectors.

In this case, the methods or classes are extracted from the execution trace to constructed DV. The content of the method's terms are tokenized and indexed to the method identifier. Each term, including method's identifier is represented by vectors that close to each other as shown in Figure 4.6. The input query locates the relevant methods based on the vector similarity between the query and the dimensional space content.

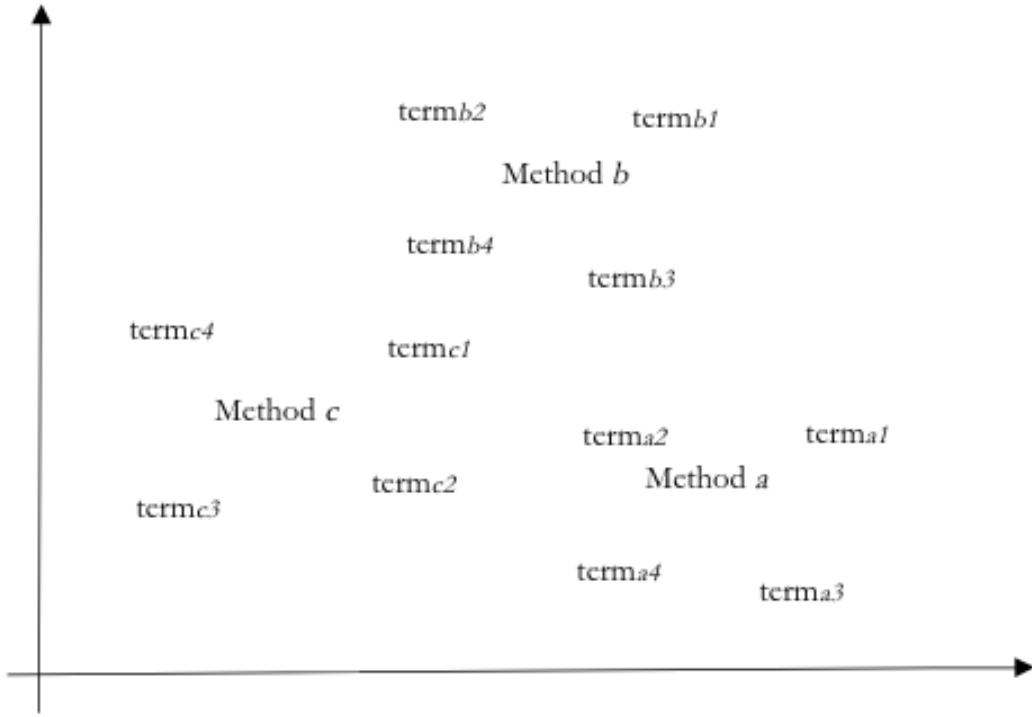


Figure 4.6: DV to show how the method is represented with terms.

The output of DV process is a list of ranked methods or classes, represented in decreasing order assigned with vector values (Corley, Damevski, & Kraft, 2015). The average of the highest values are selected for a further analysis in Processing Execution Trace component.

LDA is a topic model that generates a topic based on words frequency from a set of document. It is used probabilistic model for collections of discrete data such as text corpora (Blei, Ng, & Jordan, 2003). The proposed model represents a method or class as a probabilistic mixture of topics, where a topic is a distribution of words. Each method or class has a probability of belonging to each hidden topic, that built on the corpus phase. In order to locate the similarity between the methods generated by this model from the execution trace, it is important to compare the distribution of the topics of each one. The proposed metric enhances the calculation with the

divergence of probabilistic distributions and, therefore, a statistical comparison of which methods or classes are more similar to the probability of the topics that make up each method or class. The initial step is to implement the distribution of topics from each method or class that extracted from the execution trace. The construction of the topics is also performed in an unsupervised way, defined by a hyper-parameter K , where K is the number of topics in the model. Second, a vector representation for the location query. This representation is built based on the dictionary of terms obtained in the previous phase, where the dictionary of terms is a set of words or tokens extracted from documents. With this representation, we are able to construct a probability distribution of topics for the query, that allows comparison with other methods or classes. The JSD is used to construct an array of distance from the query, already in the topic distribution format, to each document. The lower the value found in relation to a method or class, the more similar it is to the query. The results are then sorted in ascending order, constituting the artifacts located by the technique.

The output of LDA is converted by using Gensim Library⁴ to assign probability score for each term in the input query. The retrieved method is ordered based on the number of terms that appear in the topic in descending order.

4.2.2.5 Selected Execution Trace

A comparative analysis involving three distinct techniques, namely LSI, DVs, and LDA, offers a comprehensive insight into the efficiency and effectiveness of these methods within the context of software engineering tasks, particularly in tasks such as feature location. By evaluating the output of each technique, which includes ranked methods or classes assigned with vector values and corresponding execution traces, developers can discern

⁴<https://radimrehurek.com/gensim/>

patterns, similarities, and discrepancies in the representation and retrieval of information, aiding in scenario-based feature location. The comparison of the average scores derived from these techniques not only aids in identifying the most suitable approach for a given task but also sheds light on each method in accurately locating software features based on specific scenarios. Such a comparative examination enables informed decision-making regarding the selection and optimization of text retrieval mechanisms, ultimately enhancing the precision and relevance of information retrieval processes in selecting the accurate scenario to perform DFLTs.

All the execution traces are filtered through the Threshold state based on the result of using list of IR techniques, such as LSI, DVs, and LDA. The threshold can be defined as the average of the highest values that are selected based on using the proposed IR techniques. The threshold is used to decide when the execution trace is considered to be part of the scenario to exhibit a feature. The related execution traces to the features are stored in the component called Selected Execution Traces for a mapping between execution trace and the related part of the scenarios.

4.2.2.6 Data Output in Scenario Sifter System

Figure 4.5 shows the main output of using Scenario Sifter to indicate the important part of the scenario based on filtering the irrelevant events. For instance, the Scenario n in Figure 4.5 is formulated based on four events, Event 1, Event 2, Event 3, and Event n . The use of Scenario Sifter optimises the events to two events, Event 2, and Event n , to exhibit the feature of interest.

4.2.3 Implementing and Execution Phase

In this section, the goal of the proposed methodology is presented by practical examples to show the effectiveness of using Scenarios Sifter in deleting irrelevant input of the scenario to increase the accuracy of the required scenario to exercise the feature of interest to preform DFLT. In addition, a comprehensive description is included to show how Scenario Sifter is evaluated by using two DFLT, metrics used, and methodology that is used for computing the evaluated metrics. Overall, the implementation aspect of Scenario Sifter is discussed in detail in this section.

The case study was implemented on a Fedora Linux workstation edition with a Quad-core Intel Core i5-8350U 1.70GHz processor and 8GB DDR4 RAM. The implementation consists of several subsections, and each subsection provides a full detail about proposed part in the implementation.

4.2.3.1 The case study targeted systems

The case study is implemented in three open-source Java software application that are widely used for FL implementation, analyses and evaluation: ArgoUML-SPL (Martinez et al., 2018), Notepad and Sudoku (Kim, Batory, & Dig, 2017). These datasets are commonly used to assess and test the performance of FLT in both direction static and dynamic approaches (Asadi, Antoniol, & Guéhéneuc, 2010; Michelon, Linsbauer, et al., 2021; Kästner, Dreiling, & Ostermann, 2013). Regarding in Chapter 2, research papers that used ArgoUML-SPL to generate execution trace for the system features, only used in this system as a benchmark dataset for evaluation. The ground-truth dataset for Notepad and Sudoku were generated manually. Moreover, the use of others two system are due to creating a more robust result that can be used for evaluation. Table 4.3 illustrates the number of lines of the application source code (LoC), the number of functional features (# of Features), programming language, and the software application's name.

In regard to the use of ArogUML-SPL system, a benchmark is obtained in advance (Martinez et al., 2018), which can be used as an appropriate dataset to evaluate and measure the effectiveness of using the Scenario Sifter. ArogUML-SPL has several features that are presented as diagrams, which creates the suitable environment to implement the proposed methodology where the features can be exercised manually by implementing a scenario through GUI. Nevertheless, the Scenario Sifter is suited to be used though the GUI for any Java software application. Any feature could not be exercised though GUI, it was not executed in the implemented case study. The application was executed as a JAR executable file though a command line:

```
java -jar ArogUML-SPL.java
```

Therefore, for ArgoUML-SPL, the considerable features were that beside the features that accepted to implement the scenario through the system's GUI. For instance, *Cognitive* and *Logging* features were excluded from the implemented case study due to their setup to run automatically in the background.

In order to evaluate the system with Notepad system, and Sudoku application, six features were selected for each of them to be exercised by six different scenarios respectively. The application was executed as a JAR executable file though a command line:

```
java -jar Notepad.jar
```

Therefore, the considerable features were that beside the features that accepted to implement the scenario through the system's GUI. For the Sudoku application, several features were exercised through navigating through GUI. The system was executed as a JAR executable file though a command line:

```
java -jar Sudoku.jar
```

Therefore, the considerable features were that beside the features that accepted to implement the scenario through the system’s GUI.

For the aforementioned target systems, all the implemented software artifacts and configuration that used for the evaluation are available publicly⁵.

Table 4.3: Application systems that are used to experiment Scenario Sifter

Application Name	Language	LoC	# of Feature
ArgoUML-SPL	Java	37494	8
Notepad	Java	1192	27
Sudoku	Java	1975	6

4.2.3.2 Formulate the input scenario and query

The case study systems all feature a GUI crucial for conducting scenarios. These scenarios, defined as a series of steps to demonstrate specific functional attributes within the instrumented software system for execution trace generation (Buzaid & Hammad, 2021a), are structured to emphasize the targeted functionalities. Sub-scenarios are created based on the number of mouse clicks required to complete the full scenario, exemplified by the exercise of the *Activity Diagram* in the ArgoUML-SPL tool, which necessitates a minimum of four mouse clicks. Each pair of clicks forms a distinct sub-scenario, serving as a subset of the overarching scenario. The initiation of a sub-scenario is signified by the first click, while the second click marks its conclusion and may also signify the commencement of the subsequent sub-scenario, emphasizing the sequential and precise nature of the scenario execution.

Three software systems were investigated to formulate six scenarios for six feature for every software system. A full information about creating scenarios was provided for ArgoUML-SPL, and for the briefing purpose,

⁵<https://github.com/buzaidf/Scenario-Sifter>

the reset of the others scenarios was provided on the evaluation on Chapter 5. ArogUML-SPL's GUI can be used to implement a set of scenarios, where each scenario is decomposed into a series of sub-scenarios. The developer implements the sub-scenarios of the proposed full scenario by navigating on the ArgoUML-SPL's GUI and use the mouse clicking to exercise the targeted feature. An execution trace is generated as a result of two mouse clicking in exercising the feature of interest. Table 4.4 shows examples of how scenarios are formulated to exhibit features in ArgoUML-SPL application, and allocate the relevant number of the generated execution traces. Each feature on the Table 4.4 requires at least the indicated number of mouse clicking to generate the relevant number of the execution traces. The unfamiliar user's with the system can formulate another seniors than what is illustrated on the table 4.4 which may require to implement more number of mouse clicking, and that generate a more number of the execution traces.

The input queries on the ArgoUML-SPL case study were formulated by the developer to describe the task that the developer was aiming to do. All the input queries were enhanced by Q2P system to support the developer to select the highest quality of the query in retrieving information from the created term's corpus.

Moreover, ArgoUML-SPL is written in Java programming language. Event handlers, and listeners can be used to manipulate scenarios to control the instrumentation technique based on the mouse clicking. A small program was coded⁶ to generate events. Once the event is generated, it transformed into event listener. The event listeners are interfaces to handle the events. The combination between event class, and the event interface, the event can handle a session to control the instrumentation technique (Memane, 2020). Listing 4.1 shows how to code the scenario to exhibit ArogUML-SPL's GUI application to control the instrumentation technique in order to generate the required execution traces. In this section, the goal of the proposed methodology is presented by practical examples to show the effectiveness of using Scenarios Sifter in deleting irrelevant input of the scenario to increase the accuracy of the required scenario to exercise the feature of interest to preform DFLT's. In addition, a comprehensive description is included to show how Scenario Sifter is evaluated by using two DFLT's, metrics used, and methodology that is used for computing the evaluated metrics. Overall, the implementation aspect of Scenario Sifter is discussed in detail in this section.

4.2.3.3 Instrumentation

An instrumentation technique is required to be used to profile all the invoked methods, classes and variables data during exhibiting the feature of interest. The instrumentation technique is controlled by the mouse clicking when the developer exercises the feature via the application's GUI.

The Equation 4.4 is used to quantify the relationship between mouse clicks and the generation of execution traces within a software application. By defining that one execution trace is produced for every two mouse clicks, this equation efficiently outlines the mapping between user inter-

⁶<https://github.com/buzaidf/Scenario Sifter>

actions (mouse clicks) and the system's response (execution traces). This calculation provides a straightforward means to estimate the number of execution traces that can be expected based on the frequency of user-initiated actions, aiding in the analysis and optimization of software behavior and performance.

Java bytecode toolkit (Javassist) (Horváth et al., 2019) offers an effective way to profile the invoked data during execution time by using a proxy to investigate the application at bytecode level. The Application Program Interface (API) of Javassist starts from the class pool, which acts as a class path to get all the resource from the class pool. CtClass is used to iterate and inspect on the internal structure, such as fields, methods, constant, and constructors. Listing 4.1 shows ClassPool cp is the representation of the class path. Then, CtClass ct is used to create the target presentation data on the profiled execution trace, such as indicating which method is required to be profiled by the instrumentation system.

```

1
2 public static void premain (String [] args)
3     throws Exception{
4     ClassPool cp = ClassPool.getDefault();
5     CtClass ct = cp.makeClass(
6         "com.zt.A,
7         cp.get("cp.zt.Clazz"));
8
9     CtMethod[] methods = ct.getMethods();
10    for (CtMethod method : methods) {
11        \\ add more instruction to fulfil
12        \\ what is required to include in
13        \\ the execution trace
14
15    }
16    ct.writeFile("/output");
17 }

```

Listing 4.1: Javassist instrumentation technique to profile data

For this case study, the software feature is instrumented by using java bytecode instrumentation (Horváth et al., 2019) for each scenario independently to apply OSIA. Selogger tool⁷ was used, and some modification were implemented to provide an execution trace for every running sub-scenario. The Selogger’s instrumentation technique is executed as a JAR (Java Agent) file to instrument the targeted software system which is implemented as an executable JAR file. The Java agent is the gateway to the class loading process where you can intercept the bytes being loaded and start manipulating them. The main class of the Java Agent includes premain method to provide the mechanism that is associated with the *java.lang.instrument* package, used for loading ”Agents” which make byte-code changes in Java programs. It works by specifying Java Agent arguments and run the tar-

⁷<https://github.com/takashi-ishio/selogger>

geted program. For instance, ArgoUML-SPL system was instrumented by the following command:

```
java -javaagent:Selogger.jar -jar ArgoUML
    -SPL.jar
```

All the required construction to profile the invoked data was implemented on *ClassFileTransformer*. It has many arguments, including *byte[] classfileBuffer* where all Java Agent API can be implemented as illustrated in Listing 4.2.

```
1
2 new ClassFileTransform(){
3     public byte[] transform
4         (ClassLoader loader,
5          String className,
6          Class<?> classBeingRedefined,
7          ProtectionDomain protectionDomain,
8          byte[] classfileBuffer){
9         ClassPool cp = ClassPool.getDefault();
10        CtClass ct = cp.makeClass(new
11            ByteArrayInputStream(classfileBuffer));
12
13        \\Here we can do all things to 'ct'
14
15        return ct.toBytecode();
16
17    }
18 }
```

Listing 4.2: Snapshot of ClassFileTransform to profile invoked methods

The output of the instrumentation technique is an execution trace for every running sub-scenarios. The implementation was based on OSIA to provide instant instrumented approach for the current exhibited feature or

even part of the targeted feature. The execution traces were formatted as text file, including all invoked methods, classes name, and instance variables.

4.2.3.4 Execution trace analysis

The use of Java Agent to instrument the targeted systems generates an execution trace for every running sub-scenario, corresponding to the current exercised feature or part of the feature in some cases. The generated execution trace was monitored based on OSIA to determine whether the collected execution trace is relevant to the feature of interest or not. The relevancy of the execution trace to the feature indicates how the corresponding sub-scenario is important to be included in the required scenario to exhibit the software feature. There are several techniques were used to analyse the execution trace and these implemented techniques were summarized as follow:

Term corpora was constructed to use input query, enhanced by Q2P system to retrieve the data and find whether the document (execution trace) is recommended to be included on the scenario. We extracted every methods from execution trace by using a tool called SELOGGER⁸. Helmholtz Principle (Dadachev, Balinsky, Balinsky, & Simske, 2012) was used to automatically extract the keywords for document processing. Using Helmholtz Principle was more efficient than using TF-IDF to extract the keywords from the execution trace. The execution trace was treated as a type of document. There was an execution trace for each running sub-scenario. The text of an inner method such as a method inside an anonymous class, it is included to that method, and not the containing one.

After extracting documents and tokenizing, we split the tokens based on camel case, underscores, and non-letters. We only keep the split tokens; original tokens are discarded. We normalize to lower case before filtering non-letters, English stop words, Java keywords, and words shorter than three

⁸<https://github.com/takashi-ishio/selogger>

characters long. We do not stem words. Here, we must be careful to not lose ordering of the words, as it can be used for a further stage in optimizing solution.

One aspect of working with execution trace requires to use textual IR techniques to investigate the number of documents and terms. Table 4.5 illustrated the the results of the implementation of three IR techniques, *i.e* LSI, LDA, and DVs as ML model. For each generated execution trace, that correspond to a certain sub-scenario, we assign the average value of using the three proposed IR techniques against the threshold value to indicate whether the processing execution trace is related to the feature based on the input query or not.

The threshold value of 0.1 utilized in the study was strategically chosen based on practical implementations across three case studies detailed in the chapter. The decision to opt for this threshold was primarily driven by the necessity to prevent zero values, which would not align with the practicalities of the case study scenarios (Zamani, Lee, Shokripour, & Anvik, 2014). This approach not only ensured the integrity of the data analysis process but also contributed to the overall robustness and applicability of the research findings within the context of the study.

The three implemented techniques, LSI, LDA, and DV return a vector value between (0-1) to measure the similarity between the input query and the execution trace content *e.g.*, methods, and classes. The vector value 1 indicates a complete similarity between the input query and the execution trace with value 1. A complete non-similarity between input query and the execution trace is indicated with value 0.

Table 4.5 presents the results of using Scenario Sifter in deleting irrelevant parts of the scenarios. The irrelevant parts were determined based on the decision of the average of three IR techniques, namely LSI, LDA, and DVs. The proposed retrieval system used text query that enhanced by a sys-

Table 4.5: Execution traces analysis for every sub-scenario by using IR techniques.

ArgoUML-SPL - IR Techniques								
Feature ID	Text Query Input (Q2P)	Sub-scenarios s_n	LSI	LDA	DVs	Average= μ	Threshold= τ	Result
F_1	Implement state diagram	s_1 : Click on ArgoUML application	N/A	N/A	N/A	N/A	N/A	N/A
		s_2 : Click on File	0.0021	0.0013	0.0018	0.0017	$\mu \geq \tau$	Rejected
		s_3 : Click on Open Project	0.0220	0.0187	0.0139	0.0182	$\mu \geq \tau$	Rejected
		s_4 : Select passportclassdiagram.zargo	0.0110	0.0069	0.0094	0.0091	$\mu \geq \tau$	Rejected
		s_5 : Click open	0.1310	0.0825	0.1113	0.1082	$\mu \geq \tau$	Selected
		s_6 : Select untitledMode	0.1440	0.0907	0.1224	0.1190	$\mu \geq \tau$	Selected
		s_7 : Click on Create Diagram	0.2310	0.1455	0.1963	0.1909	$\mu \geq \tau$	Selected
		s_8 : Click on New State Diagram	0.2311	0.1456	0.1964	0.1910	$\mu \geq \tau$	Selected
F_2	Implement activity diagram	s_1 : Click on ArgoUML application	N/A	N/A	N/A	N/A	N/A	N/A
		s_2 : Click on File	0.0101	0.0064	0.0086	0.0083	$\mu \geq \tau$	Rejected
		s_3 : Click on Open Project	0.0210	0.0132	0.0178	0.0083	$\mu \geq \tau$	Rejected
		s_4 : Select passportclassdiagram.zargo	0.1211	0.0763	0.1029	0.1001	$\mu \geq \tau$	Selected
		s_5 : Click open	0.1313	0.0827	0.1116	0.1001	$\mu \geq \tau$	Selected
		s_6 : Select untitledMode	0.1430	0.0901	0.1215	0.1182	$\mu \geq \tau$	Selected
		s_7 : Click on Create Diagram	0.2322	0.1463	0.1974	0.1919	$\mu \geq \tau$	Selected
		s_8 : Click on Activity Diagram	0.3310	0.2085	0.2814	0.2736	$\mu \geq \tau$	Selected
F_3	Implement use case diagram	s_1 : Click on ArgoUML application	N/A	N/A	N/A	N/A	N/A	N/A
		s_2 : Select on untitledModel	0.1431	0.0902	0.1216	0.1183	$\mu \geq \tau$	Selected
		s_3 : Click on Create	0.2310	0.1455	0.1963	0.1909	$\mu \geq \tau$	Selected
		s_4 : Select New Use Case Diagram	0.3200	0.2016	0.2720	0.2645	$\mu \geq \tau$	Selected
F_4	Implement collaboration diagram	s_1 : Click on ArgoUML application	N/A	N/A	N/A	N/A	N/A	N/A
		s_2 : Click on File	0.0120	0.0076	0.0102	0.0099	$\mu \geq \tau$	Rejected
		s_3 : Select passportclassdiagram.zargo	0.1011	0.0637	0.0859	0.0835	$\mu \geq \tau$	Rejected
		s_4 : Click Open	0.0120	0.0076	0.0102	0.0099	$\mu \geq \tau$	Rejected
		s_5 : Select passportclassdiagram.zargo	0.1011	0.0637	0.0859	0.0835	$\mu \geq \tau$	Rejected
		s_6 : Click Open	0.0120	0.0076	0.0102	0.0099	$\mu \geq \tau$	Selected
		s_7 : Select untitledMode	0.1341	0.0845	0.1140	0.1108	$\mu \geq \tau$	Selected
		s_8 : Click on Create diagrams	0.2200	0.1386	0.1870	0.1818	$\mu \geq \tau$	Selected
		s_9 : Select New Collaboration Diagrams	0.3420	0.2155	0.2907	0.2827	$\mu \geq \tau$	Selected
		s_{10} : Click File	0.0120	0.0076	0.0102	0.0099	$\mu \geq \tau$	Rejected
		s_{11} : Select Close	0.0011	0.0007	0.0009	0.0009	$\mu \geq \tau$	Rejected
		s_{12} : Click Cancel	0.0021	0.0013	0.0018	0.0017	$\mu \geq \tau$	Rejected
F_5	Implement deployment diagram	s_1 : Click on ArgoUML application	N/A	N/A	N/A	N/A	N/A	N/A
		s_2 : Click on File	0.0101	0.0064	0.0086	0.0083	$\mu \geq \tau$	Rejected
		s_3 : Select Diagrams	0.1443	0.1227	0.1443	0.1371	$\mu \geq \tau$	Selected
		s_4 : Click New Diagrams	0.2114	0.1797	0.1332	0.1747	$\mu \geq \tau$	Selected
		s_5 : Select Deployment Diagrams	0.3320	0.2092	0.2822	0.2744	$\mu \geq \tau$	Selected
F_6	Implement sequence diagram	s_1 : Click on ArgoUML application	N/A	N/A	N/A	N/A	N/A	N/A
		s_2 : Click on Create	0.2003	0.1262	0.1703	0.1656	$\mu \geq \tau$	Selected
		s_3 : Select New Sequence Diagrams	0.3200	0.2720	0.2016	0.2645	$\mu \geq \tau$	Selected

tem called Q2P to assist the developer to make a decision about the quality of the input query in retrieving the required data. However, the result indicated "Rejected" if the average results (μ) of the IR techniques was $< \tau$, which was equivalent to 0.1 at ArgoUML-SPL case study. Otherwise, the result indicated "Selected" if the $\mu \geq \tau$.

```
1
2 instrumentationMethod.addMouseListener
3     (new MouseAdapter() {{
4
5         public void mouseClicked
6             (MouseEvent e) {
7             if (e.getButton() ==
8                 MouseEvent.BUTTON) {
9                 getContentPane().add
10                    (instrumentationMethod);
11
12             } else (e.getButton() !=
13                 MouseEvent.BUTTON) {
14                 getContentPane().exit
15                    (instrumentationMethod);
16
17             }
18         }});
```

Listing 4.3: Handle mouse button click event to invoke instrumentation

4.3 The Use of UML Diagrams to Enhance Dynamic Feature Location Techniques

UML is an efficient technique that can be implemented to model and visualize an object-oriented system that suffers from lack or poor documentation. UML diagrams enhance the developers' understanding the software system

to apply software maintenance, software evaluation, or enhance the process of DFLT. The construction of UML diagrams is based on dynamic or static analysis. Dynamic analysis refers to collecting data when the system executes to generate execution traces, while static analysis refers to collecting the data from the source code. This experiment study implements UML diagrams in both directions, dynamic and static analysis, to enhance the software programmers applying AETFS. A use case scenario diagram is used to specify the inputs of the DFLT based on static analysis. In contrast, class diagrams and object diagrams are used to locate the required software artifacts of source code to construct the exercised feature based on dynamic analysis. Each created use case scenario generates an execution trace. The collected execution trace is analyzed to build class and object diagrams. The implemented case study illustrates the process of the proposed approach and shows that using UML diagrams increases the accuracy of applying AETFS rather than using random scenario input.

4.3.1 Overview

The third configuration of the AETFS is to use UML to specify the input of DFLT. The dynamic analysis becomes the preferable method for FL, despite the difficulties of creating the accurate input to exercise the feature of interest (Buzaid & Hammad, 2021b). Despite the progress made in DFLT (Koschke & Quante, 2005; Bittner et al., 2022), selecting the precise inputs to generate the required execution traces suffer from many difficulties such as the dependencies between software features, lack of the documentation of the software, and unavailability of the developers who are familiar with the system. In DFLT, understanding the input of DFLT and the effectiveness of the information in generating execution trace is the main point to be considered in the DFLT. Hence, it is a tedious and time-consuming task in DFLT to rely on the nondeterministic behavior of input by developers to

exercise the software feature to perform FL.

There are several advantages of visualizing the DFLT's input. UML diagrams are considered a proven and effective technique to ease the difficulties in understanding the behavior of the software system (Wimmer et al., 2011). UML diagrams supply a visualization solution to provide a view of the source code with the exhibited feature based on analyzing execution traces. A high-level abstraction model can be extracted from the execution trace to present the relation between the DFLT's input, demonstrated feature, and the relevant source code (Lau & Arshad, 2016). Moreover, UML diagrams are considered the preferable solution for software refactoring in the object-oriented system (Mumtaz, Alshayeb, Mahmood, & Niazi, 2019). Both approach dynamic and static models in UML are essential to comprehend the behavior of the software. Object diagrams and class diagrams are common diagrams in UML. They can be presented dynamically and statically to illustrate complex interactions between the software's behavior (Briand, Labiche, & Leduc, 2006). Hence, dynamic analysis is preferable to static analysis to allow the developers to remove the ambiguity of message sending when inheritance, delegation, polymorphism, dynamic links, and reflection are used intensively (Bennett et al., 2008). Therefore, the majority of the analysis in the proposed approach is based on dynamic analysis.

This experiment extends our previous work (Buzaid & Hammad, 2021a) to propose a new, more accurate, and precise approach for configuring AETFS that relies on using UML diagrams. The presented approach solution enhances the DFLT's input to select the required information to exercise the feature of interest by using a UML use case scenario diagram. Visualizing the DFLT's input by UML diagram improves the generating of the execution trace, which can be used in a later phase to construct object and class diagrams to locate the source code to the related exhibited feature. Indeed, using UML diagrams *i.e.*, a use case scenario diagram to translate the text

scenario into modeling language generates a smaller size and more efficient execution trace to perform DFLT's.

4.3.2 Design The experiment

The dynamic analysis depends on the software developers' input information to exercise the software's functionalities. It is a tangled task to locate all the possible inputs to run a particular feature of interest. The main objective of the proposed approach is to increase the accuracy of DFLT's input by using UML diagram *i.e.*, use case scenario diagram, to locate the required software feature's input and to minimize the generated execution trace. Hence, each use case scenario diagram dedicates to examining a software feature. Then, the object and class diagrams are constructed based on the collected execution trace.

In this section, the main steps of the proposed methodology are illustrated in Fig. 4.7. It is divided into four main steps. The first step is text scenario conversion. The second step is execution trace collection. The third step is execution trace filtering. The final step is diagram extraction. In the following subsections, each step is described in detail.

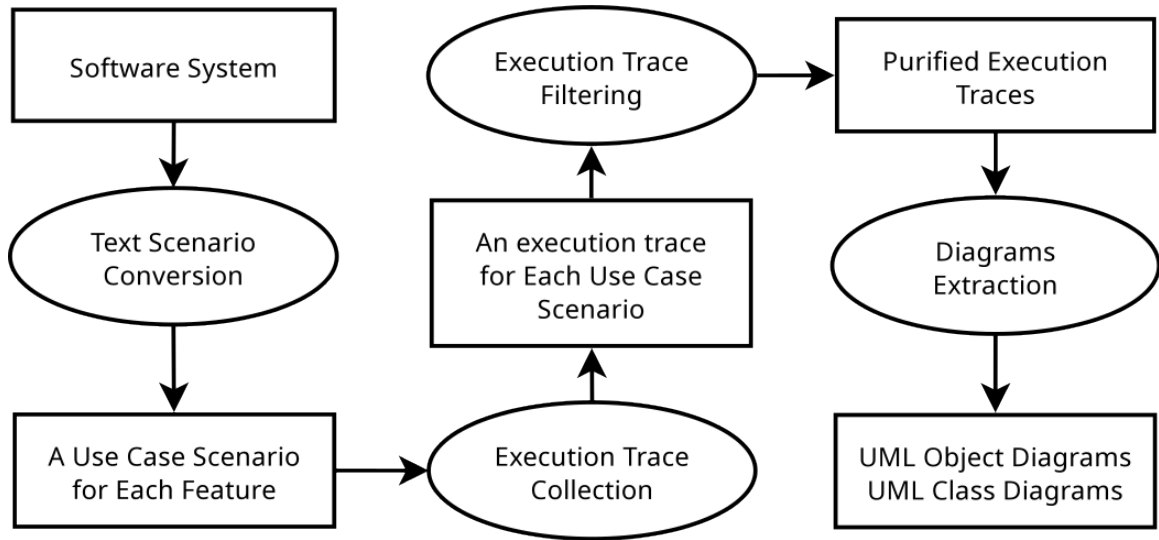


Figure 4.7: A brief methodology overview entails four key steps: text scenario conversion, execution trace collection, trace filtering, and diagram extraction.

4.3.2.1 Text Scenario Conversion

Usually, the scenarios to exercise the software's features are expressed by sentences written in natural language, which may be incomplete and inconsistent. In order to create an accurate DFLT's input, the proposed approach convert a text scenario into a use case scenario diagram to determine the required scenario to exercise the feature of interest. The proposed conversion mimics the published work in (Elallaoui, Nafil, & Touahni, 2018) to automate the transformation of the user's stories to exercise the feature into a UML use case diagram based on using NLP techniques (Copestake, 2004). The conversion includes three main phases, parsing text stories, plugin creation, and use case diagram extraction.

4.3.2.2 Parsing Text Stories

This point defines the required process to generate use case diagrams from users' text stories. Preprocessing a text file that contains a set of stories is considered to be the first step in implementing parsing text stories. There are

several techniques can be used to process a text file (Vijayarani, Ilamathi, Nithya, et al., 2015) such as using Context-Aware Stemming (CAS) algorithm (Agbele, Adesina, Azeez, & Abidoye, 2012) and stop word removal methods. Moreover, each term in the user's text stories is divided into a single Part Of Speech (POS). The classification of POS tags is classified into nouns (singular or plural), personal nouns, and proper nouns or verbs. Then, the next step is to parse the processed text file by using TreeTagger parser (Schmid, 1994). It produces a parse tree for each user's scenario that maps to a certain software feature. The generated parse tree supports the extraction process of actors (developers to exhibit software functionalities), using a use case diagram and the association relation between them. Hence, the other supportive technologies are need to accomplish parsing text stories are the Visual Paradigm tool⁹ and Java Programming Language.¹⁰

4.3.2.3 Plugin Creation

Software usability is a core element that needs to be considered to determine the success of software solutions. Java Plug-in extends the software's functionality to allow the Java Beans to be run under Sun's Java 2 runtime environment (JRE). The proposed solution dedicates to creating a java plugin to generate a use case automatically from text scenario stories. UML class diagram is used to model java plugin to generate use case diagram.

4.3.2.4 Use Case Diagram Extraction

UML provides a visualization technique to convert text data into a diagram. A set of text users' stories are extracted from the output of the subsection parsing text stories. Text mining algorithm (Kanya & Geetha, 2007) is used

⁹<https://www.visual-paradigm.com/>

¹⁰Java is a high-level, class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible.

to accept the input as a set of user stories. The input is stored in a text file to output a filtered text file that contains only nouns, compound nouns, and verbs. Algorithm 1 illustrates the required process to construct a UML file that includes the use case diagram.

Algorithm 1 Proposed Algorithm to construct UML use case diagram from the user text stories

Require: (*.txt) text file contains nouns, compound noun and verbs

Ensure: (*.uml) UML files contains UML Use Case diagram

```

1: Boolean : isFirst, isExist
2: isFirst  $\leftarrow$  True
3: isExist  $\leftarrow$  True
4: if (((POS==SN) or ((POS==PN) or (POS==PN))) and isFirst ) then
5:   for Actor x do
6:     if (isExist) then
7:       search x
8:       currentActor  $\leftarrow$  save
9:     else
10:      x  $\leftarrow$  createActor
11:    end if
12:  end for
13: end if
14: if ((POS==V) or (POS==VVP) or (POS==VPP) or (POS==VPD)) then
15:   verb  $\leftarrow$  add(token)
16: end if
17: if (((POS==PN) or ((POS==PN) and (POS==PN))) and (!isFirst)) then
18:   verb  $\leftarrow$  add(token)
19:   useCase u  $\leftarrow$  ComponentInitializer.generateUseCase (verb)
20:   Association assoc  $\leftarrow$  ComponentInitializer .generateAssociation (currentActor, u)
21: end if

```

The presented Algorithm 1 is designed to abstractly generate UML use case diagrams from user text narratives, which depict scenarios exercising software functionalities within the context of DFLT's procedures. Operating

on a text file containing nouns, compound nouns, and verbs, the algorithm outputs a UML file featuring a UML Use Case diagram by employing POS tagging for text analysis and information extraction.

Initialization of two Boolean variables, `isFirst` and `isExist`, both initially set to `True`, serves to monitor the initial appearance of actors in the textual narrative. Iterating through each word in the text, the algorithm evaluates their POS tags through conditional assessments.

Detection of a Singular Noun (SN), Plural Noun (PN), or Proper Noun (PN) as the POS, coupled with a first occurrence indication (`isFirst` is `True`), triggers a loop to process each mentioned actor. Verification of actor existence in the use case diagram, governed by the `isExist` flag, leads to the preservation of the actor as `currentActor` if present, or the creation of a new actor based on the extracted name.

Identification of a Verb (V), Past Participle Verb (VVP), or Verb with a Direct Object (VPD) as the POS prompts the inclusion of the verb in a designated list or variable named `verb`, capturing the actions associated with the actors in the narrative.

Further recognition of a Proper Noun (PN) following the first occurrence (`isFirst` is `False`) signifies the incorporation of the current noun into a use case context. The algorithm integrates the noun into the verb list, invokes the `ComponentInitializer.generateUseCase` function for use case generation, and establishes an association between the `currentActor` and the use case using the `ComponentInitializer.generateAssociation` function.

Through POS tagging and systematic management of actors, verbs, and use cases, the algorithm constructs UML Use Case diagrams that faithfully represent user-generated narratives. Nevertheless, it is essential to highlight that the provided algorithm excerpt lacks certain specifics, such as variable and function definitions, crucial for a comprehensive grasp of the algorithm's overarching purpose and operational intricacies in an abstract

context.

4.3.2.5 Execution Trace Collection

An execution trace is defined as a finite sequence of events that are invoked in a software system at runtime. The events can be any level of the software granularity, such as classes, methods, objects, or statements. The proposed solution uses dynamic analysis to collect execution traces when the software feature is exercised by the created use case diagram. The UML class diagrams and object diagrams are constructed based on the collected execution traces. There are several existing techniques that can be used to generate and collect the execution traces during software's runtime (Cornelissen et al., 2009). For instance, AspectJ language¹¹ monitors the program at a certain point in the program. The monitoring uses two techniques called *joinpoints* and *pointcuts* (Kiczales et al., 2001). Joinpoints contain an abstraction of the current call stack, while pointcuts indicate the current program state. The developers use pointcuts and joinpoints to define advice pieces of code to trace the program at specified criteria. The monitored program by AspectJ relies on the standard Aspect-Oriented Program (AOP)¹² to model a sequence of entries and exits to profile the execution trace. Listing 4.4 illustrates the standard of AspectJ source code to generate an execution trace.

AspectJ, as described in the above paragraph, plays a crucial role in bridging AOP with OOP. In the context provided, AspectJ is utilized to monitor programs at specific points, employing joinpoints and pointcuts as key techniques.

1. **Joinpoints:** Joinpoints abstract the current call stack, essentially capturing moments during the program's execution. These points are pivotal in the AspectJ framework for identifying where advice should be

¹¹<https://www.eclipse.org/aspectj/>

¹²<https://docs.spring.io/spring-framework/docs/2.5.x/reference/aop.html>

applied.

2. **Pointcuts:** Pointcuts signify the current program state, helping developers define specific criteria for tracing the program. By utilizing pointcuts and joinpoints, developers can encapsulate advice code that captures program behavior under certain conditions.
3. **Aspect-Oriented Programming (AOP):** AspectJ leverages AOP principles to create a structured sequence of program entries and exits for profiling execution traces. This integration allows for the modeling of intricate behavior without scattering the core logic across the codebase.

By combining these elements, AspectJ enhances the capabilities of OOP by providing a means to encapsulate crosscutting concerns and behavior, such as monitoring, logging, and performance optimizations, without intruding on the primary object-oriented design. This seamless integration of AOP with OOP principles enables developers to maintain a clean and modularized codebase while addressing concerns that would otherwise lead to code tangling in traditional OOP paradigms.

```
1 aspect ProgramTrace {  
2     before() : !within(ProgramTrace) {  
3         System.out.println("Entering: " + thisJoinPoint);  
4     }  
5     after() : !within(ProgramTrace) {  
6         System.out.println("Exiting: " + thisJoinPoint);  
7     }  
8 }
```

Listing 4.4: AspectJ code snippet for tracing program execution (Allan et al., 2005)

An entry event occurs before every joinpoint, and the exit point includes after every entry point. Moreover, the pointcuts allow the developers to assert the executed program a given set of joinpoints in the execution stack.

4.3.2.6 Execution Trace Filtering

Dynamic analysis encounters challenges when dealing with extensive execution traces that include irrelevant data, often referred to as noise. These traces must undergo filtering to isolate specific events of interest, a process that can be facilitated using techniques like regular expressions (Chapman & Stolee, 2016). In the context of utilizing AspectJ for trace generation, the tracematches technique (Allan et al., 2005) is employed for filtering the gathered execution trace. This selected filtering method empowers developers to define a regular pattern for tailoring the execution trace according to the requirements of a new application class. The tracematch technique operates through two core functions: filtering and binding variables consistently throughout the matching process, thereby pinpointing and tracking the precise behavior necessitating tracing. By employing regular expressions as filtering patterns, tracematches are structured into three fundamental components: defining relevant events, constructing the involved pattern, and pinpointing the code segment to execute. A match is triggered when a segment of the ongoing analyzed execution traces aligns with the specified regular expression pattern. Listing 4.5 illustrates a straightforward implementation of the tracematch filtering technique.

```
1
2 tracematch(){
3     sym ie before: call(* ie(..));
4     sym te after: call(* te(..));
5
6     ie te
7     { System.out.println("ietete!");}
8 }
```

Listing 4.5: A snapshot of tracematch technique to filter an execution trace

The tracematch method defines any tracematch regular expression. Next, two symbols are defined *ie* and *te*. The first symbol *ie* refers to matches en-

ter events on joinpoints which match the pointcut $call(*ie(..))$. The second symbol te refers to match exit events on joinpoints that match the pointcut $call(*te(..))$. While the regular expression $iete$ specifies that advice is invoked on the execution trace which end with a call to ie and te . Finally, $System.out.println("iete!")$ gives the advice body to be executed.

Algorithm 2 describes the required steps to construct an object diagram.

Algorithm 2

Require: (*.txt) an execution trace events

Ensure: (*.uml) UML object diagram $D_{n \times n}$

- 1: Initialize Graph to $D_{1 \times 1}$ with a single state BEGIN;
 - 2: Initialize $k = 1$;
 - 3: currentState = BEGIN;
 - 4: nextState = nextTransition = NULL;
 - 5: **repeat**
 - 6: Read the next line L_i from the execution trace events;
 - 7: nextState = getState(L_i);
 - 8: nextTransition = getTransition(L_i);
 - 9: **if** nextState $\notin D$ **then**
 - 10: $D_{(k+1) \times (k+1)} = D_{k \times k} \cup \{nextState\}$;
 - 11: $D [currentState, nextState] = 1$;
 - 12: Increment k ;
 - 13: **end if**
 - 14: **if** nextTransition $\notin D$ **then**
 - 15: $D [currentState, nextState] = 1$;
 - 16: **end if**
 - 17: currentState = nextState;
 - 18: **until** end of execution trace events
-

Algorithm 2, it is a novel algorithm designed to construct UML object diagrams from an execution trace. The algorithm serves as a crucial step in DFLT's formulating input, enabling the generation of UML object diagrams that capture the interactions and behavior of software systems during exe-

cution. The algorithm takes an execution trace, obtained from a previously generated UML use case diagram (Algorithm 1), and transforms it into a UML object diagram, providing valuable insights into the system's runtime behavior.

The proposed Algorithm 2, titled "Proposed Algorithm to construct UML object diagram from a single execution trace," aims to construct a UML object diagram from an execution trace file (*.txt). The algorithm's goal is to capture the dynamic behavior of the software system by representing the objects and their interactions during the execution trace. The resulting UML object diagram provides a visual representation of the system's runtime behavior.

The algorithm initializes a graph, denoted as D , with a single state labeled *BEGIN*. This state represents the initial state of the UML object diagram. Additionally, the algorithm initializes the variable k to 1, indicating the current size of the object diagram. It sets *currentState* to *BEGIN* and assigns *NULL* values to *nextState* and *nextTransition*.

The algorithm enters a repeat loop to process each line of the execution trace events. It reads the next line, L_i , from the execution trace events file and retrieves the *nextState* and *nextTransition* using the *getState* L_i and *getTransition* L_i functions, respectively.

If the *nextState* is not present in the current object diagram D , the algorithm expands the object diagram by creating a new graph, $D(k+1) * (k+1)$, and adds the *nextState* to it. This step allows for the representation of new states encountered during the execution trace.

The algorithm records the transition from the *currentState* to the *nextState* by marking it as 1 in the object diagram D . This step captures the dynamic relationships and interactions between objects during the execution trace.

After expanding the object diagram or recording a transition, the algorithm increments the variable k to reflect the updated size of the object dia-

gram.

The algorithm updates *currentState* to *nextState*, preparing for the next iteration. The loop continues until the end of the execution trace events is reached.

The output of Algorithm 2 is a UML object diagram, denoted as $D_{(n*n)}$, where n represents the final size of the object diagram after processing the entire execution trace. The object diagram provides a visual representation of the system's runtime behavior, capturing the objects, states, and transitions encountered during the execution trace. This UML object diagram serves as a valuable artifact in software testing and debugging, enabling analysts to gain insights into the system's dynamic behavior and identify potential issues or anomalies.

Algorithm 2 is an integral part of the overall methodology proposed in this thesis. It complements Algorithm 1, which constructs the initial UML use case diagram from user text stories. The output of Algorithm 1, the UML use case diagram, is utilized to exercise software features, resulting in an execution trace. Algorithm 2 then takes this execution trace as input and constructs the UML object diagram, providing a comprehensive representation of the system's runtime behavior. The seamless integration of Algorithms 1 and 2 enables a holistic approach to DFLT's process, facilitating the identification of DFLT's input and behavioral aspects of AETFS configuration.

In summary, Algorithm 2 plays a vital role in the proposed methodology by generating UML object diagrams from execution traces. These diagrams capture the dynamic behavior of the software system, providing valuable insights for formulating the DFLT's input. The algorithm's steps encompass initialization, execution trace processing, object diagram expansion, transition recording, size updating, and loop continuation. The resulting UML object diagram serves as a significant artifact to understand the system's

runtime behavior and complement the UML use case diagrams generated by Algorithm 1.

Chapter 5

Assessment and Evaluation of the AETFS Methodology

The preceding chapters introduced the evaluation design of the experiment, the research gap, the innovative AETFS methodology, and its three empirical configurations used for evaluation. This chapter begins by the evaluation design and metric that uses for the assessment of AETFS experiments. In addition detailing the outcomes of a preliminary pilot study. Insights gained from this study informed the subsequent expanded experiment, the findings of which are also discussed herein. Specifically, this chapter delineates the research inquiries addressed by the larger study, presents its findings, examines these findings, considers potential validity threats, and reviews the data on effort expended.

5.1 The Initial Investigation

This segment outlines the outcomes of the pilot study, which serves as an initial appraisal of the AETFS framework. The study encompasses a limited set of subject systems and addresses research questions specific to the pilot study. It offers a preliminary, albeit imprecise, contrast between the results obtained with the DFLTS baseline when incorporating AETFS and when it is not incorporated.

5.1.1 The Research Questions of the Initial Investigation

The primary aim of the pilot study was to evaluate the impact of AETFS on enhancing the precision of DFLT's usage. AETFS aims to solve the issue of choosing or crafting the correct scenario inputs to demonstrate the software feature for DFLT's execution. Traditional DFLT's typically utilize a random scenario to activate the software feature (Cornelissen et al., 2009; Rubin & Chechik, 2013; Razzaq et al., 2020), followed by a process that annotates the source code entities in the execution trace with the corresponding feature. Therefore, it was crucial to determine AETFS's efficacy compared to these conventional methods and to ascertain its feasibility before embarking on a more extensive study. The initial research questions focuses on AETFS's effectiveness and can be as follow:

RQ1: How do the effectiveness metrics, such as the number of accurately exercised software features and the size of collected execution traces, assess the formulated scenarios or test cases' ability to exercise specific software features in AETFS?

There's a notable discrepancy among TDFLT's providers regarding the preferred granularity of source code for their solutions, with a tendency to focus on more detailed results, typically at the method level (Biggers et al., 2014; Moslehi, Adams, & Rilling, 2020). However, the rationale behind

choosing a particular level of granularity is seldom explained by the authors of TDFLTs, and there's a lack of concrete evidence to support the preference for finer granularity. Adding to this complexity is the issue of scenario test inputs, which can significantly influence the granularity needed. Programmers may need to locate features at the file level or the method level, depending on the scenario test input. This raises a critical question: Is it more efficient to present files or methods to programmers? The answer hinges on the amount of effort required to locate features across the two common targets—methods and files—considering the scenario test inputs involved.

RQ2: How does the accuracy of formulated scenarios or test cases, as measured by the number of relevant methods/classes invoked and the presence of irrelevant retrieved data in collected execution traces, impact the performance of AETFS?

The AETFS configuration is part of DFLTs and remains unexplored in FL studies. However, insights can be gleaned from (Ziftci & Krüger, 2012) and (Arwan, Rochimah, & Fatichah, 2023) research. (Ziftci & Krüger, 2012) approach applies association rule learning to labeled test-cases for feature-to-code mapping, bypassing manual scenarios and rivaling probabilistic methods. Conversely, (Arwan et al., 2023) work enhances feature location by generating query expansions that discern use case scenario connections, employing LDA to categorize source code topics. While these methodologies are distinct, they both aim to refine feature location processes—the former through data mining of test-cases, and the latter via innovative use case scenario linkages for more precise source code identification. Their differences pose an intriguing inquiry into how execution trace content influences AETFS efficacy. This leads us to pose the third question:

RQ3: How can the effectiveness of utilizing execution traces in the feature location process be measured, considering metrics such as the size of

collected execution traces in comparison to the dataset and the proportion of the located foothold to the near-full-set feature in the execution trace?

The introduction of the AETFS framework in response to the challenges faced by DFLT's signifies a significant leap towards enhancing the effectiveness of feature identification within software systems. AETFS's innovative approach leverages runtime software output and advanced textual analysis methodologies to extract relevant data from execution traces, enabling the creation of precise scenarios for DFLT's. Through the exploration of textual analysis techniques like topic modeling and LSI, AETFS not only refines the granularity of execution traces but also identifies crucial terms to streamline the trace analysis process. The evaluation, conducted across a diverse set of eight subject systems encompassing 600 features, surpasses the depth of previous studies, showcasing the framework's robustness. The study's findings underscore AETFS's remarkable performance, evident in the emergence of distinct groups achieving superior FL results compared to conventional methods.

5.1.2 Preliminary Findings and Examination

Through a comprehensive analysis of my PhD experiment case studies, encompassing investigations such as "Evaluation of the Mapping Features to Source Code Based on Execution Trace," "Scenario Sifter to Filter Irrelevant Inputs," and "The Use of UML Diagrams to Enhance DFLT's," a significant finding emerged in response to the research question. The study indicated that method-level granularity is more effective for AETFS compared to file-level granularity, crucial for identifying entities essential for scenario accuracy in subsequent stages. This discovery aligns with previous research on granularity's impact on input scenarios within DFLT's, emphasizing the optimal use of method-level granularity for both DFLT's and AETFS. Moreover, an intriguing outcome of the study highlighted that incorporating any

form of instrumentation technique, particularly the bytecode instrumentation technique (JVM Java Agent), detrimentally impacts AETFS efficiency at the method level granularity, suggesting a potential reduction in the size of collected execution traces with the use of appropriate instrumentation techniques.

This section presents the findings of the empirical study, analyzing them to address the initial research questions outlined in Section 5.1.1. Table 5.1 illustrates the metrics of three distinct configurations of AETFS for each subject system. Delving into the metrics of Precision (P), Recall (R), and F-Measure, our objective is to replicate the fundamental processes underlying these measurements. To further clarify this concept, our methodology for replicating these metrics is as follows: True Positive (*TP*) refers to the number of methods that are retrieved when a software feature is exercised by a scenarios. Those retrieved methods are part of the dataset for the exercised software feature. Conversely, False Positive (*FP*) refers to the number of methods that are retrieved when the software feature is exercised by the scenario, but those methods are not part of the software feature's dataset. While, False Negative (*FN*) indicates to the number of methods that are retrieved when the software feature is exercised by the scenario, but those methods are not part of the all software system's dataset.

Table 5.1: Evaluation Metrics for the Configuration of AETFS in Applying DFLTS

Metric	Description	Calculation Formula
Precision (P)	The proportion of correctly identified methods out of all the identified methods in the dataset. It measures the accuracy of the feature location process by quantifying the ratio of true positives to the sum of true positives and false positives. A higher precision value indicates a lower number of false positives.	$\frac{TP}{TP+FP}$
Recall (R)	The proportion of correctly identified methods out of all the actual methods present in the system. It measures the completeness of the feature location process by quantifying the ratio of true positives to the sum of true positives and false negatives. A higher recall value indicates a lower number of false negatives.	$\frac{TP}{TP+FN}$
F-Measure	The harmonic mean of precision and recall, providing a balanced assessment of the feature location process. It combines precision and recall into a single metric, allowing for a comprehensive evaluation of the effectiveness of feature location techniques. A higher F-measure value indicates a better balance between precision and recall.	$\frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$

The dataset outlined in Table 5.2 presents a comprehensive overview of subject systems utilized for the pilot study evaluation (Dit, Holtzhauer, Poshyvanyk, & Kagdi, 2013). Among these systems, ArgoUML0.22, ArgoUML0.24, and ArgoUML0.26.2 from the ArgoUML project exhibit varying characteristics in terms of source code URL, period of evaluation, types of issues (including defects, enhancements, features, patches, and tasks), and the number of gold set methods. For instance, ArgoUML0.22 covers

a period from 0.20 to 0.22, encompassing 74 defects, 10 enhancements, 2 features, and 5 patches, with a total of 701 gold set methods. This detailed dataset offers a valuable resource for analyzing and understanding the software features and associated maintenance tasks within each subject system, providing a foundation for conducting rigorous evaluations and investigations in the field of software maintenance and evolution.

Table 5.2: Details of the subject systems that are used for the pilot study evaluation (Dit, Holtzhauer, et al., 2013)

Dataset	Source Code URL	Period	Issues	Gold Set Methods
ArgoUML0.22	[ArgoUML]	0.20-0.22	74 Defects, 10 Enhancements, 2 Features, 5 Patches (91 Total)	701
ArgoUML0.24	[ArgoUML]	0.22-0.24	32 Defects, 4 Enhancements, 15 Patches, 1 Task (52 Total)	357
ArgoUML0.26.2	[ArgoUML]	0.24-0.26.2	181 Defects, 19 Enhancements, 2 Features, 4 Patches, 3 Task (209 Total)	1,560
JabRef2.6	[JabRef]	2.0-2.6	36 Defects, 3 Features (39 Total)	280
jEdit4.3	[jEdit]	4.2-4.3	86 Bugs, 34 Features, 30 Patches (150 Total)	748
muCommander0.8.5	[muCommander]	0.8.0-0.8.5	81 Defects, 11 Enhancements (92 Total)	717

The table 5.2 provides the following details for each subject system:

- **Dataset:** The name of the dataset.
- **Source Code:** The source code URL corresponding to the subject system.
- **Period:** The time period during which the data was collected.

- **Issues:** The number of issues (defects, enhancements, features, patches, etc.) associated with the subject system. The URL to the issue tracking system is provided for further reference.
- **Number of Gold Set Methods:** The count of gold set methods available for the subject system.

Table 5.2 details data of existing baseline FL based DFLT's from the literature against AETFS. For each configuration the effort, as based on the adjustment scenarios discussed in Sections 4.1 and 4.3, is calculated and compiled in Table 5.3.

Table 5.3: Comparison of P, R, and F-Measure with and without AETFS

Dataset	Without AETFS			With AETFS		
	P	R	F-Measure	P	R	F-Measure
ArgoUML0.22	0.85	0.92	0.88	0.95	1	0.97
ArgoUML0.24	0.79	0.86	0.82	0.88	0.96	0.91
ArgoUML0.26.2	0.88	0.91	0.89	0.98	1	0.98
JabRef2.6	0.68	0.72	0.70	0.76	0.80	0.78
jEdit4.3	0.79	0.83	0.81	0.88	0.92	0.90
muCommander0.8.5	0.89	0.92	0.90	0.99	1	0.99

Table 5.3 presents the average results of P, R, and F-Measure for six software systems. Each of these software systems can exhibit various issues through scenarios. For example, in Table 5.2, the dataset ArgoUML0.22 is shown to encompass a total of 91 issues, with 701 methods designated as Gold Set Methods for these issues. Among these issues, the "Class Diagram Generation" stands out as one of the two features in the dataset, requiring 349 methods for its construction. When a scenario exercises the Class Diagram Generation feature without utilizing AETFS, the outcomes are as follows: 349 methods are retrieved and are part of the gold dataset TP , whereas 61 methods are retrieved during scenario execution but are not part of the

feature's dataset , FP . Additionally, 29 methods are retrieved during feature exercise by the scenario but are not included in any software system's dataset , FN out of the total 701 methods in the dataset.

Consequently, the rest of the issues (Features) for all the dataset are applied the proposed P, R, and F-measure in Table 5.1 to get the TP , FP , and FN for each issue by using AETFS, and without using AETFS. The results in Table 5.3 represents the average of P, R and F-Measure for the all issue feature in dataset. For instance, the average results of P, R, and F-Measure for 91 issue (features) in ArgoUML0.22 dataset are 0.85, 0.92 and 0.88, respectively without using AETFS. While the results are changed to be 0.95, 1, and 0.97, respectively when using AETFS.

Moreover, choosing a correct dynamic techniques to instrument data in the right granularity level in the source code, as well as choosing the correct static technique to extract the relevant source code that is related to the feature, results in 12% is more effective feature location technique than without using AETFS.

5.2 Assessment and Evaluation of the Mapping features to source code based on execution trace

In order to perform an evaluation of the proposed experiment, a case study has been created to experiment the proposed methodology. We use an open source ArgoUML (ArogUML, 2020) as a targeted system to be analysed in this case study. It is considered as a large size of real software application with 120 KLOC, that fully implemented by Java programming language. It consists of 14644 methods to implement 23 features. Some of the features are activated only when another feature is activated. For example, a use

case diagram is activated only when Diagram feature is activated as shown in Figure 5.1. The arrows in this figure indicate the desired features that are exhibited by the predefined scenarios. In this case study, we use two features of ArgoUML, sequence diagram, and activity diagram. A set of predefined scenarios are used to exhibit these selected two features as described in Table 5.4. This table presents a benchmark data (Martinez et al., 2018) for the ArgoUML to show the number of methods (M) that is required to be changed when a feature gets modified. In addition, it shows the number line of code (LOC) are used to construct each feature, and the steps of events (Scenario) that are required to exhibit a feature. The result of running these scenarios is to get an execution trace that is filtered based on using reformulating query technology to extract relevant methods to the feature. The extracted methods are filtered again based on the distance between the ranked methods, to output a list of methods that are mapped to the feature with a high level of accuracy. We compare the obtained result with the other works for a validity purpose in our case study.

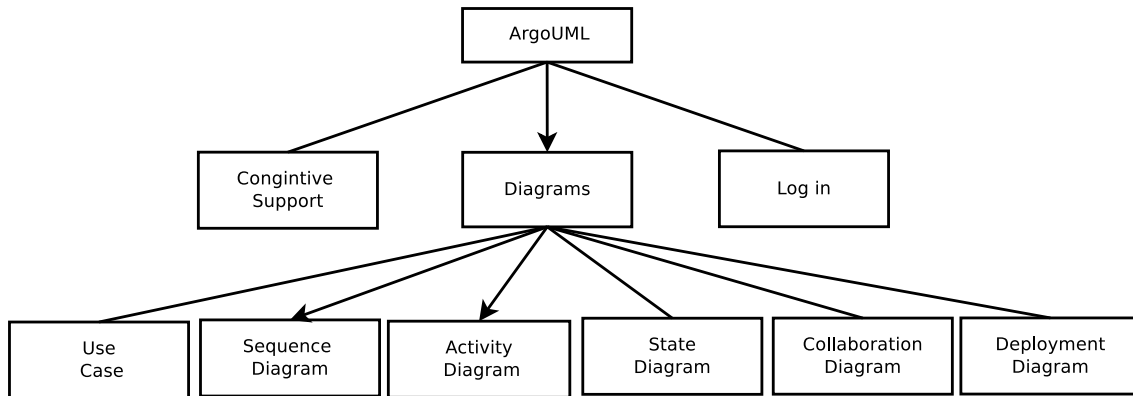


Figure 5.1: Scenarios to exhibit two features in ArgoUML.

The first required step is to instrument the target system ArogUML to monitor the application during a runtime. We use srcML tool to insert a hook to monitor the source code of ArgoUML to profile all set of methods including the methods content that are invoked when the scenario is exe-

Table 5.4: Feature, Scenarios, LOC, and Number of Methods (M)

Feature	Scenario	LOC	M
Sequence Diagram	The developer lanuch , the ArogUML application select Diagram, and choose the sequence diagram	5379	379
Activity Diagram	The developer lanuch , the ArogUML application select Diagram, and choose the activity diagram	2282	288

cuted. Figure 4.2, shows the main process that are required to instrument the ArogUML source code to log all methods including the content form the methods initialization to return method declaration, that are invoked when executed a scenario. First, the srcML converts the ArogUML java source code input to XML format. In this case study, we opted to convert the entire ArgoUML from an online resource to xml format by using the following command:

src2srcml -register-ext h=Java https://github.com/marcusvnace/argouml-spl.git -o argo.xml. The result of using srcML conversion is to get an xml file size 41MB while the original directory source code size is 69MB. Then, we formulated a basic XSLT file called *profiler.xsl*, to insert a predefined function *MonitFunc* by using Xpath, into ArogUML in the converted *argo.xml* files. The following command is used to merge xslt file with xml file to generate instrumented file: *srcml2src argo.xml — srcml2src -xslt profiler.xsl — srcml2src-to-dir instrumentedXMLfile*. Figure 5.3, shows the steps of merging XML file with XSLT profiler to output instrumented ArogUML source code in XML format, *i.e*, *instrumentedArogUML.xml*. Then, srcML

tool converts back the instrumented ArgoUML in XML format to the original java source code format with instrumentation code in java format as shown in Figure 5.2 by using the following command: *src2srcml instrumentedArogUML.xml -o argoV1.java*. The *MonitFunc* is inserted in every methods in *argov1.java* source code at two points, after method initialization and before method return.

```

Public class Example
{
    Public static method (){

        // Predefined function to hook
        // source code where to start

        MonitFunc("method", "Start")

        // Method code ...
        // Method code ...
        // Predefined function to hook
        // source code where to End

        MonitFunc("method", "End")

        Return ;}
}

```

Figure 5.2: Java source code after instrumentation.

The result of this instrumentation is to generate an execution trace that contains in all methods with the content such as methods' statement that are invoked during exhibiting a sequence diagram feature. For instance, when we run a scenario to exercise a sequence diagram feature as shown in Table 5.4, we have all invoked methods profiled in file called *loggedV1*, including relevant methods to the feature such as *org.argouml.uml.ui.SequenceDiagram()*,

and utilize methods such as *static void main ()*. Moreover, all the information between the method declaration and before method return are profiled with the relevant method.

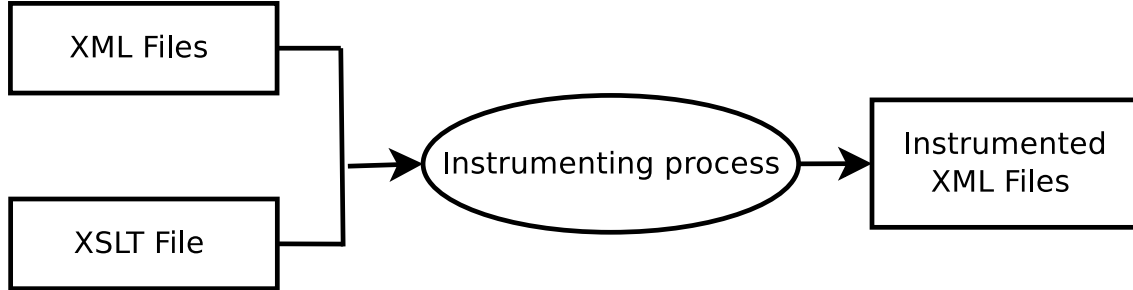


Figure 5.3: Process to instrument XML files.

The collected trace data from the instrumented ArgoUML for a sequence diagram feature is a huge file as shown. It contains 1000 methods including the methods content between the method's initialization and method's return. Filtering the execution trace is required to extract only relevant methods to the features. We have to create words corpus to index the rank source code methods by using Latent Semantic Indexing (LSI) (Rafiei & Mendelson, 1997). For each method in the loggedV1, there are a documented created for this method. This document indexes all terms that method is included. For instance, for this case study example, a document for a method called *SequenceDiagram()* is created, and it indexes to the all terms that methods are included. Apache Lucene (Foundation, 2011) was used to create a text search engine based on (LSI) to find string matching between the entered query and the created corpus. It uses a *Directory* as a collection of documents that contains all terms. We use *IndexWriter* as a writer to add a document into the corpus. *IndexSearcher* is used to execute a query to search on the document and indexes the result on *TopDocs*.

Finding the required methods from the created corpus needs to enter a query that describe a feature in a high level. It is used a Refoqus technique (Haiduc et al., 2013) to reformulate a query to have a better retrieving

data from the corpus. This technique automatically provides a suggestion to either expand or reduce the entered query to have a better retrieval information. For example, if we presume the full query that has to be formulated to retrieve a sequence diagram method is a query with two words, *sequence diagram*. When the developer writes a query *sequence*, the quality of the retrieving methods are effected. Therefore, we propose to use Refoqus tool to provide auto suggestion to add or eliminate terms. For example, adding *diagram* for a better retrieval performance. Table 5.5 shows the result of locating methods for by the proposed approach. The total number of methods (No-M) that are profiled when executed a sequence diagram scenario is 1000, included relevant and irrelevant methods to the feature. The benchmark methods (M-gold-set) that are required to be changed when modifying sequence diagram is 379. When we use Refoqus tool to reformulate query to extract the relevant methods, we get 450 method, and that indicated in Table5.5 by (M-Ref).

Lastly, 300 methods are retrieved when the list of M-Ref methods are filtered by measure distance metric (M-Distance). It is clear that using reformulating query technique to filter the execution trace decrease the size of the retrieved methods from the total number of methods of 1000 in a sequence diagram feature, to 450 methods.

Table 5.5: Feature with Number of methods by using reformulating query technique

Feature	No-M	M-gold-set	M-Ref	M-Distance
Sequence Diagram	1000	379	450	300
Activity Diagram	700	288	300	250

In addition, Table 5.5 shows an improving result to use for developer to measure the distance metrics to filter the the number of methods that are retrieved by using Refoqus strategy. We use measuring metrics to find the

distance between the ranked method on M-Ref. One metric is to group all the methods together based on the methods implementation point of view, such as finding the similarity between methods implementation based on the methods lexicon. For instance, the distance between two same methods is less than the distance between two different methods that were invoked by sequence diagram scenario based on the context similarity of the methods.

Another metric is a dynamic metric which refers to provide a distance between two methods based on the execution point of view, such as using the number of time the method is invoked and the execution time for methods to be executed (Hammad & Al-Hawawreh, 2018). Hence, the list of ranked methods for sequence diagram feature that were obtained by using query reformulation are re-ordered based on the methods distance to each other. Each method on the ranked list is revisited to find the similarity based on the text context and based on dynamic view such as a number of time that method is invoked. The result of using distance metrics between methods is to find if the previous method on the list has close similarity, the list is re-ordered to re-position the list of methods. otherwise, the list is remained with no change.

As a conclusion, the result of case study confirms that, locating methods to the desired feature based on analysing one single execution trace by using reformulating query and measuring distance provides a better result than using only one feature location solution.

5.2.1 Results and Discussion of using MappFL to configure AETFS

The MappFL technique plays a pivotal role as part of the AETFS configuration, specifically in performing DFLT. By incorporating the MappFL approach into the AETFS workflow, the framework enhances the efficiency

and accuracy of feature location during testing processes. The results of the DFLT's evaluations are presented in Table 5.6, which depicts the Precision (P), Recall (R), and F-Measure metrics for each dataset. The column labeled 'DFLTs (with MappFL)' represents the outcomes obtained when leveraging the MappFL technique within the AETFS configuration. The table showcases the significant impact of MappFL on improving the performance of feature location compared to the 'DFLTs (without MappFL)' column. These findings underscore the effectiveness of integrating MappFL within the AETFS workflow to enhance feature location accuracy and aid software developers in identifying and testing features more effectively.

Table 5.6: DFLT's Evaluation Results of Using MappFL to Configure AETFS

Dataset	DFLTs (without MappFL)			DFLTs (with MappFL)		
	P	R	F-Measure	P	R	F-Measure
ArgoUML0.22	0.72	0.85	0.78	0.80	0.95	0.87
ArgoUML0.24	0.68	0.91	0.78	0.76	1	0.86
ArgoUML0.26.2	0.79	0.82	0.80	0.89	0.92	0.90
JabRef2.6	0.75	0.84	0.79	0.84	0.94	0.89
jEdit4.3	0.76	0.88	0.81	0.85	0.99	0.91
muCommander0.8.5	0.79	0.91	0.85	0.88	1	0.94
Average Performance Increase			12%			

The presented table (Table 5.6) provides a comprehensive analysis of the performance evaluation results for DFLT's with and without the integration of the MappFL approach in AETFS. The metric P,R, and F-Measure are employed to assess the effectiveness of the techniques across multiple datasets. By comparing the 'DFLTs (without MappFL)' and 'DFLTs (with MappFL)' columns, it is evident that the incorporation of the MappFL technique yields significant improvements in DFLT's accuracy and effectiveness. Across the evaluated datasets, the use of MappFL consistently demonstrates higher P, R, and F-Measure values, indicating a more precise and compre-

hensive identification of features. Notably, the average performance increase achieved by integrating MappFL is approximately 12%.

The metrics P, R, and F-Measure are computed to showcase the average results for the features within the datasets. For example, in Table 5.2, the dataset ArgoUML 0.26.2 exhibits various issues related to software features. One such feature is the Activity Diagram, which can be executed through a potential scenario outlined in Table 5.4. To construct the Activity Diagram feature, 288 methods (gold-set) are required. However, when the Activity Diagram is exercised by the scenario, a total of 700 methods are retrieved, as shown in Table 5.5. According to the evaluation metrics presented in Table 5.1, the results for P, R, and F-Measure are 0.41, 0.91, and 0.51, respectively, without utilizing AETFS. Conversely, when AETFS (M-Ref) is employed, these values change to 0.96, 1, and 0.97, respectively. The percentage increase for P, R, and F-Measure is 134%, 10%, and 90%, respectively, with an average of 78% across the three metrics. Consequently, Table 5.6 illustrates the dataset average by calculating the average of each dataset's issues (features) for all datasets.

These results validate the efficacy of the MappFL technique in enhancing the DFLT's input process, thereby empowering software developers to locate and understand features more efficiently. The findings underscore the potential of MappFL as a valuable configuration in AETFS, enabling effective DFLT's and aiding in the DFLT's' inputs.

In other words, The line graph 5.4 represents the evaluation results of using MappFL as a part of the AETFS configuration technique. The graph shows the F-Measure values for different datasets (D1, D2, D3, D4, D5, and D6), comparing the performance of DFLT's with and without MappFL.

Upon analyzing the graph, it is evident that incorporating MappFL into the AETFS configuration technique has generally improved the F-Measure values across the datasets. The line representing DFLT's with MappFL con-

sistently exhibits higher F-Measure values compared to the line representing DFLT's without MappFL.

Specifically, for datasets D1 (ArgoUML 0.22), D3 (ArgoUML 0.26.2), D4 (JabRef 2.6), D5 (jEdit 4.3), and D6 (muCommander 0.8.5), the F-Measure values with MappFL are significantly higher than those without MappFL. This suggests that the utilization of MappFL enables software developers to address various software issues more effectively, including fault localization, feature location, and others as illustrated in Table 5.2 at Issues Column.

However, for dataset D2 (ArgoUML0.24), the F-Measure values with MappFL are slightly lower compared to those without MappFL. This deviation might suggest that the impact of incorporating MappFL in the AETFS configuration technique varies depending on the specific dataset characteristics. Further investigation is needed to understand the underlying reasons for this discrepancy.

Overall, the line graph demonstrates the effectiveness of MappFL as a component of the AETFS configuration technique, showing improved F-Measure values across multiple datasets. These findings highlight the value of MappFL in enhancing the DFLT's capabilities of the AETFS approach, providing valuable insights for DFLT's inputs formulation.

5.2.2 Threat to Validity

The proposed solution has limitations in implementing the methodology in the case study. The size of the case study is considerably small and the size of the created corpus is relatively limited. Profiling only one feature does not create enough domain that allows the developers to write a query to describe the feature in order to retrieve the source code. Moreover, in the case study, only one programming language was implemented which is Java. Another programming language should be used to validate the accuracy of

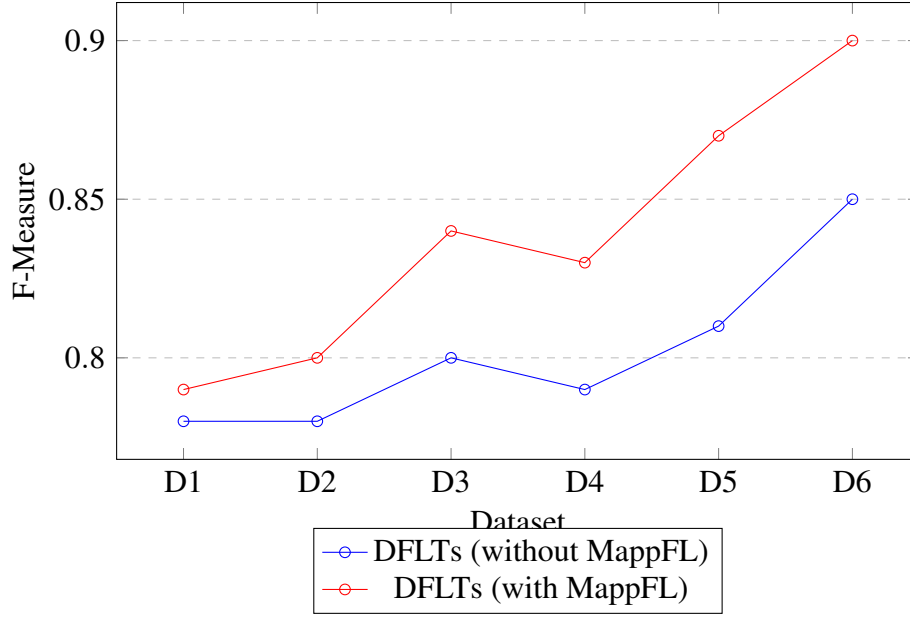


Figure 5.4: Presenting DFLT's Evaluation Results of Using MappFL to Configure AETFS by Using Line Graph

the proposed approach.

However, comparing the obtained results in the case study with the ArogUML benchmark (Martinez et al., 2018) validated the case study implementation. The validation provides a quick view of the accuracy level of the proposed methodology.

5.3 Assessment and Evaluation of Scenario Sifter to Filter Irrelevant Inputs

The use of benchmarks in evaluating the performance of DFLT's is an important direction to advance the state of the art (Strüber et al., 2019) solution in software maintenance. The evaluation was implementing under several strategies. One of them was comparing the result of performing DFLT's with the use of Scenario Sifter, and without using Scenarios Sifter against three

open source projects in Java¹. Two DFLTs were selected for the evaluation (Khouri et al., 2016; Buzaid & Hammad, 2021a). Both of the selected techniques are DFLTs, using scenarios to exhibit the feature of interest to perform the mapping between the exercised feature and the relevant source code based on analysing the generated execution trace. The output of the selected DFLTs should include a list of classes, methods, or any software source code granularity level that are related to the feature which can be used to impalement the evaluation against the dataset benchmark². The proposed approach increases the accuracy of performing DFLTs up to 45%.

Figure 5.5 illustrates the methodology process to compute the evaluation metrics for every software subjects. First step was to apply DFLTs (Khouri et al., 2016; Buzaid & Hammad, 2021a) on the all three subject systems to locate the required source code for each targeted feature. The output result stored on a designated repository. Then, the result was computed with the proposed metrics to find the P, R, and F measure with the ground truth. The same process was repeated again, enhancing DFLTs (Khouri et al., 2016; Buzaid & Hammad, 2021a) with Scenario Sifter.

To evaluate the result of using Scenario Sifter in increasing the accuracy of performing DFLTs, a comparison of using the proposed technique in enhancing the performance of DFLTs is implemented. Evaluation scores are used for the evaluation, such as P, R, and F-Measure (Beranič, Podgorelec, & Heričko, 2018).

Table 5.7 shows the P, R, and F-Measure result of using Scenario Sifter to perform FL by using a DFLT that based on Gestalt theory and the Helmholtz principle (Khouri et al., 2016) in three Java open-source projects (i.e., ArgoUML-SPL, Notepad, and Sudoku). The developer was unfamiliar with the targeted three systems. The developer was asked to perform FL by using the opted

¹Java is a high-level, class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible

²<https://github.com/fbuzaid/Scenario-Sifter/dataset>

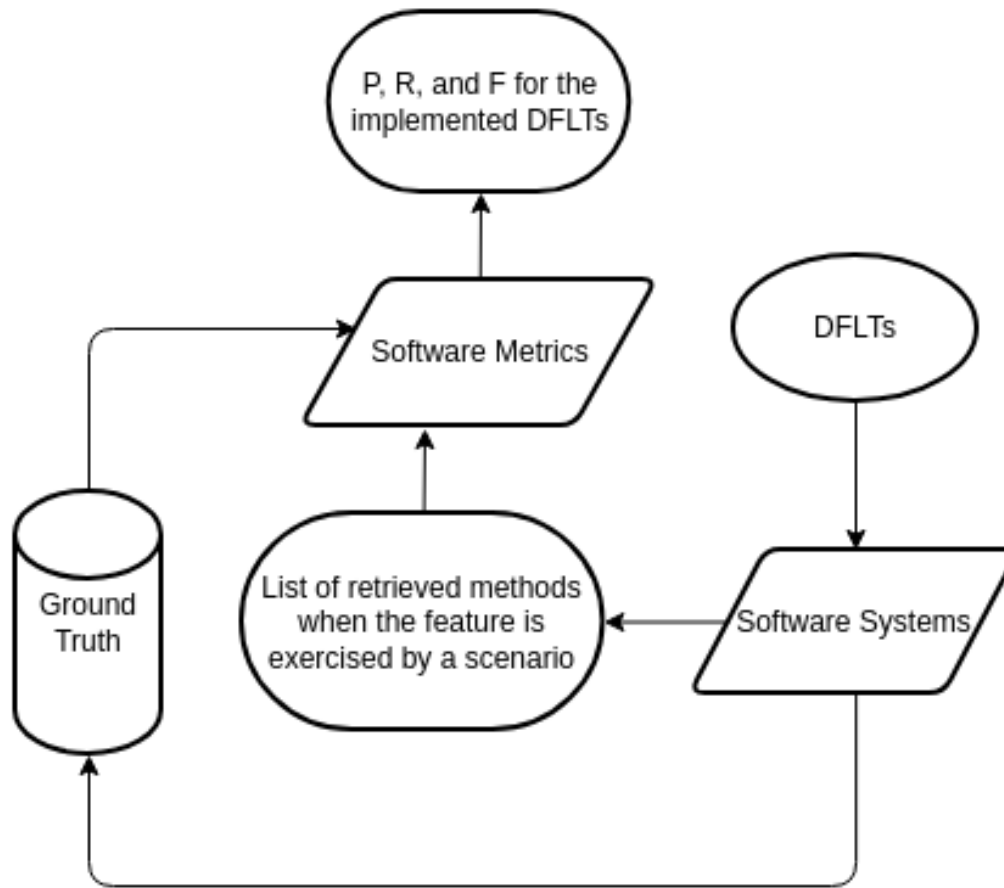


Figure 5.5: Evaluation process to measure the P, R, and F for each implemented DFLTs

DFLT (Khoury et al., 2016) with using Scenario Sifter, and the obtained result compared against the benchmark ground-truth for the three systems³. The scenarios were implemented by the developer to exercise six features in ArgoUML-SPL, Notepad, and Sudoku. Moreover, text query was formulated by the developer for each exhibited feature. It was used to describe the feature of interest in NL.

³<https://github.com/scenario-sifter/ground-truth>

Table 5.7: An evaluation of using Scenario Sifter to enhance the performance of DFLT against three Java open-source benchmark

ArgoUML-SPL with using Scenario Sifter in DFLT (Khoury et al., 2016)				
Feature ID	Feature Name	Precision (P)	Recall (R)	F-Measure (F)
F_1	State Diagrams	0.560	1.000	0.717
F_2	Activity Diagrams	0.480	1.000	0.648
F_3	Use Case Diagrams	0.849	1.000	0.918
F_4	Collaboration Diagrams	0.519	1.000	0.683
F_5	Deployment Diagrams	0.751	1.000	0.857
F_6	Sequence Diagrams	0.663	1.000	0.797
Notepad with using Scenario Sifter in DFLT (Khoury et al., 2016)				
F_1	Fonts	0.441	1.000	0.612
F_2	Open	0.403	1.000	0.574
F_3	Print	0.391	1.000	0.562
F_4	Save	0.340	1.000	0.507
F_5	Copy	0.327	1.000	0.492
F_6	Cut	0.339	1.000	0.506
Sudoku with using Scenario Sifter in DFLT (Khoury et al., 2016)				
F_1	States	0.344	1.000	0.511
F_2	Solver	0.401	1.000	0.572
F_3 1	Generator	0.355	1.000	0.523
F_4	Undo	0.347	1.000	0.515
F_5	Hint	0.333	1.000	0.499
F_6	Extended	0.341	1.000	0.508

5.3.1 Results and Discussion of using Scenario Sifter to configure AETFS

The provided table, Table 5.8, presents an evaluation of the DFLT (Khoury et al., 2016) in three Java open-source projects: ArgoUML-SPL, Notepad, and Sudoku. Specifically, the table showcases the Precision (P), Recall (R), and F-Measure (F) results obtained without enhancing the scenarios using Scenario Sifter. The developer, who was unfamiliar with these three systems, was tasked with performing fault localization using the DFLT (Khoury et al., 2016). The obtained results were then compared against the benchmark ground truth for each of the three systems. The table in-

cludes the evaluation metrics for six features exercised in ArgoUML-SPL, Notepad, and Sudoku, respectively. The P, R, and F-measure values provide insights into the effectiveness of the DFLT technique without the aid of Scenario Sifter.

Table 5.8 shows the P, R, and F result of using DFLT (Khoury et al., 2016) in three Java open-source projects (i.e., ArgoUML-SPL, Notepad, and Sudoku) without enhancing the scenario by using Scenario Sifter. The developer was unfamiliar with the targeted three systems. The developer was asked to perform FL by using DFLT (Khoury et al., 2016), and the obtained result compared against the benchmark ground truth for the three systems. The scenarios were implemented by the developer to exercise six features in ArgoUML-SPL, Notepad, and Sudoku.

Similar comparisons for the Table 5.7 and Table 5.8 were implemented in Table 5.9 and Table 5.10 to evaluate the result by using state of art DFLT called MappFL (Buzaid & Hammad, 2021a).

The comparison for the evaluations in Tables 5.7 and 5.8 were implemented based on comparing the results of locating the relevant methods to the exhibited features. DFLT (Khoury et al., 2016) was implemented in six features for every software system. The comparison was implemented based on using Scenario Sifter and without using Scenario Sifter. Figure 5.6 illustrates the comparison results for using DFLT (Khoury et al., 2016) with and without using Scenario Sifter respectively. Six features (i.e., F_1, F_2, F_3, F_4, F_5 , and F_6) were evaluated $F - Measure$ metric to record the effectiveness of using Scenario Sifter to increase the accuracy of using DFLT in (Khoury et al., 2016). The accuracy of performing DFL in (Khoury et al., 2016) increased up to 35% when Scenario Sifter was embedded on the

Table 5.8: An evaluation of using DFLT against three Java open-source benchmark without using Scenario Sifter

ArgoUML-SPL without using Scenario Sifter in DFLT (Khoury et al., 2016)				
Feature ID	Feature Name	Precision (P)	Recall (R)	F-Measure (F)
F_1	State Diagrams	0.420	1.000	0.591
F_2	Activity Diagrams	0.360	1.000	0.529
F_3	Use Case Diagrams	0.636	1.000	0.777
F_4	Collaboration Diagrams	0.389	1.000	0.560
F_5	Deployment Diagrams	0.563	1.000	0.720
F_6	Sequence Diagrams	0.497	1.000	0.663
Notepad without using Scenario Sifter in DFLT (Khoury et al., 2016)				
F_1	Fonts	0.270	1.000	0.425
F_2	Open	0.302	1.000	0.463
F_3	Print	0.293	1.000	0.453
F_4	Save	0.255	1.000	0.406
F_5	Copy	0.245	1.000	0.393
F_6	Cut	0.254	1.000	0.405
Sudoku without using Scenario Sifter in DFLT (Khoury et al., 2016)				
F_1	States	0.258	1.000	0.410
F_2	Solver	0.300	1.000	0.461
F_3 1	Generator	0.266	1.000	0.420
F_4	Undo	0.260	1.000	0.412
F_5	Hint	0.249	1.000	0.398
F_6	Extended	0.255	1.000	0.406

opted DFLT (Khoury et al., 2016) as shown in Figure 5.6. The average of F-Measure was 0.8 for ArgoUML-SPL, and 0.5 for both Notepad, and Sudoku systems.

Similarly, Figure 5.7 shows the comparison result with using state-of-art solution for DFLT called MappFL (Buzaid & Hammad, 2021a) with the use of Scenario Sifter and without the use of Scenario Sifter. In this case, the accuracy of performing FL was increased up to 10% when Scenario Sifter was embedded on the state-of-art DFLT (Buzaid & Hammad, 2021a).

Table 5.9: An evaluation of using state-of-art DFLTs that enhanced by using Scenario Sifter against three Java open-source benchmark

ArgoUML-SPL with using Scenario Sifter in MappFL (Buzaid & Hammad, 2021a)				
Feature ID	Software Features	Precision (P)	Recall (R)	F-Measure (F)
F_1	State Diagrams	0.616	1.000	0.762
F_2	Activity Diagrams	0.528	1.000	0.691
F_3	Use Case Diagrams	0.933	1.000	0.965
F_4	Collaboration Diagrams	0.570	1.000	0.726
F_5	Deployment Diagrams	0.826	1.000	0.904
F_6	Sequence Diagrams	0.729	1.000	0.843
Notepad with using Scenario Sifter in MappFL (Buzaid & Hammad, 2021a)				
F_1	Fonts	0.485	1.000	0.653
F_2	Open	0.443	1.000	0.613
F_3	Print	0.430	1.000	0.601
F_4	Save	0.374	1.000	0.544
F_5	Copy	0.359	1.000	0.528
F_6	Cut	0.372	1.000	0.542
Sudoku with using Scenario Sifter in MappFL (Buzaid & Hammad, 2021a)				
F_1	States	0.378	1.000	0.548
F_2	Solver	0.441	1.000	0.612
F_3 1	Generator	0.390	1.000	0.561
F_4	Undo	0.381	1.000	0.551
F_5	Hint	0.366	1.000	0.535
F_6	Extended	0.375	1.000	0.545

5.3.2 Threat to Validity

The Scenario Sifter faced several difficulties from the implementation to evaluation. The main threat in the implementation was the overhead on OSIA to perform DFLTs. It presses an inevitable runtime overhead on the system given that additional prob code is added to the system. Since runtime overheads are an unpleasant side effect, a substantial effort is generally used to create efficient instrumentation (Cassar, Francalanza, & Said, 2015) that reduce this overhead. Synchronous instrumentation can be lowered to be semi-synchronous instrumentation to decrease the overhead side effect. An

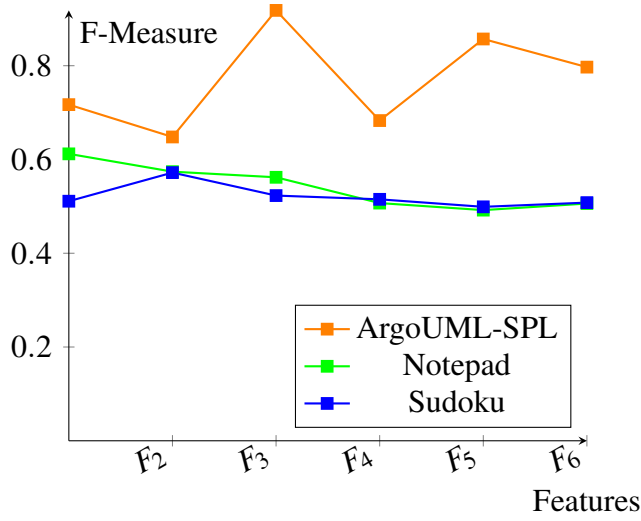
Table 5.10: An evaluation of using DFLT called MappFL against three Java open-source benchmark without using Scenario Sifter

ArgoUML-SPL without using Scenario Sifter in MappFL (Buzaid & Hammad, 2021a)				
Feature ID	Software Features	Precision (P)	Recall (R)	F-Measure (F)
F_1	State Diagrams	0.462	1.000	0.632
F_2	Activity Diagrams	0.396	1.000	0.567
F_3	Use Case Diagrams	0.699	1.000	0.822
F_4	Collaboration Diagrams	0.427	1.000	0.598
F_5	Deployment Diagrams	0.619	1.000	0.764
F_6	Sequence Diagrams	0.546	1.000	0.706
Notepad without using Scenario Sifter in MappFL (Buzaid & Hammad, 2021a)				
F_1	Fonts	0.297	1.000	0.457
F_2	Open	0.332	1.000	0.498
F_3	Print	0.293	1.000	0.453
F_4	Save	0.322	1.000	0.487
F_5	Copy	0.269	1.000	0.423
F_6	Cut	0.279	1.000	0.436
Sudoku without using Scenario Sifter in MappFL (Buzaid & Hammad, 2021a)				
F_1	States	0.283	1.000	0.441
F_2	Solver	0.330	1.000	0.496
F_{31}	Generator	0.292	1.000	0.452
F_4	Undo	0.286	1.000	0.444
F_5	Hint	0.273	1.000	0.428
F_6	Extended	0.280	1.000	0.437

intermediate approach between full online synchronous instrumentation and semi full online synchronous instrumentation provide a trade-off upon the system (Zhang et al., 2016).

In proceeding execution trace to construct term corpus, as in general IR techniques, the quality of the retrieved data can be effected by the input query. Problems such as the dictionary mismatch problem, as well as the tendency of users to issue short queries, or long query have affect semantic similarity retrieval data. Measuring the quality of the query (Mills et al., 2017) in advance of the runtime in DFLTs can be used to mitigate these problems allows the user to predict his or her queries effectiveness in

With Scenario Sifter by using DFLT (Khoury et al., 2016)



Without Scenario Sifter by using DFLT (Khoury et al., 2016)

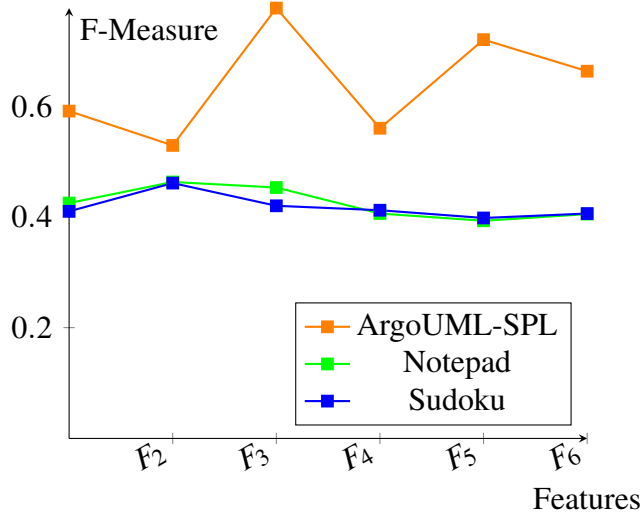
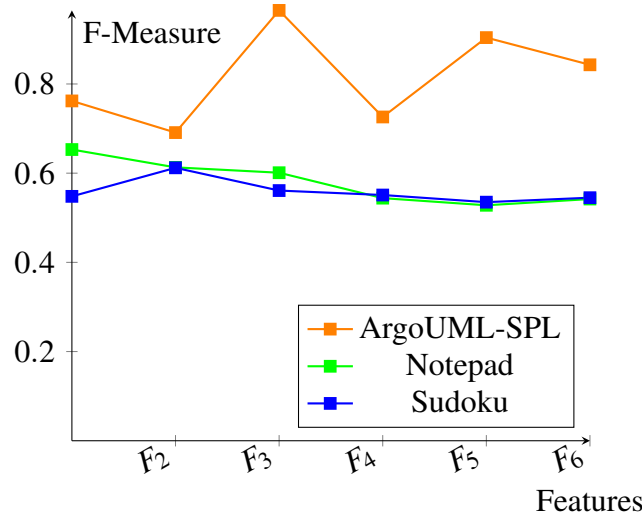


Figure 5.6: Comparison between the performance of DFLT's with using Scenario Sifter and without using Scenario Sifter.

retrieving the data. The proposed mitigation would enhance the Scenario Sifter to prevent neglecting an important part of the scenario due to the lack of creating an efficient query to determine whether that part of the scenario is important or not.

In the evaluation, two DFLT's were implemented to evaluate the effectiveness of using Scenario Sifter in increasing the accuracy of the FL perfor-

With Scenario Sifter by using DFLT (Buzaid & Hammad, 2021a)



Without Scenario Sifter by using DFLT (Buzaid & Hammad, 2021a)

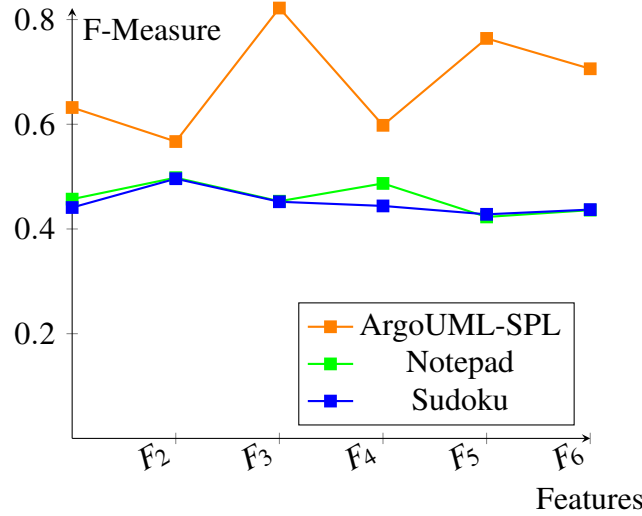


Figure 5.7: Comparison between the performance of DFLT's with using Scenario Sifter and without using Scenario Sifter.

mance. Both of the DFLT's were compared against three software systems, and the result recorded based on the use of Scenario Sifter and without the use of Scenario Sifter to prevent bias judgement when only one case was implemented, such as with use Scenario Sifter. Moreover, the selected three software systems were included a medium size software with a ground-truth data that well known used for FL evaluation (*i.e.*, ArgoUML-SPL), and relatively two small applications that analysed manually on the proposed eval-

uation to prevent any bias judgement based on the targeted application.

5.4 Assessment and Evaluation of The Use of UML Diagrams to Enhance DFLTs

In the case study, a Java application for an Automated Teller Machine (ATM) was utilized for evaluation purposes. The ATM system comprises three primary classes with multiple fields and functionalities, amounting to approximately 500 lines of code. The system includes a significant number of objects and methods, enabling functionalities such as account management, fund transactions, and balance inquiries. To evaluate the system, a specific functionality was tested, and the results were meticulously documented for analysis. The subsequent subsection provides a detailed overview of the ATM system implementation, focusing on classes, fields, and other relevant details. The source code for the ATM application can be found on GitHub⁴

Item	Count	Description
Classes	3	Primary classes in the ATM system
Methods	17	Total number of methods in the ATM system
Objects	High	Numerous objects instantiated during operation

Table 5.11: Summary of ATM System Components

The Table 5.11, labeled "Summary of ATM System Components," provides a concise overview of the key structural elements within the ATM system. It delineates the system's composition by enumerating the presence of three primary classes that form the backbone of the ATM application. These classes encapsulate various fields and functionalities crucial for the

⁴<https://github.com/buzaidf/DFLTs-Input>

Method Name	Purpose	Input Parameters	Return Type
setCustomerNumber	Sets customer number	int customerNumber	int
getCustomerNumber	Gets customer number	-	int
setPinNumber	Sets PIN number	int pinNumber	int
getPinNumber	Gets PIN number	-	int
getCheckingBalance	Gets checking account balance	-	double
getSavingBalance	Gets saving account balance	-	double
calcCheckingWithdraw	Calculates checking withdrawal	double amount	double
calcSavingWithdraw	Calculates saving withdrawal	double amount	double
calcCheckingDeposit	Calculates checking deposit	double amount	double
calcSavingDeposit	Calculates saving deposit	double amount	double
calcCheckTransfer	Transfers from checking to saving	double amount	void
calcSavingTransfer	Transfers from saving to checking	double amount	void
getCheckingWithdrawInput	Gets input for checking withdrawal	-	void
getSavingWithdrawInput	Gets input for saving withdrawal	-	void
getCheckingDepositInput	Gets input for checking deposit	-	void
getSavingDepositInput	Gets input for saving deposit	-	void
getTransferInput	Gets input for fund transfer	String accType	void

Table 5.12: Detailed Analysis of ATM System Methods

system's operation. Furthermore, the table quantifies the system's complexity by listing a total of 17 methods employed within the ATM system, signifying the breadth of operations the system can undertake. Notably, the table hints at the extensive instantiation of objects during the system's runtime, emphasizing the dynamic and interactive nature of the ATM application.

Conversely, the Table 5.12, titled "Detailed Analysis of ATM System Methods," offers a granular examination of the individual methods that collectively empower the ATM system's diverse functionalities. Each method is meticulously documented with its designated purpose, specifying the input parameters required for execution and the respective return types. This detailed breakdown facilitates a comprehensive understanding of the inner workings of the ATM system, shedding light on how specific actions such as setting customer information, managing account balances, processing withdrawals and deposits, and enabling fund transfers are orchestrated within

the application. Collectively, these tables serve as vital reference points for comprehending the structural foundation and operational intricacies of the ATM system, underpinning the subsequent in-depth analysis and evaluation of its software features.

5.4.1 Converting the Text Scenario into a Use Case Diagram

The objective of this subsection was to evaluate the proposed methodology for converting a text story to a use case diagram. It was proposed to automate this process by using the eclipse plugin (Elallaoui et al., 2018) to convert the user's text story to a UML use case diagram. The developer was commanded to write a text scenario describing how to exercise the software functionality of the ATM to create an account and deposit a fund for the new customer. A UML case diagram was generated automatically based on the user's text scenario. Fig 5.8 shows the result of converting the user's text story into a UML use case diagram to exercise the requested software feature in the ATM application. Hence, only exhibited software features were generated in the created use case diagram.

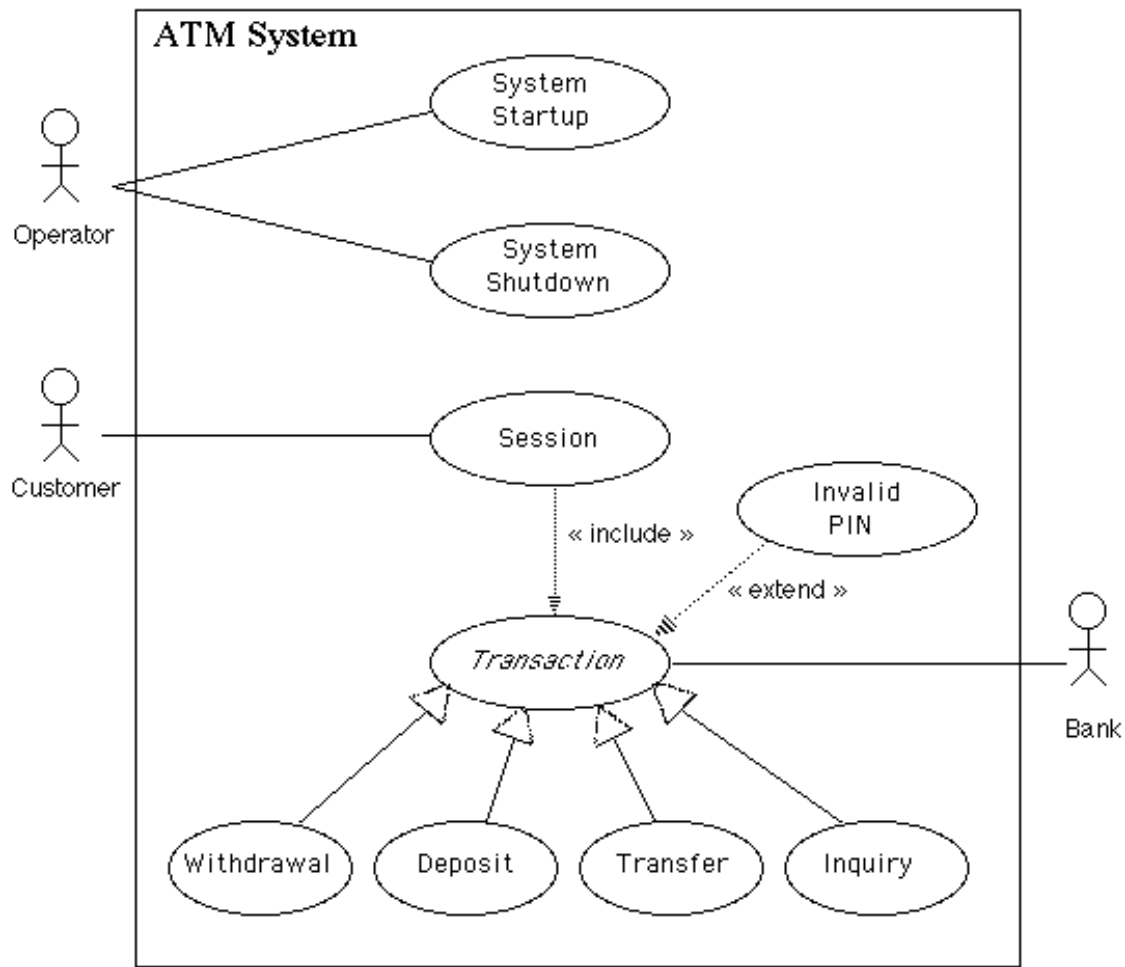


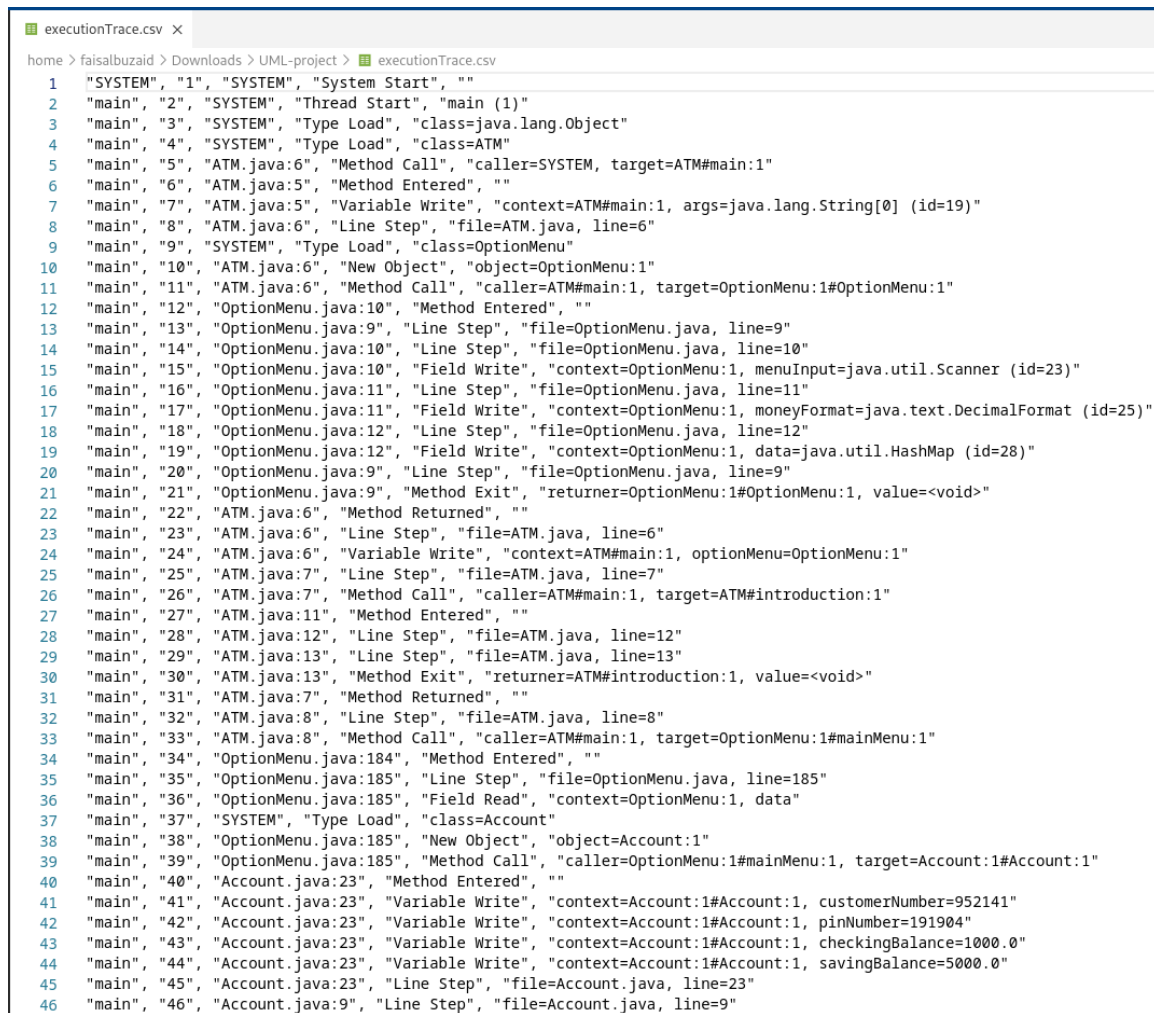
Figure 5.8: The generated UML use case diagram based on the text scenario.

Several tools were used to enhance the process of converting the text scenario into a use case diagram scenario. Terms in the text story were categorized into proper noun, personal nouns, and verbs by using a tool called TreeTagger parser. Moreover, PlantUML tool⁵ was used to display diagrams in UML format.

⁵<https://plantuml.com/>

5.4.2 Generating and Collecting Execution Traces

Generating execution traces is an essential step in dynamic analysis. Executing the use case scenario diagram generated an execution trace which presented the behavior of the exercised software feature. The collected execution trace was a text file formatted based on the proposed approach. Fig. 5.9 presents a snapshot of the generated execution trace by using the AspectJ tracer.



```
1 "SYSTEM", "1", "SYSTEM", "System Start", ""
2 "main", "2", "SYSTEM", "Thread Start", "main (1)"
3 "main", "3", "SYSTEM", "Type Load", "class=java.lang.Object"
4 "main", "4", "SYSTEM", "Type Load", "class=ATM"
5 "main", "5", "ATM.java:6", "Method Call", "caller=SYSTEM, target=ATM#main:1"
6 "main", "6", "ATM.java:5", "Method Entered", ""
7 "main", "7", "ATM.java:5", "Variable Write", "context=ATM#main:1, args=java.lang.String[0] (id=19)"
8 "main", "8", "ATM.java:6", "Line Step", "file=ATM.java, line=6"
9 "main", "9", "SYSTEM", "Type Load", "class=OptionMenu"
10 "main", "10", "ATM.java:6", "New Object", "object=OptionMenu:1"
11 "main", "11", "ATM.java:6", "Method Call", "caller=ATM#main:1, target=OptionMenu:1#OptionMenu:1"
12 "main", "12", "OptionMenu.java:10", "Method Entered", ""
13 "main", "13", "OptionMenu.java:9", "Line Step", "file=OptionMenu.java, line=9"
14 "main", "14", "OptionMenu.java:10", "Line Step", "file=OptionMenu.java, line=10"
15 "main", "15", "OptionMenu.java:10", "Field Write", "context=OptionMenu:1, menuInput=java.util.Scanner (id=23)"
16 "main", "16", "OptionMenu.java:11", "Line Step", "file=OptionMenu.java, line=11"
17 "main", "17", "OptionMenu.java:11", "Field Write", "context=OptionMenu:1, moneyFormat=java.text.DecimalFormat (id=25)"
18 "main", "18", "OptionMenu.java:12", "Line Step", "file=OptionMenu.java, line=12"
19 "main", "19", "OptionMenu.java:12", "Field Write", "context=OptionMenu:1, data=java.util.HashMap (id=28)"
20 "main", "20", "OptionMenu.java:9", "Line Step", "file=OptionMenu.java, line=9"
21 "main", "21", "OptionMenu.java:9", "Method Exit", "returner=OptionMenu:1#OptionMenu:1, value=<void>"
22 "main", "22", "ATM.java:6", "Method Returned", ""
23 "main", "23", "ATM.java:6", "Line Step", "file=ATM.java, line=6"
24 "main", "24", "ATM.java:6", "Variable Write", "context=ATM#main:1, optionMenu=OptionMenu:1"
25 "main", "25", "ATM.java:7", "Line Step", "file=ATM.java, line=7"
26 "main", "26", "ATM.java:7", "Method Call", "caller=ATM#main:1, target=ATM#introduction:1"
27 "main", "27", "ATM.java:11", "Method Entered", ""
28 "main", "28", "ATM.java:12", "Line Step", "file=ATM.java, line=12"
29 "main", "29", "ATM.java:13", "Line Step", "file=ATM.java, line=13"
30 "main", "30", "ATM.java:13", "Method Exit", "returner=ATM#introduction:1, value=<void>"
31 "main", "31", "ATM.java:7", "Method Returned", ""
32 "main", "32", "ATM.java:8", "Line Step", "file=ATM.java, line=8"
33 "main", "33", "ATM.java:8", "Method Call", "caller=ATM#main:1, target=OptionMenu:1#mainMenu:1"
34 "main", "34", "OptionMenu.java:184", "Method Entered", ""
35 "main", "35", "OptionMenu.java:185", "Line Step", "file=OptionMenu.java, line=185"
36 "main", "36", "OptionMenu.java:185", "Field Read", "context=OptionMenu:1, data"
37 "main", "37", "SYSTEM", "Type Load", "class=Account"
38 "main", "38", "OptionMenu.java:185", "New Object", "object=Account:1"
39 "main", "39", "OptionMenu.java:185", "Method Call", "caller=OptionMenu:1#mainMenu:1, target=Account:1#Account:1"
40 "main", "40", "Account.java:23", "Method Entered", ""
41 "main", "41", "Account.java:23", "Variable Write", "context=Account:1#Account:1, customerNumber=952141"
42 "main", "42", "Account.java:23", "Variable Write", "context=Account:1#Account:1, pinNumber=191904"
43 "main", "43", "Account.java:23", "Variable Write", "context=Account:1#Account:1, checkingBalance=1000.0"
44 "main", "44", "Account.java:23", "Variable Write", "context=Account:1#Account:1, savingBalance=5000.0"
45 "main", "45", "Account.java:23", "Line Step", "file=Account.java, line=23"
46 "main", "46", "Account.java:9", "Line Step", "file=Account.java, line=9"
```

Figure 5.9: Snapshot of the generated execution trace by using AspectJ to profile a software functionality.

The collected execution trace consisted of several fields, and each field

was dedicated to a different aspect of the analysis. For instance, the execution trace included event, filed read, filed write, methods call, method exit, etc. Recording the size of the ATM application, a single thread system was executed. On the execution trace context, line number indicates the set of events that were traced when the program gets executed, such as caller, target, class, method, object, etc that were invoked when the use case scenario was executed.

5.4.3 Trace Filtering

Despite the size of the ATM application, the size of the collected execution trace was relatively large. It included irrelevant data to the scenario called noise (Antoniol & Guéhéneuc, 2006). The number of utility data that were invoked while executing the scenario affected the quality of the execution trace to construct the UML object diagram. The tracematch filtering technique controlled the size of the collected execution trace by using a regular expression to delete all the noise. For instance, the tracematch regular expression technique set to filter to delete all the context with the prefix such as the Java Development Kit (JDK)⁶, org.eclipse, org.apache, and javax. Table 1 shows all the prefix-list that were applied to filter the execution trace by using the tracematch technique.

Excluding specific data, such as the prefixes listed in the Table 5.13, from the execution trace can significantly enhance the performance of dynamic analysis processes. By filtering out these prefixes, which are likely related to system libraries, frameworks, or external dependencies, the size of the execution trace can be reduced. This reduction in size leads to improved efficiency during dynamic analysis as it focuses on the relevant parts of the codebase rather than getting bogged down by the excessive details from external sources. Filtering out these prefixes helps in streamlining the

⁶<https://www.oracle.com/java/technologies/>

Table 5.13: Tracematch technique uses the list of prefix to filter the execution trace.

List of Prefix			
antlr.*	com.apple.*	com.mysql.*	com.sun.*
java.*	javax.*	org.apache.*	org.eclipse.*
org.hibernate.*	org.jcp.*	org.junit.*	org.postgresql.*
org.w3c.*	org.xml.*	sun.*	jdk.*

analysis process, allowing for quicker processing, easier identification of critical issues, and a more targeted approach to understanding the behavior of the system under examination. This optimization not only speeds up the analysis but also enhances the accuracy and effectiveness of the dynamic analysis techniques being employed.

5.4.4 Visualizing the Object Diagram by Using Execution Trace

Creating a UML diagram like an object diagram by using an execution trace enhances the DFLT's to map between the source code and executed scenario. The object diagram depicted an easy visualization view that can be used in DFLT's to understand a basic knowledge about the executed scenario's behavior. The inner behavior interaction in the ATM application was presented by an object diagram to locate a snapshot of the source code to the exercised feature. Fig. 5.10 showed only the relevant objects, methods, and classes to the executed scenario were invoked. Hence, the object diagram in Fig. 5.10 illustrated to the developer which basic data were relevant to the exercised feature of interest.

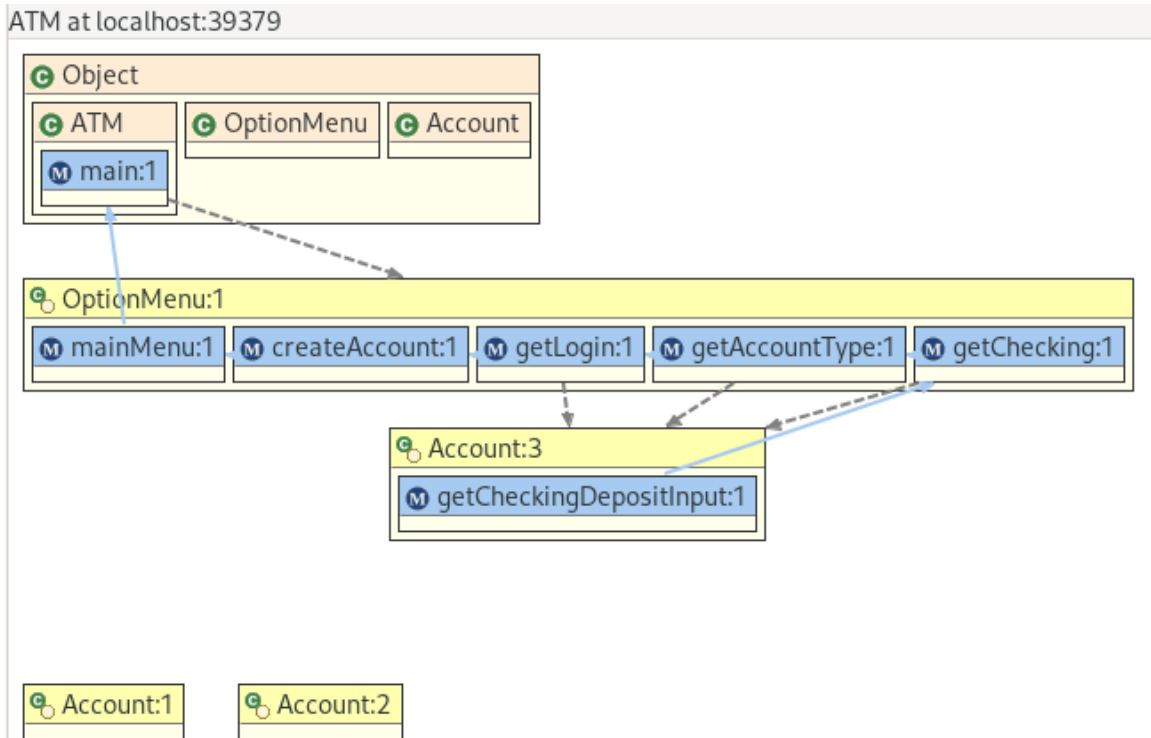


Figure 5.10: Snapshot of the generated UML Object Diagram by using JIVE tool.

For instance, the use case scenario diagram in Fig. 5.9 exercised the software feature in the ATM application to show how the developer can execute the feature to create a bank account and deposit a fund. Fig. 5.10 illustrated a basic and precise approach in which source code was required to handle the task for the exhibited software feature.

However, constructing a UML object diagram required using the supporting tool. PlantUML tool was used to help in creating the required diagram. It provided the developer with plug-in functionalities that were compatible with the Eclipse environment. The syntax of the proposed supported tool was defined by using a simple and intuitive language. For example, the UML diagram begin with *@startuml* tag and the end of the diagram end with *@enduml* tag.

5.4.5 Visualizing the Class Diagram by Using Execution Trace

Fig. 5.11 depicts the required classes that are used to implement the proposed use case plugin. Each class in UML class diagram has a specific functionality to implement (Planas & Cabot, 2020). For example, the `ComponentInitializer` class aims to generate all components of the use case diagram, such as creating the method to generate actors, creating a method to generate the use case, and creating a method to represent their association relation. Creating a model, generating a model, or updating the model are handled by `ModelInitializer` class. Moreover, the collection of actors is presented by the `Model` class. Finally, the created plugin accepts the text sentences based on the syntax proposed by Wautelet *et al* (Wautelet, Heng, Kolp, & Mirbel, 2014). For instance, the developer should start the sentence with (I want to), (I can), or (I am able) to exercise the feature of interest. The plugin's output is managed by `FileGenerator` class, which consists of three methods, `write method`, `profileExist method`, and `generateOutPut method`. Each generation model is specified by a unique identifier managed by `IdCreator` class.

The objective of creating the Java plugin described is to streamline and automate the process of generating use cases from text scenario stories. By developing this plugin, the aim is to enhance software usability by facilitating the conversion of textual scenarios into structured use case diagrams. The plugin utilizes a structured approach, as depicted in a UML class diagram, to model the functionality required for generating these use case diagrams. Specific classes within the plugin, such as `ComponentInitializer`, `ModelInitializer`, `Model`, `FileGenerator`, and `IdCreator`, are designed with distinct functionalities to handle various aspects of the use case generation process. The plugin's design allows for the interpretation of text inputs following predefined syntax guidelines, enabling developers to eas-

ily map textual descriptions to specific software features. Ultimately, the creation of this plugin aims to improve the efficiency and accuracy of use case generation by automating the translation of text scenarios into visual representations, thereby contributing to the success of software solutions.

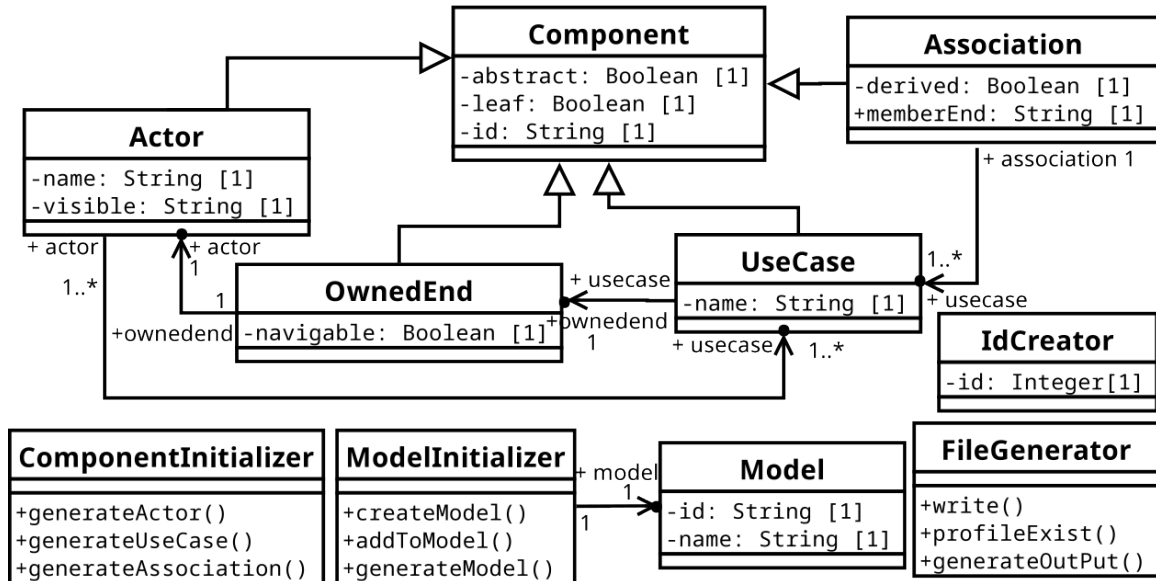


Figure 5.11: The UML class diagram template of Meta-model to generate use case diagram (Elallaoui et al., 2018)

5.4.6 Results and Discussion of using UML Use Case Diagram to configure AETFS

The result of the third configuration of the AETFS by using UML is up to 17% of the accuracy in the DFLT's is increased when the developers are precise in the input of DFLT's. Therefore, solving the problem of creating DFLT's inputs motivates the software researchers to enhance the developers in determining the accurate inputs of DFLT's and ease the difficulties of understanding the relationship between the software input and execution trace in dynamic analysis.

Based on the provided table, Table 5.6, I have created a comparison table that demonstrates the increase in precision (P), recall (R), and F-measure (F) when utilizing the UML Use Case technique.

Table 5.14: Comparison of DFLTs Performance with and without UML Use Case Techniques in AETFS Configuration

Dataset	DFLTs (without UML Use Case)	DFLTs (with UML Use Case)	Performance Increase
ArgoUML0.22	P: 0.72, R: 0.85, F: 0.78	P: 0.77, R: 1, F: 0.87	P: +7%, R: +18%, F: +12%
ArgoUML0.24	P: 0.68, R: 0.91, F: 0.78	P: 0.78, R: 1, F: 0.88	P: +15%, R: +10%, F: +13%
ArgoUML0.26.2	P: 0.79, R: 0.82, F: 0.80	P: 0.95, R: 1, F: 0.97	P: +20%, R: +22%, F: +21%
JabRef2.6	P: 0.75, R: 0.84, F: 0.79	P: 0.88, R: 0.99, F: 0.93	P: +17%, R: +18%, F: +18%
jEdit4.3	P: 0.76, R: 0.88, F: 0.81	P: 0.92, R: 0.98, F: 0.94	P: +21%, R: +11%, F: +16%
muCommander0.8.5	P: 0.79, R: 0.91, F: 0.85	P: 0.92, R: 1, F: 0.95	P: +17%, R: +10%, F: +12%
Average Performance Increase	N/A		17%

Table 5.14 compares the DFLT performance without the UML Use Case technique to the performance with the UML Use Case technique implemented. It includes the precision, recall, and F-measure values for each dataset in both scenarios, as well as the percentage increase in performance.

The table shows that when the UML Use Case technique is employed, there is an improvement in precision, recall, and F-measure for most datasets. The percentage increase varies for each metric and dataset.

However, in our proposed solution configuration of the AETFS, we leverage the ATM basic system to exemplify the application of UML Use Case diagrams. To conduct a comprehensive analysis of the results, we utilize six real project size datasets to compare the performance of DFLTs with and without the incorporation of UML Use Case techniques. The comparison, as depicted in Table 5.14, demonstrates substantial performance improvements when UML Use Case techniques are employed across various datasets, showcasing enhancements in P, R, and F-Measure. Notably, the average performance increase across the datasets stands at 17%, underscoring the effectiveness of integrating UML Use Case techniques within the AETFS configuration for enhanced performance outcomes.

The line chart diagram 5.12 presents a comparison of the performance of DFLTs with and without the utilization of UML Use Case techniques (Buzaid, Albalooshi, & Elmedany, 2022). The x-axis represents the datasets, labeled as D1, D2, D3, D4, D5, and D6, while the y-axis represents the performance metric.

The blue line on the graph represents the performance of DFLTs without employing UML Use Case techniques, while the red line represents the performance when UML Use Case techniques are incorporated.

Analyzing the chart, we observe that the performance varies across different datasets. In the case of D1 (ArgoUML0.22), the red line slightly surpasses the blue line, indicating a performance increase of 7% in precision

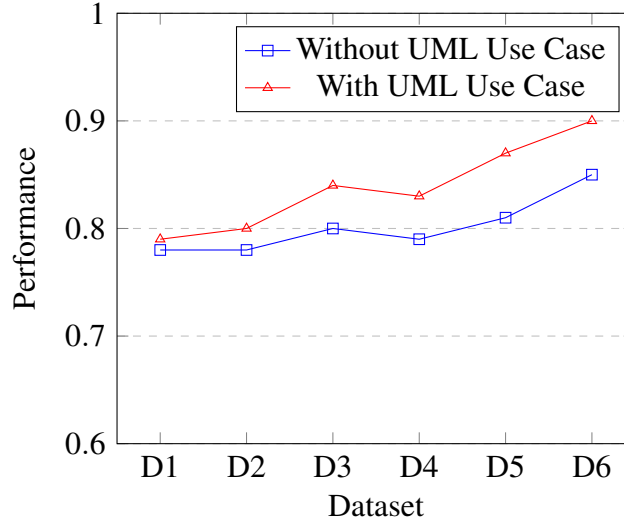


Figure 5.12: Performance Comparison of DFLTs with and without UML Use Case Techniques

(P) and 1% in F1-score (F). However, the recall (R) decreases by 2%.

For D2 (ArgoUML0.24), the blue line outperforms the red line in precision, but the red line exhibits a 10% improvement in recall and a 2% increase in F1-score.

Moving on to D3 (ArgoUML0.26.2), the red line shows a substantial improvement in precision (9%) and F1-score (5%) compared to the blue line, with a minimal increase in recall (1%).

In the case of D4 (JabRef2.6), D5 (jEdit4.3), and D6 (muCommander0.8.5), the red line consistently outperforms the blue line in all performance metrics, demonstrating performance gains ranging from 3% to 8% across precision, recall, and F1-score.

Overall, the line chart analysis suggests that the incorporation of UML Use Case techniques in DFLT yields improvements in performance for most datasets. However, the extent of improvement varies depending on the dataset and the specific performance metric considered. Consequently, researchers and practitioners can use this information to make informed decisions regarding the utilization of UML Use Case techniques in their DFLT.

5.4.7 Threat to Validity

The use case study used a relatively small size of a java application to evaluate the proposed methodology. The result of the case study cannot be used to measure the effectiveness of the proposed approach for a real-life software system without implementing a case study that may contain KLOC. However, the used application had the required number of methods, objects, and classes to evaluate the correctness of the proposed methodology. In addition, some minor changes to the source code were implemented to address the cases of the limited number of classes and objects. For instance, the conversion automated process of transferring a text story to a use case diagram faced difficulties in handling a complex text sentence. A text simplification technique (Siddharthan, 2014) was used to mitigate the sentence complexity.

However, the execution of the use case scenario in the ATM application generated a large size of the execution trace. Dynamic analysis monitored all invoked events in the running program, including the packages and import libraries. Filtering the execution trace manually was a tangled task and time-consuming. Therefore, filtering the execution trace automatically by using the tracematch technique was an important process in the proposed methodology. For instance, regular expression was used to element all imported external libraries from JDK.

Although eliminating the noise from the execution trace decreased the file size, visualizing reverse-engineering creates difficulties during the visualization process. Understanding the behavior of the Object-Oriented Programming language (OOP) such as Java is a challenging task in terms of reverse engineering. To mitigate the obstacle of understating the inner behavior of Java, it was proposed to use source code with structural and behavioral aspects of the program with UML diagrams, specifically object diagram (J. Yang, Lee, Gandhi, & Valli, 2017).

5.5 Evaluation Summary for the Three Experiments Configuration of the AETFS

In the evaluation summary presented in the three AETFS configurations, a technique called AETFS was applied to improve the DFLT's process. The initial evaluation of P, R, and F-measure (F) was conducted without utilizing AETFS, using the DFLT's approach. The obtained results provided a baseline for comparison. However, subsequent evaluations were performed after implementing DFLT's with AETFS. This integration of AETFS significantly influenced the results, leading to improvements in P, R, and overall F-measure. Table 5.15 presents the comparative evaluation results for the case study, highlighting the impact of incorporating AETFS into the DFLT's.

Table 5.15: Comparison of Case Study Evaluation for Three AETFS Configuration

AETFS Configuration	P (without AETFS)	R (without AETFS)	F (without AETFS)
Experiment Evaluation Number 5.2	75%	87%	80%
Experiment Evaluation Number 5.3	34%	100%	49%
Experiment Evaluation Number 5.4	75%	87%	80%
AETFS Configuration	P (with AETFS)	R (with AETFS)	F (with AETFS)
Experiment Evaluation Number 5.2	84%	97%	90%
Experiment Evaluation Number 5.3	48%	100%	65%
Experiment Evaluation Number 5.4	87%	100%	92%

The Table 5.15 provided offers a detailed comparison of case study evaluations for three distinct AETFS configurations. Each P), R, and F-Measure score presented represents an average result derived from the specified Experiment Evaluation Number.

For example, in Experiment Evaluation Number 5.2, the P is recorded at 75% without utilizing AETFS, whereas the P score for the same evaluation stands at 84%. This P value is derived as the average of the P values from Table 5.6 across the six datasets, both with and without the application of MappFL. Similarly, the averages for R and F-Measure are computed from Table 5.6, and the results average values are displayed in Table 5.15. The subsequent three paragraphs delve into a detailed explanation of the findings presented in Table 5.15.

In the first Experiment Evaluation Number 5.2, the AETFS configuration demonstrates improvements across all metrics when compared to the scenario without AETFS. Precision increases from 75% to 84%, Recall from 87% to 97%, and the F-Measure from 80% to 90%. This significant enhancement underscores the positive impact of integrating AETFS in this particular setting.

Moving to the second Experiment Evaluation Number 5.3, the introduction of AETFS leads to varied outcomes. While Precision and F-Measure experience slight improvements, with values rising from 34% to 48% and from 49% to 65%, respectively, the Recall metric remains unchanged at 100%. This suggests a specific aspect where further optimization may be necessary.

In the context of the third Experiment Evaluation Number 5.4, the AETFS configuration showcases notable advancements across all metrics. Precision increases from 75% to 87%, Recall remains at a perfect 100%, and the F-Measure sees a significant rise from 80% to 92%. These results highlight the substantial benefits of incorporating AETFS in this specific configuration,

emphasizing its effectiveness in optimizing system evaluation processes.

These results highlight the value of incorporating AETFS into the feature location process. The utilization of AETFS significantly improved the precision, recall, and F-measure, leading to more accurate and comprehensive identification of relevant features. This suggests that AETFS can effectively assist in automating the extraction of testable feature scenarios and enhance the overall efficiency and effectiveness of the feature location process.

In other words, The line chart presents in Figure 5.13 provides a scientific analysis of the performance comparison for three AETFS configurations. The x-axis represents the different AETFS configurations, labeled as Exp 5.2, Exp 5.3, and Exp 5.4. The y-axis represents the performance metric, with higher values indicating better performance.

Analyzing the chart, we observe two lines: a blue line representing performance without AETFS and a red line representing performance with AETFS.

For Exp 5.2, the blue line shows a performance of 75%, while the red line demonstrates an improvement to 84%. This indicates that the AETFS configuration leads to a performance increase of 9% in this particular scenario.

Moving to Exp 5.3, the blue line shows a performance of 34%, while the red line exhibits a substantial improvement to 48%. This represents a significant performance increase of 14% when using the AETFS configuration.

In Exp 5.4, the blue line displays a performance of 75%, while the red line demonstrates an improvement to 87%. This signifies a notable performance increase of 12% when applying the AETFS configuration.

Overall, the line chart analysis illustrates that the AETFS configuration consistently improves performance across all three evaluated sections. The

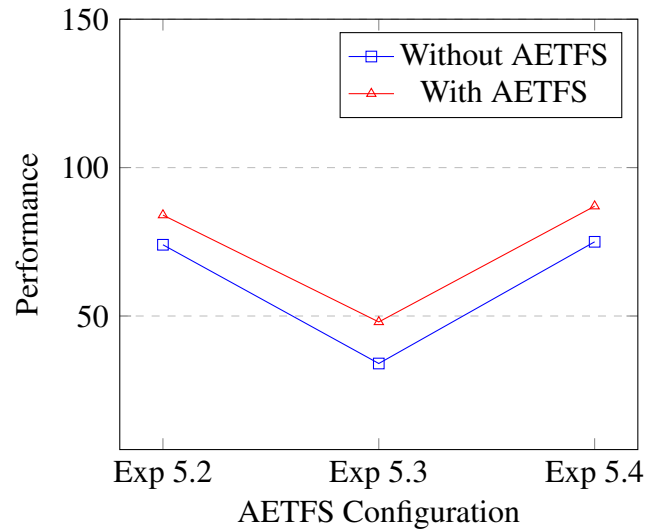


Figure 5.13: Comparison of Case Study Evaluation for Three AETFS Configurations

performance gains range from 9% to 39%, depending on the specific section considered. These findings provide scientific evidence for the effectiveness of AETFS in enhancing the performance of the system under evaluation.

Researchers and practitioners can utilize this information to make informed decisions about the adoption of AETFS configurations in similar scenarios, where improved performance is desired.

Chapter 6

Conclusions and Future Research Work

This study introduces a novel method, AETFS, which leverages the content of execution traces to refine the input formulation necessary for exercising software features, thereby enhancing the precision of DFLT. The ideal configuration for this method was identified, and its effectiveness was benchmarked against leading IR techniques such as LSI, LDA, VSM, and UML, which serve as the baseline for DFLT. Ultimately, the performance of AETFS in executing DFLT was thoroughly evaluated.

The ideal setup for AETFS was determined through empirical research to involve application at the method level, including variables, classes, and objects, and it employs bytecode instrumentation within the JVM. Additionally, this configuration incorporates a filtering process for utility data sets that are called upon during execution.

This research highlights the importance of defining inputs for DFLT to optimize configurations. Variations in AETFS configurations showed unpredictable performance differences. Pre-characterizing analyzed systems

can aid in selecting appropriate input scenarios. While past literature often overlooks DFLT configuration and input details, this study suggests prioritizing them in future research.

This chapter is organized as follows: it begins by listing the contributions made by this study; it then provides responses to the initial research questions posed; it examines potential limitations and their impact on the study's validity; it outlines the significance of the findings for both researchers and practitioners; and it concludes with a dialogue on prospective research directions.

6.1 Contributions

This work's unique additions to the current body of knowledge encompass:

- The study introduced and implemented a new textual filtering method for execution traces, known as AETFS, designed to address the challenge of limited vocabulary within execution trace content. This method is grounded in sophisticated (IR) techniques, including LSI, LDA, and VSM, and is fine-tuned for:
 - Selecting the most efficient way to manage tracing the invoked data (Bytecode Instrumentation)
 - Refining the textual query that delineates the software feature in use to augment the capabilities of the Information Retrieval system.
 - Applying sophisticated Information Retrieval models like LSI, LDA, and VSM to filter the gathered execution trace data.
- The conclusion that the choice of inputs for DFLT's should be influenced by the attributes of the system, and the identification of several of these attributes pertinent to AETFS.

- The assessment of this method demonstrates its competitiveness with contemporary, cutting-edge DFLTs.
- An integrated method that combines execution trace with source code analysis, employing UML visualization techniques to improve AETFS's precision in choosing the correct inputs for DFLTs.

6.2 Addressing the Study's Inquiries

RQ1: How can we effectively formulate scenarios or test cases to exercise specific software features in DFLTs?

The AETFS method, as outlined in Chapter 3, provides a solution to this question within the context of DFLTs. This method enables the aggregation of execution traces collected when a software feature is demonstrated by a scenario. This aggregated data provides developers with feedback, helping them identify which executed scenarios were most effective in implementing DFLTs.

When combined with IR, it facilitates the indexing and searching of necessary entities based on a search query. The inputs for AETFS are scenarios or test cases and a text query that describes the feature at a high level.

AETFS depends on instrumentation techniques used to generate execution traces. These traces are collected as documents for further processing in subsequent stages, with each document representing a segment of a scenario.

RQ2: What impact does the accuracy of scenario formulation have on the performance of DFLTs?

The empirical study in Chapter 5 provides an answer to this research question. Execution traces that are generated can be grouped in various unique ways to tackle the issues of the large size and significant noise in the collected execution trace. This includes considerations of granularity,

the volume of the execution trace collected from a single scenario, and the filtering of the collected execution trace content. The residual properties in the execution trace could potentially influence the success of scenario reformulation.

The initial discovery in response to this research question is that the use of method-level granularity proved more effective for AETFS than file-level granularity. This was particularly true when identifying entities that could be utilized in subsequent stages to ascertain the accuracy of the scenario. This finding aligns with prior research on the impact of granularity on input scenarios in DFLTs (Hayashi et al., 2016). It appears that both DFLTs in general, and AETFS in particular, are more optimally utilized at the method level of granularity.

An additional intriguing outcome of this study indicates that the incorporation of any form of instrumentation technique adversely affects the efficiency of AETFS at the method level granularity. A robust, positive, and statistically significant correlation was noted with the use of bytecode instrumentation technique (JVM Java Agent) across all subject systems. This could potentially suggest that employing the appropriate instrumentation technique may reduce the size of the collected execution traces.

Interestingly, the performance of DFLTs improved significantly when the content of the execution traces was filtered using text classification based on text query input. In fact, DFLTs integrated with AETFS showed a 12% improvement in terms of Precision, Recall, and F-Measure compared to DFLTs without the use of AETFS. Future research could explore whether more advanced classification techniques could further enhance this performance.

Through a systematic comparison and elimination of the least effective configurations, it was determined that the best response to RQ2 involves segmenting the execution traces and applying a filtered method-level

AETFS configuration. This configuration, which includes both possible configurations, yielded the most favorable results. Specifically, the configuration involving text classification and scenario segmentation proved to be significantly more efficient than the others.

RQ3: How can execution traces be optimally utilized in the process of feature location when using accurately formulated scenarios or test cases?

For RQ2, we identified the most effective configuration of AETFS within DFLT's and compared it with a standard DFLT's approach. A distinct split was observed among the subject systems, particularly when DFLT's and AETFS were implemented. The AETFS method outperformed in three out of eight systems, while the DFLT's baseline proved superior for the remaining four systems. Therefore, in response to RQ3, we observed varied results: AETFS was more effective for half of the studied software systems and less so for the other half.

RQ4: What are the challenges and potential solutions in formulating scenarios or test cases for DFLT's in large and complex software systems?

A significant challenge in AETFS was to identify DFLT's that could be uniformly applied across all configurations of AETFS. To ensure a fair evaluation of AETFS, various metrics of the subject systems were examined. One metric that appeared to be relevant was the size of the scenario required to exhibit the targeted feature, F. Systems that required fewer scenario segments to exercise the software feature responded more favorably to AETFS. Another less obvious characteristic to consider is the age of the software system. Younger systems seemed to respond better to AETFS. However, these preliminary findings require further empirical evaluation.

RQ5: How can we measure the effectiveness and efficiency of different scenario formulation strategies in DFLT's?

Considering the variety of systems identified in RQ3 and RQ4, we evaluated the combination of AETFS and the DFLT's baseline. We discovered that integrating DFLT's with AETFS significantly enhances performance by up to 45%. In essence, the assessment of AETFS was conducted by comparing the performance of DFLT's with and without the use of AETFS. There are several existing DFLT's techniques (Cornelissen et al., 2009; Dit, Holtzhauer, et al., 2013; Rubin & Chechik, 2013; Buzaid & Hammad, 2021b), and all of them require scenarios or test cases. The performance of these DFLT's was evaluated both with and without the use of AETFS. The results of these evaluations are detailed in Chapter 5.

In conclusion, and to address the primary research question, “Does careful formulation of input scenarios enhance the efficiency of DFLT's in mapping software features to source code?”:

- The proposed approach, that leverages the collected execution trace (AETFS), is effective.
- There are multiple configurations of AETFS at the method level. Segmenting the scenario to generate several execution traces and including only the most pertinent one improves the effectiveness of DFLT's.
- This configuration AETFS seems to work for DFLT's in systems that have been developed by Java.
- When used in conjunction with DFLT's, AETFS produces more effective outcomes than DFLT's used independently. However, it's important to note that AETFS on its own is not a solution for associating source code with the corresponding feature.

6.3 Conclusion

Over more than ten years of DFLT's research, a considerable number of FLT's have been produced: 60 FLT papers were pinpointed by (Dit, Revelle, Gethers, & Poshyvanyk, 2013), and an additional 27 were identified as part of this study's literature review. Presently, the focus of DFLT's research is on those that incorporate a blend of other methods (over half of the papers in the literature describe such approaches). However, these methods have seldom been analyzed to determine the most effective configurations and have been uniformly applied to all software systems, following the "one-size-fits-all" paradigm. The need for more refined tuning of FLT's has been recently acknowledged (Rubin & Chechik, 2013; Sulír & Porubán, 2017a), and this should become an increasing trend in the field. The goal should be to examine existing DFLT's to evaluate whether the configuration of other cutting-edge techniques in the field influences effectiveness.

The use of instrumentation techniques to generate execution traces has proven to be highly effective in dynamic feature location. These techniques provide valuable insights into the runtime behavior of software systems, enabling more accurate mapping of features to source code. However, there is still room for improvement, particularly in the area of bytecode instrumentation (Soueidi, Monnier, & Falcone, 2023). Future work could focus on developing more sophisticated bytecode instrumentation techniques to enhance the precision and efficiency of dynamic feature location. This could involve exploring different strategies for bytecode modification, optimizing the process of trace generation, and investigating how to handle complex scenarios such as multithreading or asynchronous operations (Troelsen, Japikse, Troelsen, & Japikse, 2020). Ultimately, the goal is to continue advancing the field of dynamic feature location to better support software maintenance and evolution.

This study uncovered an intriguing discovery that points to a new, meta-

level DFLT. It was observed that various subject systems have different responses to scenarios and DFLT based on collected execution traces. Consequently, future research should aim to pinpoint the characteristics of software systems that align with specific DFLT. Indeed, there are other characteristics, not related to source code, that might influence the choice of different DFLT. For instance, in TFLT, the quantity of source code tests (which could serve as scenarios) and the extent of source code coverage by these tests could be considered as one of the characteristics. The quality of the text query is crucial for TFLT (Refer to Chapter 2).

The integration of the AETFS solution in the formulation of accurate scenarios or test cases has shown to significantly enhance the performance of dynamic feature location techniques. AETFS, with its ability to adaptively trace features, provides a more refined path through the software system, leading to a more precise mapping of features to their corresponding source code.

Accurate scenarios, when combined with AETFS, illuminate the specific sequences of events and interactions that activate a particular feature. This allows dynamic feature location techniques to trace these paths back to the relevant sections of source code, thereby identifying the code responsible for implementing the feature.

Test cases, when enhanced with AETFS, provide a benchmark against which the actual behavior of the software can be compared. This comparison can reveal discrepancies between the intended and actual functionality, guiding the feature location process towards the problematic areas of code.

However, the effectiveness of these techniques is not without its challenges. The complexity of modern software systems, coupled with the inherent variability in user behavior, can make it difficult to create scenarios and test cases that accurately represent all possible paths through the system. This is where AETFS comes into play, adapting to these complexities

and providing more accurate traces.

Despite these challenges, the benefits of using AETFS in conjunction with accurate scenarios and test cases in dynamic feature location techniques are clear. They provide a structured and systematic approach to feature location, reducing the time and effort required to understand complex software systems.

Looking ahead, there is considerable scope for further research and development in this area. Future work could explore new methods for generating and refining scenarios and test cases, leveraging advances in areas such as machine learning and artificial intelligence. Additionally, research could investigate ways to integrate dynamic feature location techniques more closely with other software development practices, enhancing their utility and applicability.

In conclusion, the use of AETFS in conjunction with accurate scenarios and test cases in dynamic feature location techniques represents a powerful tool for understanding and maintaining software systems. While challenges remain, the potential of these techniques to improve the efficiency and effectiveness of feature location is significant. As research continues to advance in this field, it is anticipated that these techniques will play an increasingly important role in the future of software development.

6.4 Future Work

The field of dynamic feature location in source code is ripe for future exploration and development. One potential area of research could be the development of more sophisticated algorithms for dynamic feature location. These algorithms could leverage advances in machine learning and artificial intelligence to improve the accuracy and efficiency of feature location. For example, machine learning models could be trained on large datasets

of software systems to learn patterns and correlations that can aid in feature location. These models could then be used to predict the location of features in new or modified software systems, reducing the time and effort required for manual feature location.

Another promising direction for future research is the integration of dynamic feature location techniques with other software development practices. For instance, dynamic feature location could be incorporated into the software testing process to help identify the parts of the source code that are associated with specific features. This could improve the effectiveness of testing by ensuring that all relevant parts of the code are adequately tested. Similarly, dynamic feature location could be used in the context of software maintenance and evolution to help developers understand the impact of their changes on the functionality of the software.

Moreover, future research could also focus on improving the scalability of dynamic feature location techniques. As software systems continue to grow in size and complexity, it becomes increasingly challenging to locate features in the source code. New methods and tools could be developed to handle large-scale software systems, making it easier to locate features in these systems. This could involve the development of distributed or parallel algorithms for feature location, or the use of cloud computing resources to perform feature location on a large scale. Overall, the field of dynamic feature location in source code offers many exciting opportunities for future research and development.

Finally, in the forthcoming stages of this research, a more detailed exploration and analysis will be dedicated to addressing the research questions outlined above (RQs 6 to 10). These inquiries are designed to provide a comprehensive understanding of the intricacies involved in improving Dynamic Feature Location Techniques (DFLTs) by focusing on the development of tools and techniques to enhance scenario formulation accuracy

(RQ6), ensuring comprehensive coverage of exercised features in formulated scenarios (RQ7), incorporating developer expertise into the DFLT process (RQ8), leveraging ML or AI for scenario formulation enhancement (RQ9), and standardizing best practices for scenario formulation across projects and teams (RQ10). Each of these research questions will be explored in more depth, utilizing specific metrics to assess effectiveness, usability, coverage, expertise impact, automation levels, consistency, and quality of scenarios formulated in the context of DFLTs.

References

- Abbasli, N., & Ganiz, M. C. (2021). Log and execution trace analytics system. In *2021 international conference on innovations in intelligent systems and applications (inista)* (pp. 1–7).
- Agbele, K. K., Adesina, A. O., Azeez, N. A., & Abidoye, A. P. (2012). Context-aware stemming algorithm for semantically related root words.
- Alhindawi, N., Dragan, N., Collard, M. L., & Maletic, J. I. (2013). Improving feature location by enhancing source code with stereotypes. In *2013 ieee international conference on software maintenance* (pp. 300–309).
- Ali, K., Lai, X., Luo, Z., Lhoták, O., Dolby, J., & Tip, F. (2019). A study of call graph construction for jvm-hosted languages. *IEEE transactions on software engineering*, 47(12), 2644–2666.
- Allan, C., Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, O., ... Tibble, J. (2005). Adding trace matching with free variables to aspectj. *ACM SIGPLAN Notices*, 40(10), 345–364.
- Almulla, H., & Gay, G. (2022). Learning how to search: generating effective test cases through adaptive fitness function selection. *Empirical Software Engineering*, 27(2), 38.
- Al-Sharif, Z. A., Abdalrahman, W. F., & Jeffery, C. L. (2021). Encoding test cases using execution traces. In *2021 12th international conference on information and communication systems (icics)* (pp. 239–244).

- ANDAM, B. K. (2018). Migrating and evolving software product lines: An industrial case study of feature location and visualization techniques.
- Antoniol, G., & Guéhéneuc, Y.-G. (2006). Feature identification: An epidemiological metaphor. *IEEE Transactions on Software Engineering*, 32(9), 627–641.
- Arcega, L., Font, J., Haugen, Ø., & Cetina, C. (2017). On the influence of models at run-time traces in dynamic feature location. In *European conference on modelling foundations and applications* (pp. 90–105).
- ArogUML. (2020). <http://argouml.tigris.org> [Computer software manual]. Retrieved from <http://argouml.tigris.org>
- Arwan, A., Rochimah, S., & Fatichah, C. (2022). A comparison study: The effect of nouns and verbs in finding feature location. In *Proceedings of the 7th international conference on sustainable information engineering and technology* (pp. 310–315).
- Arwan, A., Rochimah, S., & Fatichah, C. (2023). Query expansion using novel use case scenario relationship for finding feature location. *International Journal of Electrical & Computer Engineering* (2088-8708), 13(5).
- Asadi, F., Antoniol, G., & Guéhéneuc, Y.-G. (2010). Concept location with genetic algorithms: A comparison of four distributed architectures. In *2nd international symposium on search based software engineering* (pp. 153–162).
- Belmonte, J., & Dugerdil, P. (2018). Program understanding using ontologies and dynamic analysis. In *Proceedings of the 33rd annual acm symposium on applied computing* (pp. 1552–1559).
- Bennett, C., Myers, D., Storey, M.-A., German, D. M., Ouellet, D., Salois, M., & Charland, P. (2008). A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4),

291–315.

- Beranič, T., Podgorelec, V., & Heričko, M. (2018). Towards a reliable identification of deficient code with a combination of software metrics. *Applied Sciences*, 8(10), 1902.
- Béziers la Fosse, T., Tisi, M., & Mottu, J.-M. (2018). Injecting execution traces into a model-driven framework for program analysis. In *Software technologies: Applications and foundations: Staf 2017 collocated workshops, marburg, germany, july 17-21, 2017, revised selected papers* (pp. 3–13).
- Biggers, L. R., Bocovich, C., Capshaw, R., Eddy, B. P., Etzkorn, L. H., & Kraft, N. A. (2014). Configuring latent dirichlet allocation based feature location. *Empirical Software Engineering*, 19(3), 465–500.
- Binder, W., Hulaas, J., & Moret, P. (2007). Advanced java bytecode instrumentation. In *Proceedings of the 5th international symposium on principles and practice of programming in java* (pp. 135–144).
- Binkley, D., Lawrie, D., Uehlinger, C., & Heinz, D. (2015). Enabling improved ir-based feature location. *Journal of Systems and Software*, 101, 30–42.
- Bittner, P. M., Schultheiß, A., Thüm, T., Kehrer, T., Young, J. M., & Linsbauer, L. (2021). Feature trace recording. In *Proceedings of the 29th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (pp. 1007–1020).
- Bittner, P. M., Schultheiß, A., Thüm, T., Kehrer, T., Young, J. M., & Linsbauer, L. (2022). Feature trace recording-summary. *Software Engineering 2022*.
- Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent dirichlet allocation. *the Journal of machine Learning research*, 3, 993–1022.
- Bogatinovski, J., & Kao, O. (2023). Auto-logging: Ai-centred logging

- instrumentation. In *2023 ieee/acm 45th international conference on software engineering: New ideas and emerging results (icse-nier)* (pp. 95–100).
- Bousse, E., Mayerhofer, T., Combemale, B., & Baudry, B. (2019). Advanced and efficient execution trace management for executable domain-specific modeling languages. *Software & Systems Modeling*, 18, 385–421.
- Braun, E., Amyot, D., & Lethbridge, T. C. (2015). Generating software documentation in use case maps from filtered execution traces. In *International sdl forum* (pp. 177–192).
- Briand, L. C., Labiche, Y., & Leduc, J. (2006). Toward the reverse engineering of uml sequence diagrams for distributed java software. *IEEE Transactions on Software Engineering*, 32(9), 642–663.
- Buzaid, F., Albalooshi, F., & Elmedany, W. (2022). The use of uml diagrams to enhance dynamic feature location techniques. In *2022 international conference on innovation and intelligence for informatics, computing, and technologies (3ict)* (pp. 285–292).
- Buzaid, F., Albalooshi, F., & Hammad, M. (2023). Scenario sifter to enhance the performance of dynamic feature locations techniques. In *International journal of computing and digital systems (ijcds)* (pp. 285–292).
- Buzaid, F., Albalooshi, F., & Rouached, M. (2022). Utilizing the output of azure iot hub to comprehend the software functionalities. In *6th smart cities symposium (scs 2022)* (Vol. 2022, pp. 258–263).
- Buzaid, F., & Hammad, M. (2021a). Mapping features to source code based on execution trace. In *4th smart cities symposium (scs 2021)* (Vol. 2021, pp. 72–77).
- Buzaid, F., & Hammad, M. (2021b). Reviewing dynamic feature location techniques: Basic elements and challenges. In *2021 international*

- conference on innovation and intelligence for informatics, computing, and technologies (3ict)* (pp. 242–249).
- Cai, Z., Yang, X., Wang, X., & Kavs, A. J. (2012). Feature location in source code by trace-based impact analysis and information retrieval. *IEICE TRANSACTIONS on Information and Systems*, 95(1), 205–214.
- Caldiera, V. R. B. G., & Rombach, H. D. (1994). The goal question metric approach. *Encyclopedia of software engineering*, 528–532.
- Cassar, I., Francalanza, A., & Said, S. (2015). Improving runtime overheads for detector. *arXiv preprint arXiv:1503.04912*.
- Chapman, C., & Stolee, K. T. (2016). Exploring regular expression usage and context in python. In *Proceedings of the 25th international symposium on software testing and analysis* (pp. 282–293).
- Chen, B., & Jiang, Z. M. (2021). A survey of software log instrumentation. *ACM Computing Surveys (CSUR)*, 54(4), 1–34.
- Chochlov, M., English, M., & Buckley, J. (2017). A historical, textual analysis approach to feature location. *Information and Software Technology*, 88, 110–126.
- Collard, M. L., Decker, M. J., & Maletic, J. I. (2013). srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 ieee international conference on software maintenance* (pp. 516–519).
- Collard, M. L., & Maletic, J. I. (2016). srcml 1.0: Explore, analyze, and manipulate source code. In *Icsme* (p. 649).
- Copestake, A. (2004). Natural language processing. *University of Cambridge*.
- Corley, C. S., Damevski, K., & Kraft, N. A. (2015). Exploring the use of deep learning for feature location. In *2015 ieee international conference on software maintenance and evolution (icsme)* (pp. 556–560).
- Cornelissen, B., Zaidman, A., Holten, D., Moonen, L., Van Deursen, A.,

- & Van Wijk, J. J. (2008). Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software*, 81(12), 2252–2268.
- Cornelissen, B., Zaidman, A., Van Deursen, A., Moonen, L., & Koschke, R. (2009). A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5), 684–702.
- Cruz, D., Figueiredo, E., & Martinez, J. (2019). A literature review and comparison of three feature location techniques using argouml-spl. In *Proceedings of the 13th international workshop on variability modelling of software-intensive systems* (pp. 1–10).
- Dadachev, B., Balinsky, A., Balinsky, H., & Simske, S. (2012). On the helmholtz principle for data mining. In *2012 third international conference on emerging security technologies* (pp. 99–102).
- de Almeida Maia, M., & Lafetá, R. F. (2013). On the impact of trace-based feature location in the performance of software maintainers. *Journal of Systems and Software*, 86(4), 1023–1037.
- de Carvalho, T. D. R. (2019). Programming and mapping strategies for embedded computing runtime adaptability.
- DeLozier, G. S., Decker, M. J., Newman, C. D., & Maletic, J. I. (2018). Leveraging the agile development process for selecting invoking/excluding tests to support feature location. In *Proceedings of the 26th conference on program comprehension* (pp. 370–379).
- Desnos, A. (2012). Android: Static analysis using similarity distance. In *2012 45th hawaii international conference on system sciences* (pp. 5394–5403).
- Dhamija, A., & Sikka, S. (2019). A systematic review of feature location techniques under software change impact analysis. *Int. J. Comput. Sci. Eng*, 7(3), 184–192.

- Di Martino, S., Fasolino, A. R., Starace, L. L. L., & Tramontana, P. (2021). Comparing the effectiveness of capture and replay against automatic input generation for android graphical user interface testing. *Software Testing, Verification and Reliability*, 31(3), e1754.
- Dit, B., Guerrouj, L., Poshyvanyk, D., & Antoniol, G. (2011). Can better identifier splitting techniques help feature location? In *2011 ieee 19th international conference on program comprehension* (pp. 11–20).
- Dit, B., Holtzhauer, A., Poshyvanyk, D., & Kagdi, H. (2013). A dataset from change history to support evaluation of software maintenance tasks. In *2013 10th working conference on mining software repositories (msr)* (pp. 131–134).
- Dit, B., Revelle, M., Gethers, M., & Poshyvanyk, D. (2013). Feature location in source code: a taxonomy and survey. *Journal of software: Evolution and Process*, 25(1), 53–95.
- Dit, B., Revelle, M., & Poshyvanyk, D. (2013). Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software Engineering*, 18(2), 277–309.
- Eaddy, M., Aho, A. V., Antoniol, G., & Guéhéneuc, Y.-G. (2008). Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *2008 16th ieee international conference on program comprehension* (pp. 53–62).
- Eghbal, N. (2020). *Working in public: the making and maintenance of open source software*. Stripe Press.
- Eisenberg, A. D., & De Volder, K. (2005). Dynamic feature traces: Finding features in unfamiliar code. In *21st ieee international conference on software maintenance (icsm'05)* (pp. 337–346).
- Elallaoui, M., Nafil, K., & Touahni, R. (2018). Automatic transformation of user stories into uml use case diagrams using nlp techniques. *Procedia*

computer science, 130, 42–49.

- Eyal Salman, H., Seriai, A.-D., & Dony, C. (2022). Feature location in software variants toward software product line engineering. In *Handbook of re-engineering software intensive systems into software product lines* (pp. 3–30). Springer.
- Eyal-Salman, H., Seriai, A.-D., & Hammad, M. (2016). Quality-driven feature identification and documentation from source code. *Journal of Theoretical and Applied Information Technology*, 84(2), 183–195.
- Flageol, W., Menaud, E., Guéhéneuc, Y.-G., Badri, M., & Monnier, S. (2023). A mapping study of language features improving object-oriented design patterns. *Information and Software Technology*, 107222.
- Florez, J. M., Perry, J., Wei, S., & Marcus, A. (2022). Retrieving data constraint implementations using fine-grained code patterns. In *Proceedings of the 44th international conference on software engineering* (pp. 1893–1905).
- Foundation, A. S. (2011). Apache lucene - scoring.. Retrieved from <http://lucene.apache.org/java/340/scoring.html> (letzter Zugriff: 20. Oktober 2011)
- Fujioka, D., & Nitta, N. (2017). Constraints based approach to interactive feature location. In *2017 ieee international conference on software maintenance and evolution (icsme)* (pp. 499–503).
- García Holgado, A., Marcos Pablos, S., García Peñalvo, F. J., et al. (2020). Guidelines for performing systematic research projects reviews. *International Journal of Interactive Multimedia and Artificial Intelligence*, 6(2), 9.
- Gay, G., Haiduc, S., Marcus, A., & Menzies, T. (2009). On the use of relevance feedback in ir-based concept location. In *2009 ieee international conference on software maintenance* (pp. 351–360).

- Ghaleb, T. A., Alturki, M. A., & Aljasser, K. (2018). Program comprehension through reverse-engineered sequence diagrams: A systematic review. *Journal of Software: Evolution and Process*, 30(11), e1965.
- Gosain, A., & Sharma, G. (2015). A survey of dynamic program analysis techniques and tools. In *Proceedings of the 3rd international conference on frontiers of intelligent computing: Theory and applications (ficta) 2014* (pp. 113–122).
- Haiduc, S., Bavota, G., Marcus, A., Oliveto, R., De Lucia, A., & Menzies, T. (2013). Automatic query reformulations for text retrieval in software engineering. In *2013 35th international conference on software engineering (icse)* (pp. 842–851).
- Hammad, M., & Al-Hawawreh, M. (2018). Generating sequence diagram and call graph using source code instrumentation. In *Information technology-new generations* (pp. 641–645). Springer.
- Hammoudi, M., Mayr-Dorn, C., Mashkoor, A., & Egyed, A. (2021). On the effect of incompleteness to check requirement-to-method traces. In *Proceedings of the 36th annual acm symposium on applied computing* (pp. 1465–1474).
- Hayashi, S., Kazato, H., Kobayashi, T., Oshima, T., Natsukawa, K., Hoshino, T., & Saeki, M. (2016). Guiding identification of missing scenarios for dynamic feature location. In *2016 23rd asia-pacific software engineering conference (apsec)* (pp. 393–396).
- Henderson, T. A., & Podgurski, A. (2018). Behavioral fault localization by sampling suspicious dynamic control flow subgraphs. In *2018 ieee 11th international conference on software testing, verification and validation (icst)* (pp. 93–104).
- Hendriks, M., Verriet, J., Basten, T., Theelen, B., Brassé, M., & Somers, L. (2017). Analyzing execution traces: critical-path analysis and distance analysis. *International Journal on Software Tools for Tech-*

- nology Transfer*, 19, 487–510.
- Hilsdale, E., & Hugunin, J. (2004). Advice weaving in aspectj. In *Proceedings of the 3rd international conference on aspect-oriented software development* (pp. 26–35).
- Hojaji, F., Mayerhofer, T., Zamani, B., Hamou-Lhadj, A., & Bousse, E. (2019). Model execution tracing: a systematic mapping study. *Software and Systems Modeling*, 18, 3461–3485.
- Horváth, F., Gergely, T., Beszédes, Á., Tengeri, D., Balogh, G., & Gyimóthy, T. (2019). Code coverage differences of java bytecode and source code instrumentation tools. *Software Quality Journal*, 27(1), 79–123.
- Huh, S. M., & Kim, W.-J. (2021). Locating core modules through the association between software source structure and execution. *Applied Sciences*, 11(4), 1685.
- Jaspan, C., Jorde, M., Knight, A., Sadowski, C., Smith, E. K., Winter, C., & Murphy-Hill, E. (2018). Advantages and disadvantages of a monolithic repository: a case study at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice* (pp. 225–234).
- Kanya, N., & Geetha, S. (2007). Information extraction-a text mining approach.
- Kästner, C., Dreiling, A., & Ostermann, K. (2013). Variability mining: Consistent semi-automatic detection of product-line features. *IEEE Transactions on Software Engineering*, 40(1), 67–82.
- Kazato, H., Hayashi, S., Kobayashi, T., Oshima, T., Okada, S., Miyata, S., ... Saekii, M. (2013). Incremental feature location and identification in source code. In *2013 17th european conference on software maintenance and reengineering* (pp. 371–374).
- Kazato, H., Hayashi, S., Oshima, T., Miyata, S., Hoshino, T., & Saeki,

- M. (2013). Extracting and visualizing implementation structure of features. In *2013 20th asia-pacific software engineering conference (apsec)* (Vol. 1, pp. 476–484).
- Keenan, E., Czauderna, A., Leach, G., Cleland-Huang, J., Shin, Y., Moritz, E., ... others (2012). Tracelab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In *2012 34th international conference on software engineering (icse)* (pp. 1375–1378).
- Khatiwada, S., Kelly, M., & Mahmoud, A. (2016). Stac: A tool for static textual analysis of code. In *2016 ieee 24th international conference on program comprehension (icpc)* (pp. 1–3).
- Khoury, R., Shi, L., & Hamou-Lhadj, A. (2016). Key elements extraction and traces comprehension using gestalt theory and the helmholtz principle. In *2016 ieee international conference on software maintenance and evolution (icsme)* (pp. 478–482).
- Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., & Griswold, W. G. (2001). An overview of aspectj. In *European conference on object-oriented programming* (pp. 327–354).
- Kim, J., Batory, D., & Dig, D. (2017). Refactoring java software product lines. In *Proceedings of the 21st international systems and software product line conference-volume a* (pp. 59–68).
- Koschke, R., & Quante, J. (2005). On dynamic feature location. In *Proceedings of the 20th ieee/acm international conference on automated software engineering* (pp. 86–95).
- Kuang, H., Mäder, P., Hu, H., Ghabi, A., Huang, L., Lü, J., & Egyed, A. (2015). Can method data dependencies support the assessment of traceability between requirements and source code? *Journal of Software: Evolution and Process*, 27(11), 838–866.
- Lau, K.-K., & Arshad, R. (2016). A concise classification of reverse engi-

- neering approaches for software product lines. In *International journal on advances in software: Icsea 2016: The eleventh international conference on software engineering advances*.
- Le, Q., & Mikolov, T. (2014). Distributed representations of sentences and documents. In *International conference on machine learning* (pp. 1188–1196).
- Le, T.-D. B., Oentaryo, R. J., & Lo, D. (2015). Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering* (pp. 579–590).
- Lee, J. H., & Ostwald, M. J. (2024). Latent dirichlet allocation (lda) topic models for space syntax studies on spatial experience. *City, Territory and Architecture*, 11(1), 3.
- Lessa, D., Czyz, J. K., & Jayaraman, B. (2010). Jive: A pedagogic tool for visualizing the execution of java programs. *Bericht, Univ. of New York, Buffalo*.
- Li, C.-T., Cao, J., Ma, C., Shen, J., & Wong, K. H. (2022). An agnostic and efficient approach to identifying features from execution traces. *Knowledge-Based Systems*, 250, 108988.
- Liu, D., Marcus, A., Poshyvanyk, D., & Rajlich, V. (2007). Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the twenty-second ieee/acm international conference on automated software engineering* (pp. 234–243).
- Lu, X. A., & Keefer, R. B. (1995). Query expansion/reduction and its impact on retrieval effectiveness. *NIST SPECIAL PUBLICATION SP*, 231–231.
- Maalej, W., Tiarks, R., Roehm, T., & Koschke, R. (2014). On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4), 1–37.

- Marcus, A., & Haiduc, S. (2013). Text retrieval approaches for concept location in source code. *Software Engineering: International Summer Schools, ISSSE 2009-2011, Salerno, Italy. Revised Tutorial Lectures*, 126–158.
- Marcus, A., Sergeyev, A., Rajlich, V., & Maletic, J. I. (2004). An information retrieval approach to concept location in source code. In *11th working conference on reverse engineering* (pp. 214–223).
- Márquez, G., Astudillo, H., & Kazman, R. (2023). Architectural tactics in software architecture: A systematic mapping study. *Journal of Systems and Software*, 197, 111558.
- Martinez, J., Ordoñez, N., Tërnavá, X., Ziadi, T., Aponte, J., Figueiredo, E., & Valente, M. T. (2018). Feature location benchmark with argouml spl. In *Proceedings of the 22nd international systems and software product line conference-volume 1* (pp. 257–263).
- Memane, R. (2020). Event listener in java.
- Michelon, G. K., Linsbauer, L., Assunção, W. K., Fischer, S., & Egyed, A. (2021). A hybrid feature location technique for re-engineering single systems into software product lines. In *15th international working conference on variability modelling of software-intensive systems* (pp. 1–9).
- Michelon, G. K., Obermann, D., Assunção, W. K., Linsbauer, L., Grünbacher, P., & Egyed, A. (2021). Managing systems evolving in space and time: four challenges for maintenance, evolution and composition of variants. In *Proceedings of the 25th acm international systems and software product line conference-volume a* (pp. 75–80).
- Michelon, G. K., Obermann, D., Assunção, W. K., Linsbauer, L., Grünbacher, P., Fischer, S., ... Egyed, A. (2022). Evolving software system families in space and time with feature revisions. *Empirical Software Engineering*, 27(5), 112.

- Miller, G. A. (1995). Wordnet: a lexical database for english. *Communications of the ACM*, 38(11), 39–41.
- Mills, C., Bavota, G., Haiduc, S., Oliveto, R., Marcus, A., & Lucia, A. D. (2017). Predicting query quality for applications of text retrieval to software engineering tasks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(1), 1–45.
- MILOUDI, C., CHEIKHI, L., ABRAN, A., & IDRI, A. (2022). Open source software maintenance effort estimation: a systematic mapping study. *Journal of Engineering Science and Technology*, 17(6), 3843–3861.
- Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L., & Vijay-Shanker, K. (2013). Automatic generation of natural language summaries for java classes. In *2013 21st international conference on program comprehension (icpc)* (pp. 23–32).
- Moslehi, P., Adams, B., & Rilling, J. (2020). A feature location approach for mapping application features extracted from crowd-based screencasts to source code. *Empirical Software Engineering*, 25(6), 4873–4926.
- Müller, R., & Eisenecker, U. (2019). A graph-based feature location approach using set theory. In *Proceedings of the 23rd international systems and software product line conference-volume a* (pp. 88–92).
- Mumtaz, H., Alshayeb, M., Mahmood, S., & Niazi, M. (2019). A survey on uml model smells detection techniques for software refactoring. *Journal of Software: Evolution and Process*, 31(3), e2154.
- Nakano, M., Noda, K., Hayashi, S., & Kobayashi, T. (2017). Mediating turf battles! prioritizing shared modules in locating multiple features. In *2017 ieee 41st annual computer software and applications conference (compsac)* (Vol. 1, pp. 363–368).
- Oliveto, R., Gethers, M., Poshyvanyk, D., & De Lucia, A. (2010). On the equivalence of information retrieval methods for automated trace-

- ability link recovery. In *2010 ieee 18th international conference on program comprehension* (pp. 68–71).
- Planas, E., & Cabot, J. (2020). How are uml class diagrams built in practice? a usability study of two uml tools: Magicdraw and papyrus. *Computer Standards & Interfaces*, 67, 103363.
- Poshyvanyk, D., Marcus, A., Rajlich, V., Gueheneuc, Y.-G., & Antoniol, G. (2006). Combining probabilistic ranking and latent semantic indexing for feature identification. In *14th ieee international conference on program comprehension (icpc'06)* (pp. 137–148).
- Qayum, A., Khan, S. U. R., Akhunzada, A., et al. (2022). Finecodeanalyzer: Multi-perspective source code analysis support for software developer through fine-granular level interactive code visualization. *IEEE Access*, 10, 20496–20513.
- Rafiei, D., & Mendelzon, A. (1997). Similarity-based queries for time series data. In *Proceedings of the 1997 acm sigmod international conference on management of data* (pp. 13–25).
- Razzaq, A., Le Gear, A., Exton, C., & Buckley, J. (2020). An empirical assessment of baseline feature location techniques. *Empirical Software Engineering*, 25(1), 266–321.
- Razzaq, A., Ventresque, A., Koschke, R., De Lucia, A., & Buckley, J. (2021). The effect of feature characteristics on the performance of feature location techniques. *IEEE Transactions on Software Engineering*.
- Razzaq, A., Wasala, A., Exton, C., & Buckley, J. (2018). The state of empirical evaluation in static feature location. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(1), 1–58.
- Rekabsaz, N., Lupu, M., & Hanbury, A. (2017). Exploration of a threshold for similarity based on uncertainty in word embedding. In *Advances in information retrieval: 39th european conference on ir research, ecir*

- 2017, aberdeen, uk, april 8-13, 2017, *proceedings 39* (pp. 396–409).
- Revelle, M., & Poshyvanyk, D. (2009). An exploratory study on assessing feature location techniques. In *2009 ieee 17th international conference on program comprehension* (pp. 218–222).
- Robertson, S. E., Walker, S., Beaulieu, M., Gatford, M., & Payne, A. (1996). Okapi at trec-4. *Nist Special Publication Sp*, 73–96.
- Rocchio, J. (1971). Relevance feedback in information retrieval. *The Smart retrieval system-experiments in automatic document processing*, 313–323.
- Rohatgi, A., Hamou-Lhadj, A., & Rilling, J. (2008). An approach for mapping features to code based on static and dynamic analysis. In *2008 16th ieee international conference on program comprehension* (pp. 236–241).
- Rohatgi, A., Hamou-Lhadj, A., & Rilling, J. (2009). Approach for solving the feature location problem by measuring the component modification impact. *IET software*, 3(4), 292–311.
- Romano, S., Scanniello, G., Antoniol, G., & Marchetto, A. (2018). Spiritus: A simple information retrieval regression test selection approach. *Information and Software Technology*, 99, 62–80.
- Rubin, J., & Chechik, M. (2013). A survey of feature location techniques. In *Domain engineering* (pp. 29–58). Springer.
- Saifan, A. A., & Obeidat, L. (2021). Feature location enhancement based on source code augmentation with synonyms of terms. *Software: Practice and Experience*, 51(2), 235–259.
- Salis, V., Sotiropoulos, T., Louridas, P., Spinellis, D., & Mitropoulos, D. (2021). Pycg: Practical call graph generation in python. In *2021 ieee/acm 43rd international conference on software engineering (icse)* (pp. 1646–1657).
- Santos, A. L., & Albuquerque, I. (2016). Viscte: Runtime exploration

- of component-based systems. In *2016 19th international acm sigsoft symposium on component-based software engineering (cbse)* (pp. 91–96).
- Sartipi, K., & Safyallah, H. (2010). Dynamic knowledge extraction from software systems using sequential pattern mining. *International Journal of Software Engineering and Knowledge Engineering*, 20(06), 761–782.
- Savage, T., Reville, M., & Poshyvanyk, D. (2010). Flat3: Feature location and textual tracing tool. In *2010 acm/ieee 32nd international conference on software engineering* (Vol. 2, pp. 255–258).
- Schmid, H. (1994). Treetagger-a language independent part-of-speech tagger. <http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/>.
- Siddharthan, A. (2014). A survey of research on text simplification. *ITL-International Journal of Applied Linguistics*, 165(2), 259–298.
- Silva, C. C., Galster, M., & Gilson, F. (2021). Topic modeling in software engineering research. *Empirical Software Engineering*, 26(6), 120.
- Sinkala, Z. T., & Herold, S. (2021). Hierarchical code-to-architecture mapping. In *European conference on software architecture* (pp. 86–104).
- Sisman, B., Akbar, S. A., & Kak, A. C. (2017). Exploiting spatial code proximity and order for improved source code retrieval for bug localization. *Journal of Software: Evolution and Process*, 29(1), e1805.
- Soueidi, C., Falcone, Y., & Hallé, S. (2023). Dynamic program analysis with flexible instrumentation and complex event processing. In *2023 ieee 34th international symposium on software reliability engineering (issre)* (pp. 742–751).
- Soueidi, C., Monnier, M., & Falcone, Y. (2023). Efficient and expressive bytecode-level instrumentation for java programs. *International Journal on Software Tools for Technology Transfer*, 1–27.
- Soueidi, C., Monnier, M., Kassem, A., & Falcone, Y. (2021). Efficient and

- expressive bytecode-level instrumentation for java programs. *arXiv preprint arXiv:2106.01115*.
- Strüber, D., Mukelabai, M., Krüger, J., Fischer, S., Linsbauer, L., Martinez, J., & Berger, T. (2019). Facing the truth: benchmarking the techniques for the evolution of variant-rich systems. In *Proceedings of the 23rd international systems and software product line conference-volume a* (pp. 177–188).
- Sulír, M. (n.d.). Program comprehension: A short literature review..
- Sulír, M., & Porubán, J. (2017a). Exposing runtime information through source code annotations. *Acta Electrotechnica et Informatica*, 17(1), 3–9.
- Sulír, M., & Porubán, J. (2017b). RuntimeSearch: Ctrl+ f for a running program. In *2017 32nd IEEE/ACM international conference on automated software engineering (ASE)* (pp. 388–393).
- Takahashi, A., Sae-Lim, N., Hayashi, S., & Saeki, M. (2021). An extensive study on smell-aware bug localization. *Journal of Systems and Software*, 178, 110986.
- Tiwari, U. K., & Kumar, S. (2020). *Component-based software engineering: Methods and metrics*. CRC Press.
- Troelsen, A., Japikse, P., Troelsen, A., & Japikse, P. (2020). Multithreaded, parallel, and async programming. *Pro C# 8 with .NET Core 3: Foundational Principles and Practices in Programming*, 525–570.
- Vale, T., & de Almeida, E. S. (2019). Experimenting with information retrieval methods in the recovery of feature-code spl traces. *Empirical Software Engineering*, 24, 1328–1368.
- Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., & Sundaresan, V. (2010). Soot: A java bytecode optimization framework. In *Cascon first decade high impact papers* (pp. 214–224).
- Vijayarani, S., Ilamathi, M. J., Nithya, M., et al. (2015). Preprocessing tech-

- niques for text mining-an overview. *International Journal of Computer Science & Communication Networks*, 5(1), 7–16.
- Wang, J., Peng, X., Xing, Z., Fu, K., & Zhao, W. (2017). Contextual recommendation of relevant program elements in an interactive feature location process. In *2017 IEEE 17th international working conference on source code analysis and manipulation (SCAM)* (pp. 61–70).
- Wautelet, Y., Heng, S., Kolp, M., & Mirbel, I. (2014). Unifying and extending user story models. In *International conference on advanced information systems engineering* (pp. 211–225).
- Whitworth, F. D. (2022). Execution path detection through dynamic analysis in black-box testing environments. In *Proceedings of the 37th IEEE/ACM international conference on automated software engineering* (pp. 1–3).
- Wilde, N., & Scully, M. C. (1995). Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1), 49–62.
- Williams, R., Ren, T., De Carli, L., Lu, L., & Smith, G. (2021). Guided feature identification and removal for resource-constrained firmware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2), 1–25.
- Wimmer, M., Schauerhuber, A., Kappel, G., Retschitzegger, W., Schwinger, W., & Kapsammer, E. (2011). A survey on uml-based aspect-oriented design modeling. *ACM Computing Surveys (CSUR)*, 43(4), 1–33.
- Wong, W. E., & Gokhale, S. (2005). Static and dynamic distance metrics for feature-based code analysis. *Journal of Systems and Software*, 74(3), 283–295.
- Wong, W. E., Gokhale, S. S., & Horgan, J. R. (2000). Quantifying the closeness between program components and features. *Journal of Systems and Software*, 54(2), 87–98.

- Wu, J., Zhang, Z., Xu, J., He, J., Mao, X., Meng, X., & Li, P. (2022). Detraque: Dynamic execution tracing techniques for automatic fault localization of hardware design code. *Plos one*, 17(9), e0274515.
- Yang, A. Z., da Costa, D. A., & Zou, Y. (2019). Predicting co-changes between functionality specifications and source code in behavior driven development. In *2019 ieee/acm 16th international conference on mining software repositories (msr)* (pp. 534–544).
- Yang, J., Lee, Y., Gandhi, D., & Valli, S. G. (2017). Synchronized uml diagrams for object-oriented program comprehension. In *2017 12th international conference on computer science and education (iccse)* (pp. 12–17).
- Zamani, S., Lee, S. P., Shokripour, R., & Anvik, J. (2014). A noun-based approach to feature location using time-aware term-weighting. *Information and Software Technology*, 56(8), 991–1011.
- Zamani, S., Lee, S. P., Shokripour, R., & Anvik, J. (2016). A feature location approach supported by time-aware weighting of terms associated with developer expertise profiles. *Knowledge and Information Systems*, 49(2), 629–659.
- Zangeneh, S. (2021). *Test2vec: An execution trace embedding for behaviour coverage analysis of test cases* (Unpublished master’s thesis). Schulich School of Engineering.
- Zhang, T., Gebhard, P., & Sokolsky, O. (2016). Smedl: combining synchronous and asynchronous monitoring. In *International conference on runtime verification* (pp. 482–490).
- Zhao, W., Zhang, L., Liu, Y., Sun, J., & Yang, F. (2006). Sniafl: Towards a static noninteractive approach to feature location. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(2), 195–226.
- Ziftci, C., & Krüger, I. (2012). Feature location using data mining on

existing test-cases. In *2012 19th working conference on reverse engineering* (pp. 155–164).

ملخص الدراسة

تهدف تقنيات تحديد الموقع الديناميكي (DFLTs) إلى أتمتة عملية تحديد كود المصدر المسؤول عن تنفيذ ميزات محددة داخل أنظمة البرامج. يستغرق التنفيذ اليدوي لـ DFLTs وقتًا طويلاً ويتطلب الكثير من المطورين، مما يؤدي إلى اقتراح أساليب شبه آلية. يتضمن أحد الأساليب الشائعة إنشاء آثار التنفيذ عن طريق تنفيذ سيناريوهات متعددة لكل ميزة برنامج ثم تعيين كود المصدر المقابل بناءً على هذه الآثار. ومع ذلك، فإن آثار التنفيذ غالبًا ما تكون كبيرة وتحتوي على تشويش، مما يتطلب حلولاً لتقليل حجمها وإزالة البيانات غير ذات الصلة. يتضمن أحد هذه الحلول تقليل عدد السيناريوهات اللازمة لممارسة إحدى ميزات البرنامج، ولكن لم يتم إنجاز سوى القليل من العمل في هذا المجال.

ولمعالجة هذه الفجوة، تم تقديم إطار عام يسمى تجميع آثار التنفيذ لصياغة سيناريو (AETFS) في هذا العمل. يستفيد AETFS من مخرجات برامج وقت التشغيل ويستخدم تقنيات تحليل المشاعر النصية لاستخراج البيانات ذات الصلة من آثار التنفيذ لإنشاء السيناريو. يستكشف تحليل المشاعر النصية، بما في ذلك نمذجة الموضوع، كوسيلة لتحديد سيناريوهات دقيقة لـ DFLTs. يتميز أداء AETFS من حيث دقة تتبع التنفيذ، مما يتيح تحديد المصطلحات ذات المعنى التي يمكنها تصفية تتبع التنفيذ باستخدام التصنيف النصي المطبق في أداة مصممة خصيصًا. يشمل التقييم ثمانية أنظمة موضوعية تحتوي على 600 ميزة، مما يجعلها أكثر شمولاً من الدراسات السابقة.

تحدد الدراسة سمات معينة لآثار التنفيذ والاستعلامات النصية التي تؤثر على أداء AETFS. تظهر مجموعتان متميزتان، إحداها تحقق موقع ميزة متفوق (FL) باستخدام AETFS والأخرى تحقق موقع ميزة أفضل باستخدام طريقة أساسية تقليدية. يؤدي الجمع بين AETFS والطريقة الأساسية إلى تحسين الأداء بشكل كبير، حيث تتجاوز أفضل النتائج التجريبية خط الأساس بنسبة 45% وأسوأ النتائج التجريبية تتجاوز AETFS بنسبة 12%.

في الختام، يسلط هذا العمل الضوء على أهمية التوصيف الدقيق لإطار عمل DFLTs المقترح لتحديد السيناريوهات المثلى لممارسة ميزات البرنامج. ويؤكد على الحاجة إلى التمييز بين السيناريوهات وتوصيفاتها لتوليد الأفكار اللازمة. توضح النتائج فعالية AETFS مع توفير رؤية قيمة لمزيد من التقدم في مجال DFLTs.

جامعة البحرين

كلية تقنية المعلومات



إطار موقع الميزة الديناميكي لمشروع البرمجيات

أطروحة مقدمة كجزء من متطلبات الحصول على
درجة الدكتوراة في علوم الحوسبة والمعلومات

إعداد

فيصل محمد بوزيد

٢٠٢٠١١٢٢٢

إشراف

د. فوزي البلوشي

(أستاذ مشارك)

جامعة البحرين

مملكة البحرين

أبريل ٢٠٢٤