POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Towards Understanding Modern Multi-Language Software Systems

MANEL GRICHI
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Août 2020

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Towards Understanding Modern Multi-Language Software Systems

présentée par **Manel GRICHI**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

DEDICATION

To my parents
To my grand-mother
To my husband
To my sister and my brother
To my friends
For their endless love, support, and encouragement

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to my supervisor, Professor Bram Adams, for his exceptional support, encouragement, and for pushing further my confidence to finish my Ph.D. I am thankful for all the opportunities he gave me, his guidance, as well as all the professionalism he showed while dealing with me.

My sincere thanks go also to my internship supervisor, Fehmi Jaafar, for allowing me to be an intern in the Computer Research Institute of Montréal (CRIM). I believe that his effort and support throughout the thesis, his endless advice, and his contributions led me to complete my Ph.D. study. I would also like to convey my deepest appreciation to the CRIM, especially to Mr.Hans Bherer, for accepting me among their team and for the professional environment that they have.

I would like also to thank my thesis committee, Professor Giuliano Antoniol, Professor Houari Sahraoui, and Professor Ghizlane ElBoussaidi for accepting my invitation to be jury members.

I am thankful to all my co-authors, especially Ellis Emmanuel Eghan, for their endless help and guidance in getting this thesis to an end.

Furthermore, not to forget all my friends and colleagues for sharing ideas, productive discussions, joy, and happiness.

Last but not least, I'm so grateful and thankful to my beloved family for their endless love and sacrifice. You were my strengths through these difficult four years and without you, this day would never have existed and I would never get to achieve my dream. I love you and may God bless you all.

RÉSUMÉ

Aujourd'hui, la plupart des applications et sites Web tels que Google et Facebook sont des systèmes multi-langages. Google est codé en C, C++, Go, Java, Python et JS. Facebook, quant à lui, utilise Hack, PHP, Python, C++, Java, Erlang, D, Haskell et JS. De plus, de précédentes études ont permis de constater que les développeurs PHP utilisaient régulièrement deux langages en plus du PHP. Les développeurs Java utilisent également le C/C++ avec du code Java à travers la Java Native Interface (JNI) qui permet d'appeler des fonctions natives à partir de méthodes Java et des méthodes Java à partir des fonctions natives. De plus, les développeurs d'applications Android préfèrent utiliser Android NDK (qui permet d'utiliser du code C/C++ avec Android) en plus de Java, plutôt que d'utiliser uniquement Java. Dans tous ces exemples, on observe le phénomène du développement multi-langage.

Bien qu'il y ait, le plus souvent, un seul langage principal (Java ou C/C++) avec divers contributions dans d'autres langages (par exemple, bash ou make), les logiciels modernes sont de plus en plus hétérogènes dans le sens où ils combinent de multiples langages de programmation qui interagissent d'une façon significative avec le langage principal. Le développement multi-langage représente une bonne pratique dans le développement de logiciels car il tire profit des bibliothèques écrites dans d'autres langages de programmation et de la réutilisation de code, ce qui permet un gain en terme de temps de développement et de budget des projets.

Par contre, puisque le développement multi-langage consiste à combiner des langages ayant des règles de programmation différentes, cela contribue à rendre le code plus difficile à comprendre et, surtout, à rendre plus difficile la maintenance du code, la synchronisation des dépendances entre les différents composants, la communication entre langages, la gestion des exceptions, et l'analyse d'impact des changements. Ces difficultés ne sont pas entierement abordées dans la littérature. Les principales préoccupations des développeurs identifiées dans la littérature sont : l'analyse d'impact des changements ainsi que son lien avec les dépendances de composants, et le choix des pratiques de programmation appropriées au multi-langage.

Ainsi, dans cette dissertation, nous prenons d'abord du recul pour mieux comprendre comment et pourquoi les développeurs optent pour le développement multi-langage avant d'évaluer les défis associés à ce type de développement et son impact sur la qualité et la sécurité des logiciels. Pour cela, nous avons élaboré plusieurs études empiriques qualitatives et quantitatives sur des projets multi-langages. Nous avons d'abord réalisé une étude systématique de la littérature où nous avons tenté de comprendre la démarche du développement du multi-langage, quels langages à combiner, quelles techniques permettent cette combinaison, quand il est préférable de faire appel au développement multi-langage, et quels défis et problèmes rencontrent les développeurs durant ce processus. Cette étude nous amené à établir principalement que : (1) Java Native Interface est la technique la plus utilisée dans le développement multi-langage depuis 2010 et que (2) l'analyse d'impact de changement et les bonnes practiques sont les principales préoccupation des utilisateurs du développement du multi-langage.

Afin de mieux comprendre comment les développeurs gèrent le développement multi-langage, nous avons, lors d'une seconde étude, mené une analyse qualitative afin de comprendre l'utilisation de Java Native Interface par les développeurs (un sous-type de développement multi-langage). Nous avons identifié 11 bonnes pratiques et nous avons investigué leur présence dans 100 projets JNI. Il s'agit d'une liste de bonnes pratiques que nous recommandons fortement aux développeurs durant le développement JNI.

Nous nous sommes ensuite intéressés à la seconde principale préoccupation des développeurs multi-langages, c'est l'analyse d'impact des changements. Nous avons mené une enquête où nous avons sondé 69 développeurs expérimentés, ensuite nous avons interrogé par un appel conférence huit d'entre eux. Nous leur avons demandé quelles sont les motivations des développeurs pour l'analyse d'impact des changements, comment ils gèrent les changements dans les systèmes multi-langages. En particulier, nous avons enquêté sur les outils, les techniques et les méthodes suivies par les développeurs pour l'analyse d'impact des changements. Également nous avons interrogé les participants sur les difficultés auxquelles ils font face lorsqu'ils mènent ces analyses et comment celles-ci se répercutent sur la qualité et la sécurité des systèmes. D'après le sondage et les entrevues, nous avons principalement constaté que les développeurs rencontrent des difficultés au niveau de l'analyse des dépendances du code à travers les différents composants écrits dans différents langages de programmation.

À partir de cela, nous avons mené une quatrième étude empirique sur dix projets Java Native Interface afin d'identifier leurs dépendances multi-langages et leur impact sur la qualité et la sécurité logicielles. Nous avons également introduit deux approches pour l'analyse des dépendances multi-langages : SMLDA, analyseur des dépendances statiques multi-langages qui réalise une analyse des dépendances statiques à l'aide d'heuristiques et de régles conventions et H-MLDA, analyseur des dépendances historiques multi-langages qui réalise une analyse des dépendances historiques.

Au cours de cette thèse, la popularité croissante de l'intelligence artificielle (IA) a commencé à se répercuter sur le domaine du développement multi-langage, conduisant à l'adoption concrète de dizaines de frameworks d'apprentissage automatique multi-langages. Ainsi, dans notre dernière étude, nous avons analysé la prévalence du développement multi-langage dans les frameworks d'apprentissage automatique afin de comprendre si ces frameworks suivaient la tendance du développement multi-langage, en nous posant la question si la pratique du développement multi-langage augmente-t-elle la difficulté à gérer les frameworks d'apprentissage automatique ? Nous avons donc analysé empiriquement dix projets de frameworks d'apprentissage automatique multi-langage et dix projets traditionnels dont nous avons étudié l'impact du développement multi-langage en terme de taux d'acceptation, de la durée du processus de révision et de la propension aux bogues introduits aux pull-requests. Nous avons observé une corrélation entre le développement multi-langage et les frameworks d'apprentissage automatique. Le développement multi-langage influence les contributions du logiciel (pull-requests) aux frameworks d'apprentissage automatique dans la mesure où la nature multi-langage de ces frameworks est due à la collaboration de ces deux groupes de personnes (développeurs en génie logiciel et data scientist).

Notre thèse a validé que, malgré les divers avantages du développement multi-langage, une attention particulière et une bonne manipulation sont encore nécessaires pour les développeurs afin d'en surmonter les limitations et inconvénients du développement du multi-langage. Nous proposons aux développeurs une liste de bonnes pratiques afin de les aider à augmenter la qualité de leurs programmes. Nous avons constaté que le maintien du système multi-langage présente un défi, en particulier en ce qui concerne l'analyse des dépendances. Nous constatons que l'augmentation des dépendances multi-langages entraine le risque d'introduire des bogues. Nous avons ainsi proposé deux approches (statique et historique) pour l'analyse des dépendances multi-langages afin d'aider les développeurs dans le développement multi-langage. Nous avons observé que ce type de développement avait un impact sur les contributions du logiciel (les pull-requests) et menait à une augmentation du pourcentage du rejet.

ABSTRACT

Today, most popular applications and websites such as Google and Facebook are multi-language systems. Google is developed with C, C++, Go, Java, Python, and JS, while Facebook is using Hack, PHP, Python, C++, Java, Erlang, D, Haskell, and JS. Furthermore, previous research studies reported that PHP developers regularly use two languages besides PHP. Java developers also use C/C++ with Java code through the Java native interface (JNI) that allows to call native functions from Java methods and Java methods from native functions. Moreover, Android application developers prefer to use the Android NDK (allows to use C/C++ code with Android) along with Java compared to using only Java. In all these examples, we observe the phenomenon of multi-language development.

While in many cases, there is one clear main language (*e.g.,* Java or C/C++) with various smaller contributions from other languages (*e.g.,* bash or make), increasingly more modern software are heterogeneous in the sense that they are composed of multiple programming languages that interact with the host language to a large extent. Multi-language development presents a good practice for software development because it takes advantage of existing libraries written in other programming languages (code reuse), which leads the industry to save development time and project budgets.

As multi-language development consists of combining languages with different semantics and lexical programming rules, this leads to complicate the code comprehension and, especially, the code maintenance, *i.e.,* dependency tracking, data synchronisation between the different components, communication between the combined languages, exception management, and change impact analysis. Not all of these challenges are discussed in the literature. Based on the literature, the major concerns of developers consist of: Change impact analysis and its relation with component dependencies and the adequate choice of multi-language design patterns.

Hence, in this thesis, we first take a step back to better understand how and why developers opt for multi-language development, before evaluating the challenges related to this kind of development and their impact on software quality and security. For these purposes, we conducted several qualitative and quantitative empirical studies on large open-source multi-language software systems. We first conducted a systematic literature review study where we tried to understand the multi-language process, *i.e.,* what are the languages available to be combined, what are the techniques allowing that combination, when is it preferable to use multi-language development, and the challenges and problems developers face during

this process. We mainly identified from this study that (1) Java Native Interface is the technique most commonly used in multi-language development since 2010 and (2) change impact analysis and design patterns are the major concerns of multi-language users.

To better understand how developers manage their multi-language development, in a second study, we conducted a qualitative analysis to understand the usage of Java Native Interface by developers, *i.e.,* a sub-type of multi-language development. For that, we identified a set of 11 practices where we investigated their usage in 100 JNI systems. This set presents a list of best practices that we highly recommend developers to follow during their JNI development.

We then focused on the major concern of the multi-language developers *i.e.,* change impact analysis. We conducted a survey where we surveyed 69 experienced developers and interviewed eight of them. We asked about the developers' motivations for change-impact analysis, how they deal with changes in multi-language systems. In particular, we investigate tools, techniques, and methodologies followed by developers in industry for change-impact analysis and we also inquire about the difficulties that they face when conducting such analysis and how those reflect in the system quality and security. From the survey and the interview, we mainly identified that developers are suffering from the difficulties of tracking code dependencies across components written in different programming languages.

Based on that, we conducted a fourth study where we performed an empirical study on ten large open-source Java Native Interface software systems to identify their multi-language dependencies and the impact on software quality and security. We also introduced two approaches for multi-language dependency analysis: S-MLDA (Static Multi-language Dependency Analyzer) that performs a static dependency analysis using heuristics and naming conventions, and H-MLDA (Historical Multi-language Dependency Analyzer) that performs historical dependency analysis.

During the course of my thesis, the increasing popularity of Artificial Intelligence (AI) has started to impact the area of multi-language development in the form of dozens of multi-language machine learning frameworks being developed and adopted in practice. Thus, in the last study of this thesis, we were interested to analyze the prevalence of multi-language development in machine learning frameworks in order to understand to what extent these frameworks are following the multi-language development trend. Does the practice of multi-language development increase the difficulty of dealing with machine learning frameworks? Hence, we empirically analyze the ten largest open-source multi-language machine learning frameworks and the ten largest open-source traditional systems to study the impact of multi-language development in terms of the volume, acceptance rate, review process duration, and bug-proneness of pull requests. We identified a correlation between the existence of

multi-language development with machine learning frameworks. Multi-language development influences software contributions (pull requests) to machine learning frameworks. To some extent, the multi-language nature of these frameworks is due to the collaboration of these two groups of people, *i.e.,* software developers and data scientists.

Our thesis confirmed that, despite the various advantages that offers multi-language development, it still needs special attention by developers to overcome the associated limitations and inconveniences. We proposed a set of best practices to the multi-language developers to help them increase the quality of their development. We found that maintaining a multi-language system is challenging, in particular in terms of dependency analysis. Our findings show that the more multi-language dependencies, the higher the risk of bug introduction. Thus, we proposed two approaches (Static and Historic) for multi-language dependency analysis to support developers within their multi-language development. We also find that this kind of development has an impact on the software contributions (*i.e.,* pull-requests) and leads to an increase in the percentage of rejected contributions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

A software component can be implemented in any programming language. Depending upon the nature and functionality provided by a component, it may be developed more efficiently or elegantly in one programming language than in another, or perhaps the choice of programming language is merely determined by a programmer's preference. In any case, since any component theoretically could be developed in any language, a software application may contain several components written in different languages. The creation of such systems is called multi-language development [4], *i.e.,* opting in the same software system for the use of two or more programming languages having different lexical, semantic, and syntactical programming rules.

The software development industry has been taking advantage of multi-language development not only in low-level languages such as Assembly, but also in high-level programming languages such as C, Ada, Pascal, Cobol, etc [5]. For example, the integration of C (high-level language) with Assembly (low-level language) provides extra flexibility to the program where the static libraries are written in Assembly and called from C, and vice-versa [1]. As such, with the passage of time, the usage of multi-language development has increased. A recent survey has shown that the use of multi-language development in open source software systems is getting popular with, on average, five languages used in a system [6, 7].

We can observe its usage in many popular websites and software systems that are built using a combination of heterogeneous components written in multiple programming languages, *e.g.,* C, C++, Java, JSP, PHP [8]. Their integration is performed through techniques that play the role of a bridge to ensure the synchronisation between the combined languages. The most known integration techniques are Java Native Interface (JNI), which ensures the communication between Java and C/C++, and Python-C-extension, that ensures the communication between Python and C/C++. JNI and Python-C-extension are two examples of a Foreign Function Interface (FFI). We will discuss these concepts in more detail in Section 2.1.

The multi-language development has brought several advantages to software engineering. Today, developers are often using more than one single programming language in the same software system to gain the benefits from the advances made in other programming languages [9]. First, multi-language development has made it easier for industry and developers to keep up with market development speed, since it allows code reuse. Thus, developers no longer need to reinvent the wheel and re-code everything from scratch. Instead, they are asked

1. `https://www.devdungeon.com/content/how-mix-c-and-assembly`

to integrate external libraries or external API written in a foreign programming language to gain time and effort [1]. Second, multi-language offers to benefit from the specificity of particular programming languages. As we know, each programming language has weaknesses and strengths ; thus, combining languages allows to benefit from each language's strengths and overcome its weaknesses. Last, multi-language offers developers to keep using legacy implementations of their existing libraries and systems by encapsulating them within a system developed in a different language [10].

As an example, Java being a high-level programming language offers many advantages over other programming languages, such as the absence of pointers and explicit memory management, built-in security mechanism (*e.g.,* sandboxing) and the possibility of building code portable across platforms. However, since Java also has some limitations, it offers the facility to integrate and benefit from different programming languages. For example, JNI integrates Java with C/C++, while Jython integrates Java with Python for embedded scripting, rapid application development and interactive experimentation. Also, Jperl was developed by integrating Perl and Java, for example to use CPAN modules without having to rewrite them in Java. Several other languages like Tcl, OpenGL for GUI features, Groovy, Scala for build, REDUCE for algebraic features are also integrated with Java [11].

Several popular software applications are implemented using multiple languages. For example, the Linux operating system consists of a kernel written in C with several parts implemented in assembler (for performance reasons) and several utilities written in C++, Perl, and Python. Similarly, OpenCV, an open-source computer vision library, is developed using multiple languages *i.e.,* C++, C, Python, Java, and JavaScript [12]. Many other popular software applications have been built using multi-language development such as Google, Facebook, Youtube, Mozilla Firefox, MS Office, and OpenOffice to name a few.

While the benefits of multi-language development have been deeply discussed in the literature, the associated software engineering challenges are still an ongoing problem. Despite the advantages presented above (*e.g.,* code reuse, encapsulating legacy code, gain in development time, etc), multi-language development leads to more complex and complicated software systems. Using multiple languages in a software system means that developers must have a good command of each of these languages for fully understanding, analyzing, and evolving the software system [13]. Developers working on any of the system's components are required to have experience in multiple programming languages at the same time. In addition, they need to consider the synchronisation and the data conversion between the different integrated languages since each programming language has its own rules (*i.e.,* lexical, semantic, and syntactical). Hence, multi-language development increases the cognitive overhead of code comprehension and leads to difficulties in code maintenance and adaptability.

Such difficulties unfortunately might lead to bugs that are hard to detect and debug. `ID200551` is a bug identified in Rhino, developed with Java, JS, Perl, and C , that was caused by a mismatch between changes applied in one component *i.e.,* Java class Kit and the corresponding changes that should be applied in a second component written in a different language C class DefiningClassLoader [14]. Another bug in libguests happened in 2018 and was due to a misuse of the rules of Java Native Interface (JNI) when integrating Java and C++ code [2]. The developer did not synchronized correctly the Java exceptions with the JNI calls. Furthermore, bug `322222`, found in the Eclipse bug repository led to crashes in the JVM through a segmentation fault in C when the program throws an exception in Java [15]. A final example concerns an issue reported by Microsoft Word users when using Visual Basic macros. These issues were due to a mismatch between unique identifiers in C++ libraries and in Visual Basic [3]. As showed in these examples, multi-language bugs is mainly due to the mismatch and the incompatibilities between the entities written in different programming languages. The challenge does not come necessary from the combination of languages itself but more from the lack of means *i.e.,* tools, documentations, methodologies that could provides support for developers to avoid or limit these kind of bugs.

In addition to the multi-language development trend within "traditional" software systems, in recent years multi-language development has been adopted massively in the domain of AI-based software systems. In particular, many (open-source) frameworks for machine learning (ML), such as Pytorch and Theano, have been engineered using multi-language practices, typically to integrate a low-level language for efficient computations with a high-level language for building robust software frameworks [16]. Similar to traditional (non-machine learning) frameworks, these ML frameworks need to evolve to incorporate new features, optimizations, etc., and their evolution is impacted by the interdisciplinary development teams needed to develop them : data scientists and software developers. Scientists have to write optimized low-level code while developers need to integrate the latter into a robust framework. Based on that, in this dissertation, we are interested in studying the prevalence of multi-language development in both traditional and machine learning frameworks, and their impact on the software system's quality and security.

2. `https://bugzilla.redhat.com/show_bug.cgi?id=1536762`
3. `support.microsoft.com/kb/292744`

1.1 Thesis Hypothesis

While multi-language development offers facilities to developers, it potentially makes a software system more complex and hard to maintain, which can cause more bugs that are harder to track down, more severe, and harder to debug. Therefore, it is important to obtain a better understanding of the usage and maintenance of multi-language development as well as the challenges that developers could face during their development activities. Thus, in the following, we present our research hypothesis composed of three sub-hypotheses :

Thesis Hypothesis :
We hypothesize that (1) multi-language development is common, spanning many combinations of languages through specific techniques, and, over time, has led to a catalog of best practices ; (2) the integration of multiple languages with different development rules (semantic and lexical) makes maintenance activities, such as change impact analysis, more complex and bug/vulnerability-prone ; (3) the impact of multi-language development on software contributions and bug introduction is larger in machine learning frameworks than in traditional systems.

To validate the three sub-hypotheses :

1. We systematically review the research literature (initial set of 3964 papers, filtered down to 138 papers) to understand the concept of multi-language development, the process followed to apply it and the challenges that are faced by developers during that process. Then, we qualitatively study the usage of multi-language development by developers in open-source systems, in which we identified a set of 11 technical practices of multi-language development. In this qualitative study, we focus on the most commonly used technology for integrating languages, *i.e.,* the Java Native Interface. This addresses the first sub-hypothesis, and is detailed in Chapter 4 and Chapter 5.
2. Then, we perform a survey with developers working in 52 software companies across a diverse set of domains to evaluate their process for the change impact analysis in multi-language systems, *i.e.,* to investigate the challenges, the tools, methodologies, the impact on system quality and system security, and the consequences of a lack of change impact analysis practices.

   One of the major and important steps of the change impact analysis identified by this survey consists of tracking down all dependant components that might have to be changed according to a given change made to the system. Hence, we empirically study multi-language open source systems to identify the dependencies in multi-language development, analyze their prevalence, and their impact on software quality and security. Finally, we propose static and history-based approaches for multi-language dependency analysis. This addresses the second sub-hypothesis, and is detailed in Chapter 6 and Chapter 7.

3. Lastly, we investigate the prevalence of multi-language development in machine learning frameworks and evaluate the impact of multi-language software contributions (pull-requests) made to those frameworks on the acceptance rate, the frameworks' software quality, and the frameworks' software security. We perform a comparison between multi-language development in machine learning and in traditional multi-language systems. This addresses the third sub-hypothesis, and is detailed in Chapter 8.

1.2 Summary of Thesis Contributions

We describe in the following a general overview of the different contributions of this dissertation, grouped by chapters. We detail them and show their interactions in Chapter 3, before presenting each study in detail in their respective chapters. Figure 1.1 shows a general overview of the three parts of this thesis and the output of each part.

1.2.1 Contribution of Chapter 4 (Sub-hypothesis 1)

First major contribution : a systematic literature review about multi-language development, *i.e.,* existing combinations of languages, contexts of use, challenges of multi-language development, and existing techniques.

C1 : Identification of the most common language combinations used, the technical mechanisms allowing the communication between these languages, and the contexts of multi-language usage : We conduct a state-of-the-art review on the use of multi-language development from the researcher's perspective (literature review). We aim to understand what is multi-language development, what are the combinations of languages that match together, and what are the existing techniques to bridge between the different languages ? To achieve this goal, we performed a systematic literature review [17] of 3964 papers, refined with different exclusion steps to 138 papers, published between 2010 and 2020.

Regarding the contexts of multi-language usage, we aim to know in which cases multi-language development is preferred to be used ? Are there specific contexts related to the use of multi-language development ? What are the most discussed situations in the

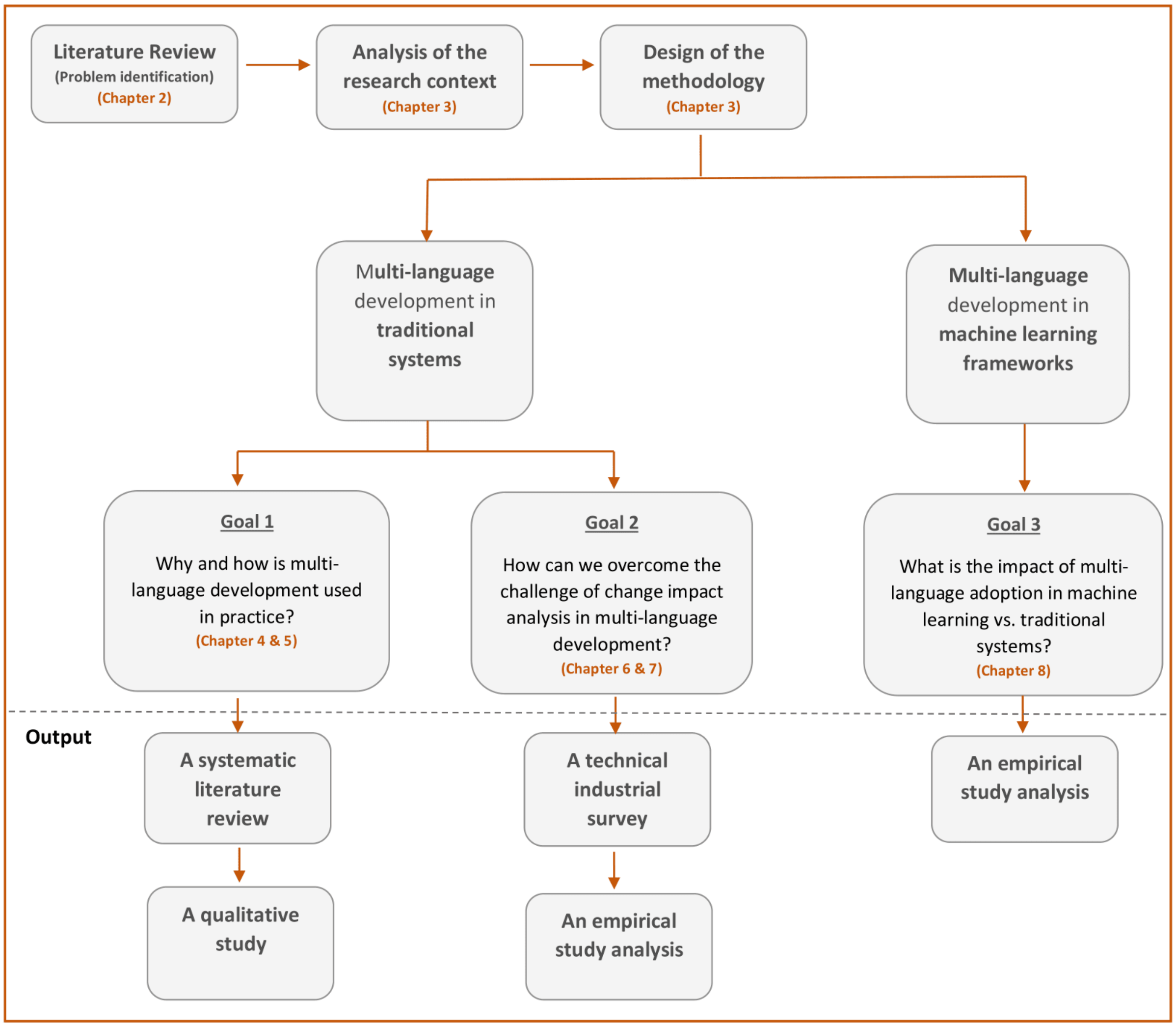

Literature Review
(Problem identification)
(Chapter 2)
Analysis of the research context
(Chapter 3)
Design of the methodology
(Chapter 3)
Multi-language development in traditional systems
Multi-language development in machine learning frameworks
Goal 1
Why and how is multi-language development used in practice?
(Chapter 4 & 5)
Goal 2
How can we overcome the challenge of change impact analysis in multi-language development?
(Chapter 6 & 7)
Goal 3
What is the impact of multi-language adoption in machine learning vs. traditional systems?
(Chapter 8)
Output
A systematic literature review
A qualitative study
A technical industrial survey
An empirical study analysis
An empirical study analysis

Figure 1.1 Overview of the thesis methodology.

literature where multi-language development usage presented a challenge ? Based on the papers identified in the first contribution, we perform an open coding process [18] in which we identify the context of the use of multi-language development categorised in different groups. From these groups, we identified the most discussed and challenging usage contexts of use of multi-language development.

1.2.2 Contribution of Chapter 5 (Sub-hypothesis 1)

Second major contribution : a catalog of good practices of Java Native Interface development.

C2 : Qualitative study of the practices used for multi-language development using Java Native Interface : How are developers implementing multi-language systems ? Do developers follow any specific practices to deal with multi-language development ? How do developers ensure data flow propagation between two components written in different languages ? To answer these questions, we refer to the integration of languages used the most in the literature, which is the integration of Java with C/C++ via the Java Native Interface. We qualitatively analyze the source code of 100 JNI systems *i.e.,* we semi-automatically analysed only the part of the Java Native Interface in the source code, to identify the most common and recurrent practices developers are following.

1.2.3 Contribution of Chapter 6 (Sub-hypothesis 2)

Third major contribution : a technical survey from the developer's perspectives about the change impact analysis in multi-language development.

C3 : Technical industrial survey to understand how developers manage changes in multi-language systems : This contribution tries to answer the following questions : What are the different steps followed by developers when conducting their changes in multi-language systems ? Are there specific testing methods or methodologies that they use ? How do they ensure the code quality of their multi-language system while making different changes ? We perform a technical survey with 69 professional developers working in different industries around the world. The goal is to trace the different steps considered by developers before and after making a change to a multi-language system.

Then, we identify the risks and the consequences on the software quality and security when applying a change to a multi-language software system without performing a change impact analysis to initially verify the potentially impacted components. Thus, we perform a coding process on the answers of the open-ended questions where we apply three steps : *open Coding*, *axial coding*, and *selective coding*. Based on the findings, we suggest a set of practices developers should pay more attention to their future maintenance changes in a multi-language context.

Lastly, we want to identify the requirements, from developer's perspectives, for an effective change impact analysis approach for multi-language systems. Using the same coding process as before, we identify 20 requirements that we categorised in three categories *i.e.,* Easily realisable, Medium realisable, and Hardly realisable.

1.2.4 Contributions of Chapter 7 (Sub-hypothesis 2)

Fourth major contribution : static and history-based approaches (S-MLDA and H-MLDA) to identify the dependencies within multi-language systems.

C4 : Approaches for a static and history-based dependency analysis in multi-language systems : We propose two approaches where the first one is based on historical dependency analysis, named H-MLDA, and the second one is based on the static dependency analysis, named S-MLDA. Our approaches show precision (recall) values of 100% (68%) for S-MLDA and 68% (87%) for H-MLDA.

Fifth major contribution : Analysis of the impact of multi-language dependencies on software quality and security, and comparing them with the impact of mono-language dependencies.

C5 : Empirical evaluation of the impact of multi-language dependencies on software quality and security : Dependencies between components written in different programming languages is a hard task, in which tracking the data dependency flow requires specific effort and rules. We aim to answer the following question : Are multi-language dependencies more risky for multi-language software systems in terms of software quality and security ? We perform an empirical study on the commits of ten open-source multi-language systems (JNI) where we used the SZZ algorithm to identify changes that are likely to introduce issues. Thus, we collect the log messages of the multi-language commits and based on a set of keywords, we identify the commits that contain a fix to a bug. Those commits are then analyzed by SZZ to determine the initial bug-introducing commits. We perform the same process for the mono-language commits involving dependencies between the same language to compare their findings with those for multi-language commits involving dependencies between different languages.

1.2.5 Contribution of Chapter 8 (Sub-Hypothesis 3)

Sixth major contribution : comparison of the impact on software quality of the multi-language development in traditional systems and multi-language development in machine learning frameworks.

C6 : Empirical analysis of the prevalence of multi-language in machine learning frameworks : To what extent are open-source machine learning frameworks following the multi-language development trend ? What is the prevalence of multi-language development in machine learning frameworks ? Does the practice of multi-language development decrease the acceptance rate of the software contributions ? Are multi-language machine learning frameworks more bug-prone than multi-language traditional systems ? We perform an empirical study on the top ten common open source machine learning frameworks where we analyzed the percentage of programming languages involved in these frameworks. Then, we focus on analyzing the software contributions (*i.e.,* pull requests) made to those frameworks, to study the prevalence of multi-language pull requests and to compare them to the mono-language pull requests within these frameworks. Last, to understand the correlation between multi-language development and the introduction of bugs in machine learning frameworks, compared to traditional systems, we analyse the bug proneness within the accepted multi-language and mono-language pull requests of machine learning frameworks and traditional systems.

1.3 Organization of the Thesis

The remainder of this dissertation is organised as follows. Chapter 2 provides background and related work on multi-language development, dependencies within multi-language systems, the integration of SE in machine learning frameworks, and the implication of multi-language development in machine learning frameworks. Chapter 3 provides a high-level overview about the research process used in this thesis. Chapter 4 studies the prevalence of multi-language development, and the existing techniques for that. Chapter 5 identifies best practices within multi-language development. Chapter 6 reports on our analysis of one of the most common challenges of multi-language development, *i.e.,* change impact analysis, while Chapter 7 discusses its relation with identifying the multi-language dependencies between components. Chapter 8 examines the prevalence of multi-language development in machine learning frameworks, their impact on the acceptance rate of the software contributions (pull requests), and on the quality and the security. Finally, Chapter 9 draws conclusions of this dissertation, discusses limitations of the work, and also outlines avenues for future work.

CHAPTER 2 BACKGROUND AND LITERATURE REVIEW

In this chapter, we present a survey of related work on multi-language development, their evolution, their different types, and their challenges on the impact on software quality. First, we will start by a background section where we provide some source code examples to define and explain the typical multi-language development concepts.

2.1 Background

For many years, the development of systems through the combination of multiple programming languages has been the subject of interest by the software community and it is still increasingly popular today [19]. Multi-language development can be used across a wide range of systems (e.g., web applications, embedded systems, mobile applications, etc.) ; its importance and advantages have been discussed by many researchers [20–24].

The communication between the programming languages is ensured by the Foreign Function Interface (FFI). The FFI [25] allows a program written in one programming language to call components/libraries/APIs or make use of services written in another one. Java Native Interface (Java/C(++)), Python C extensions (Python/C(++)), and Ruby C extensions (Ruby/C(++)) are some examples of FFI technology that consist of a bridge between the combined languages. Before continuing, we need first to understand how we can integrate languages and how we can ensure a safe synchronisation between languages that have different semantic and lexical programming rules. Thus, in the following, we present two popular examples of multi-language FFI techniques *i.e.,* Java Native Interface and the Python C extension.

2.1.1 Java Native Interface

Java Native Interface is a framework that allows the interaction between Java and C(++) code. It allows to call native functions from Java methods and Java methods from native functions. In JNI, a native method name can be any valid Java method name declared as native and should not be implemented within Java. Instead, the corresponding implementation should be done in the C/C++ part of the system, using a function name concatenated from the following components [1, 26], such that the JVM knows which function to execute : the prefix `Java_`, a fully-qualified class name, an underscore (_) separator, and a method name. More details about the other criteria checked by the JVM to validate the exact native

```
class HelloWorld {
    static {
         AccessController.doPrivileged(
         new PrivilegedAction<Void>() {
         public Void run() {
           System.loadLibrary("HelloWorld");
         return null; }
         }
    }
    private native void print();
    public static void main(String[] args) {
        new HelloWorld().print();
    }
}
```

Figure 2.1 HelloWorld Example : JNI Method Declaration [1]

```
#include <jni.h>
#include <stdio.h>
#include "HelloWorld.h"
JNIEXPORT void JNICALL
Java_HelloWorld_print(JNIEnv *env, jobject obj)
{
    printf("Hello World!\n");
    return;
}
```

Figure 2.2 HelloWorld Example : Implementation Function [1]

method to call [1], are provided in Section 7.2.1.

Figures 2.1 and 2.2 [27] show a common and simple example of the use of JNI in a "HelloWord" class. Figure 2.1 shows the Java class that contains a native method declaration, `Print()`, and that loads the native library providing an implementation for this method. Figure 2.2 presents the C file that implements the function `Print()`. `JNIEXPORT` and `JNICALL` are the macros needed to link the method declaration in Java with the corresponding implementation in C. These macros ensure that functions are exported correctly from the native library and that C compilers generate code with the correct calling convention for that function [27].

```
>>> import ntuple
>>> ntuple.create_ntuple(5)
```

Figure 2.3 Ntuple example : Python Code [2]

```
    static PyObject* create_ntuple(PyObject *self, PyObject *args) {
         int n, i, err;
         PyObject *tup = NULL;
         PyObject *item = NULL;
         if (!PyArg_Parse(args, "(i)", &n)) return NULL;
         tup = PyTuple_New(n);
         if (tup == NULL) return NULL;
         for (i=0; i<n; i++) {
           item = PyInt_FromLong(i);
           if (item == NULL) {Py_DECREF(tup); return NULL;}
           err = PyTuple_SetItem(tup, i, item);
           if (err) {
             Py_DECREF(tup);
             return NULL; } }
         return tup;}
```

Figure 2.4 Ntuple example : C Code [2]

2.1.2 Python-C-extension

Python-C-extension allows a Python program to interact with a native module written in C or C++ by developing a native extension module [28]. The extension module provides a set of different native functions. In practice, developers write a C/C+ code that can be imported into Python code as an extension module. Figures 2.3 and 2.4 provide further details. Figure 2.3 shows the extension module called `ntuple` included in Python code. Figure 2.4 shows the C function `create_ntuple()` that will be called when the Python expression `create_ntuple(5)` is treated. The Python-C-extension defines the PyObject type and a set of subtypes that can be used by extension code, such as PyIntObject and PyStringObject. The parameter `args` presents the list of objects passed from Python code [2].

2.2 Literature Review

2.2.1 Multi-language Development in Traditional Systems

Practicing multi-language development is challenging and presents several difficulties. Mushtaq *et al.* [19], through their study, emphasized the significance of addressing the complications that emerge from using more than one language for software development. Also, they proposed several approaches to minimize the challenges of multi-language development and list the few existing code comprehension and maintenance tools that are reserved for multi-language development, along with their limitations. Similarly, Kul-bach *et al.* [29] highlighted the importance of program comprehension in multi-language systems. They argued that it plays a vital role in improving the efficiency of software development and maintenance processes in multi-language systems.

To evaluate the impact of using multi-language development, Bissyandé *et al.* [30] explored the interoperability and the effect of multiple programming languages in open-source software systems, also from GitHub, on software quality attributes. They reported that multi-language development should be used with caution in software systems. In the context of our research, a similar empirical study has been done to investigate how multi-language development influences software quality and leads to increase the bug-prone in the context of machine learning frameworks, not only traditional multi-language systems.

In order to provide developers with multi-language development guidelines that they should follow to decrease bug-prone, Abidi *et al.* [10] investigated the developers' level of knowledge on the standard practices of multi-language development by surveying 93 developers. A set of good practices, initially collected from the literature, was proposed to these developers. The authors identified that these practices are not equally used in practice *i.e.,* industry. Furthermore, they signified the need for developers to be more attentive with the use of the good practices of multi-language development, such as managing exceptions between Java and C, loading libraries, etc. In Chapter 4, we propose a set of practices more specific to the use of the Java Native Interface, a sub-type of multi-language development.

Other researchers carried out studies specific to JNI multi-language systems. Moise and Wong [31] were among the first researchers who investigated and recorded the inter-language dependencies among multi-language components. They studied the calls to the Java Native Interface API in Java and C and they also used the available API for each language to identify multi-language call dependencies.

Gong *et al.* [32] analyzed the source code of JDK v1.2, where they focused on the Java Native Interface part. They outlined the main limitations of its security features including, but not limited to, access rights, library loading, and exception management. Furthermore, they analyzed the impact of deploying new features on the existing code without respecting the Java Native Interface security. However, their study was limited to only one JDK version. During our qualitative study presented in Chapter 5, we use the JDK v9 in addition to 10 open-source Java native interface to identify the best practices in JNI development.

Tan *et al.* [33], on the other hand, investigated the JNI usage in the JDK v1.6. Their study focused on the analysis of a set of bug patterns in the native source code where they highlighted two bug categories *i.e.,* buffer-overflows and unexpected control flow path, due to a misuse of JNI exceptions. They managed to identify bugs that concerned the usage of native methods resulting into JVM crashes and security breaches being exposed. They proposed a static and a dynamic approach to help in minimising these bugs. However, they restricted their study to a portion of the native code in JDK v1.6 *i.e.,* java.util.zip in Sun's JDK, about 800k lines of code. Kondoh *et al.* [34] emphasized four types of common JNI issues employed by the developers. They carried out a study to analyze in depth the following JNI issues : error checking, invalid references, and the calling of native methods in critical regions. The authors confirmed that these poor practices may lead to several problems and open up the software system to security vulnerabilities. Therefore, they recommended BEAM, a static tool to identify such issues in Java Native Interface systems.

Li *et al.* [35] examined the risks caused by the software exceptions between C and Java. They reported that when the exceptions are misused in the Java Native Interface software, they lead to issues in the implementation functions and risk to decrease the software security. They proposed a static analysis tool to examine and report potential risks in JNI systems. In our thesis, we propose a set of JNI practices that support developers to overcome some of the JNI risks. Additionally, we proposed approaches to developers to better manage their JNI maintenance and avoid the introduction of bugs to the software system. Our proposal consists of a combination of two analysis type *i.e.,* static and history-based approaches.

2.2.2 Change Impact Analysis in Multi-language Systems

Change-impact analysis is a process that consists of evaluating the risk associated with a change, *i.e.,* its potential consequences [36, 37]. To make a safe change to a software system, it is necessary to investigate the dependencies between the source-code entities and how the change flow (change propagation) is moving from one source code entity to others in the system. If a given method is renamed because of a new client requirements, then all the dependant source-code entities that call or use this method need to be changed as well according to the new requirement [14, 38].

Let's consider the same source code example presented in Figure 2.1 and 2.2 on page 11. Based on a new requirement, the Java method `print()` should have a new parameter, thus the signature method should become `private native void print(int x)`. As shown in Section 2.1.1, this method has a JNI dependency with the C implementation function `Java_HelloWorld_print(JNIEnv *env, jobject obj)`, thus, the change that will be applied to the Java method `print()` will also impact the implementation function in the C file. This is what we call the impact of a change in multi-language development. The signature of the C implementation function also needs to be changed and the same parameter added in the Java declaration method should be added in the C function with respect to the JNI rules *e.g.,* JNI type conversion. Hence, the function signature in C file should became : `JNIEXPORT void JNICALL Java_HelloWorld_print(JNIEnv *env, jobject obj, jint x)`.

Since the 1980s, there have been many research works published on change impact analysis in mono-language systems. Wilkerson [39] classified change impact analysis in direct and indirect changes. Entities are directly affected if they have been modified in any way since the prior version. This impact is easy to identify using for example the lexical `diff` algorithm [39]. The `Chianti` algorithm aims to address the analysis of direct impact [40]. Indirect impact is a modification that results from a dependency on other entities. `RA` [41], `CHA` [42], `RTA` [43], and `0-CFA` [44] are all algorithms for indirect analysis change impact. Li *et al.* [45] presented through their survey a taxonomy of change impact tools, categorized according to the change application area : `Software comprehension`, `Change propagation`, `Regression testing`, `Debugging`, and `Software measurement`. These tools are dedicated to mono-language systems, offering different options such as mono-language dependency analysis, proposing adequate testing methods, quality metrics, etc. During this thesis, in Chapter 6, we studied the context of changes application within multi-language development.

The JRipples [46] and ROSE [47] tools can be used for software comprehension before the implementation of a change. Jtracker [48] and JRipples [46] are mono-language analysis tools for change propagation. They can be used by maintainers when a change is made to ensure the security for other entities in the software systems. Chianti [40] and Celadon [49] used for regression testing. Crisp [50], AutoFlow [51] and Chianti are able to perform debugging. They are used when regression tests fail. The last application area concerns software measurements that can be made with the ROSE tool.

Jiang *et al.* [36] investigated through their paper the ways that developers follow to perform a change impact analysis in mono-language systems. By a study conducted on 35 professional developers, they found that developers usually did a static impact analysis before applying the change and a dynamic change impact analysis after they made changes. The static analysis is done using IDE navigational functionalities, while the dynamic analysis is made by executing the programs. The authors found that developers in their study did not use any dedicated change impact analysis tools. As this study was done on mono-language systems, during this dissertation we investigate how developers perform and assess changes in multi-language systems.

Hassaine *et al.* [52] used a static change impact analysis technique to study the change propagation in time by analyzing the syntax and semantic of the code. They proposed a novel approach based on an earthquake metaphor where it analyze how far the propagation will proceed from a given class to the others.

In the context of change impact analysis methodologies, Maia *et al.* [53] proposed a solution that combines static and dynamic analysis for Java software systems. They used static analysis to identify structural dependencies between source code where as the dynamic analysis was used to identify the dependencies based on the code execution. They have compared their results, finding that the hybrid technique (combining static and dynamic methodologies) improves the recall (able to detect better the potential impact of a change).

Another change impact analysis research direction was followed by Bano *et al.* [54], who showed that software requirements are naturally changing and the changes in general are related to many causes. They performed a systematic literature review to identify these causes and their frequency on mono-language systems. They performed a coding process on the causes identified where they classified them into `Essential` and `Accidental` causes. Essential causes cannot be controlled by developers such as `Changing market demand`, however, accidental causes are the changes that can be avoided such as `Requirements not sufficiently specified and analyzed`. A part of our research is following this study to investigate the causes of changes in multi-language systems.

Other researchers were interested in another aspect of change impact analysis *i.e.,* software co-change analysis. Co-change considers sets of files that have been observed to change together to exhibit some form of logical coupling. Zimmerman *et al.* [47] and Ying *et al.* [55] proposed association rules to identify co-changing files. They suggested that co-changes may be important in recommending dependent entities that are prone to changes in the future. They identified co-changing files by the use of the co-change history in CVS. However, the authors did not discuss the dependency aspects between the identified co-changes. In chapter 6, we perform a co-change study on multi-language systems where we focused on comparing multi-language co-changes versus mono-language co-changes and their impact of the software security and quality.

Abdeen *et al.* [37] have found that predicting change impact by combining semantic and structural coupling information outperforms using one of them individually. They also found that semantic coupling produces better recall values compared to structural coupling metrics.

Furthermore, Jaafar *et al.* [56] presented a novel approach called *Macocha* to validate two change patterns when analyzing co-changes : the asynchronous change pattern, corresponding to macro co-changes (MC) and the dephased change pattern, corresponding to dephased macro co-changes (DC). They applied *Macocha* on seven mono-language systems, either Java or C/C++ source code.

All of the above-mentioned works exclusively focus on mono-languages. However, we came across a few studies that focused on change impact analysis in multi-language systems. An ongoing work by Angerer [57] is focused on developing a change impact analysis approach for multi-language software product lines (SPLs). His motivation is due to the limitations of existing change impact analysis for multi-language SPLs. His proposed approach is based on system dependency graphs (SDG). However, it is limited to SPLs and does not address the change in programming language sets. Deruelle *et al.* [58] have recommended a model for change impact analysis applied on multi-language databases applications. Their model is based on graph rewriting and it deals with centralized and distributed environments. They used a CORBA-based framework containing three different databases and environments to analyze and manage the changeability of multi-database applications.

Nguyen *et al.* [59], on the other hand, proposed a technique called Web-Slice to calculate program slices for dynamic and multi-language web applications. It is considered as a preventive solution for assisting developers in the analysis and comprehension of change impact in the multi-language web development context. Cossette *et al.* [60] argued that dependency analysis is very important in determining change impact in a multi-language software. They have presented the limitations of techniques available for supporting multi-language dependency analysis, which are mostly lexical or semantically-based. While semantic analyses are expensive to model, lexical analyses give poor accuracy and primarily depend on developers' skills to write appropriate pattern expressions. The authors suggested the use of island grammars in order to detect dependencies in multi-language software systems, same as Moonen [61]. Shatnawi *et al.* [62] proposed DeJEE, a tool that detect and build dependency graphs for J2EE applications. Their approach, limited to Servlets and JSPs, parses the source code into knowledge-discovery-model (KDM) to mine the dependencies within the J2EE applications. Hecht *et al.* [63] extracted declarative specification of the hidden dependencies of the J2EE applications not directly visible in the user code. They performed a codification of these dependencies into rules to make them automatically detectable using a software tool. This work was limited in proposing solution and did not investigate the impact of these hidden dependencies on the quality and the security of multi-language systems.

Additional research work focused on identifying and analysing in depth the difficulties and the impact of using multi-language development. Shatnawi *et al.* [8] analyzed the challenges that a multi-language system could face and that make static code analysis a hard task for the developer. They proposed a solution based on KDM (Knowledge Discovery Meta-model) that identified dependencies between different artifacts. Their study focused on Java container servers where they studied the case of server-side Java with client-side Web dialects (*e.g.,* JSP, JSF, etc.). Still, in the context of multi-language dependency analysis, Sayagh *et al.* [64] highlighted the challenge of identifying configuration options through a multi-layer software system. They were the first researchers to perform an empirical study toward identifying the configuration dependencies through multiple layers as configuration options in each layer might contradict each other. One of the main findings was that there is more indirect use of configuration options than direct use, and they concluded that the detection and fixing of configuration errors could become more difficult for developers.

2.2.3 Multi-language Development in Machine Learning Frameworks

Historically, traditional machine learning developers preferred dedicated programming languages such as Lisp or Prolog for their development. Nevertheless, in recent years, Python emerged as the most commonly used programming language for the development of Machine Learning frameworks as it offers a wide range of powerful and advanced features [65]. Yet, again the need arose of involving multi-language development in these frameworks because Python was observed to have some critical shortcomings, notably, its lack of efficient computational performance required for high-frequency real-time predictions [66]. Therefore, using the Python C extension is prevalent to overcome this shortcoming and as a solution to interface with highly performant C code for frequently executed low-level algorithms, such as required by the gaming industry [67], multi-agents [68], and so on. We report in the following related work on using machine learning frameworks within multi-language development.

Poggi *et al.* [69] presented HOMAGE, an environment for the development of multi-agent systems using three object-oriented programming languages *i.e.,* C++, Common Lisp, and Java. Tasharrofi *et al.* [70] developed a modular framework written in multiple languages. The use of multi-language development is also found in machine learning-based games. On the other hand, Phelps *et al.* [67] argued that multi-language development is propagating quickly proportionally with the arrival of more games leveraging AI, which leads to development challenges.

Other studies have been done over the past years to conduct research on incorporating traditional software engineering practices in the machine learning domain. Braiek *et al.* [65] explored the association, if any, between open source software and machine learning frameworks. Furthermore, they investigated the advantages and inconveniences of employing software engineering practices in machine learning frameworks. In Chapter 8, we discuss in depth 17 of the 20 largest open-source machine learning frameworks presented in their study in the context of analysing the impact of multi-language machine learning frameworks on the bug-proneness. Khomh *et al.* [71] highlighted yet another aspect of software engineering (SE) challenges for machine learning (ML) frameworks. They suggested a stronger and even deeper synergy between the communities of SE and ML in order to deal with these development challenges. Their study catalogues two primary challenges of integrating SE practices in ML frameworks including software testing and software evolution.

Dhasade *et al.* [72] presented the mono-language tool `Prioritizer` to assist developers in handling high volumes of issues in systems. They developed a machine learning based tool for prioritizing pull requests based on different criteria, such as issue life time, to fix these issues. They evaluated the efficiency of this tool by testing it on a data-set of 3000 issues. Similarly, Veen *et al.* [73] proposed a tool for pull request prioritization called “PRioritizer". This tool uses machine learning to work as a priority inbox for pull requests, recommending the top pull requests the project owner should focus on. Zhao *et al.* [74] addressed yet another aspect of given issue and suggested a learning-to-rank (LTR) methodology to recommend pull requests for quick reviews within mono-language development.

2.3 Chapter Summary

In this chapter, we presented some background information and simple examples (*e.g.,* Java Native Interface and Python-C-Extension) of multi-language source code in order to give an overview of multi-language development in practice. Furthermore, we provided an overview of literature on multi-language development in traditional systems and its adoption in machine learning frameworks.

CHAPTER 3 RESEARCH PROCESS AND ORGANIZATION OF THE THESIS

In this chapter, we present the research methodology and the structure of this dissertation. As previously mentioned in Chapter 1, this thesis aims to understand and analyze how multi-language systems are developed. Multi-language systems are systems designed with more than one programming language using specific techniques to create the bridge between the combined languages. To achieve the thesis goal, we will study two families of multi-language systems : traditional software systems and machine learning frameworks, as shown in Figure 1.1. As multi-language development appeared first within the traditional systems, studying them first presents an opportunity to understand later the adoption of the multi-language practice in more modern systems that use machine learning frameworks.

To achieve our goal, this thesis presents the following three main sub-goals that map to the three sub-hypotheses presented in our research hypothesis in Section 1.1. Chapters 4 and 5 belong to the first sub-hypothesis, Chapters 6 and 7 belong to the second sub-hypothesis, and Chapter 8 belong to the third sub-hypothesis.

1. We first need to understand the multi-language development phenomenon in traditional systems. Thus, we need to investigate the prevalence of multi-language development by identifying the different sets of programming languages used within systems, the techniques applied to ensure their interoperability, and the communication between the combined languages. We also investigate the practices followed to mitigate the challenges associated with multi-language development (language synchronisation) and enhance the benefits of this development paradigm. This is addressed in Chapter 4 and Chapter 5.
2. The second step consists of empirically evaluating the major challenge identified from the previous step *i.e.,* the change impact analysis in multi-language systems and its impact on the quality and security of software systems. We propose new approaches and recommendations to help developers with this in their multi-language development. This is addressed in Chapter 6 and Chapter 7.
3. Finally, we examine more modern and advanced multi-language systems. We analyze the adoption of multi-language development in machine learning frameworks and its impact within these frameworks in terms of acceptance rate of the software contributions (*i.e.,* pull requests) and the risks of bug introduction. Finally, we compare the findings with those of multi-language traditional systems. We address this part of the

thesis in Chapter 8.

We briefly outline the three different parts of this thesis below.

3.1 Part 1 : Why and how is multi-language development used in practice ? (Chapters 4 and 5)

Modern software is no longer developed in a single programming language. Instead, programmers tend to exploit the strengths of different programming languages, thus developing multi-language systems [19, 75]. As such, we observe an increase in the number of multi-language systems being developed, with most of the recent software systems today being heterogeneous *i.e.,* they are composed of components of different programming languages that interact with the host language. Developers also opt for this practice to keep using legacy implementations of their existing libraries and systems, while still benefiting from code reuse of modern software components [1, 76].

3.1.1 Investigating multi-language usage, the possible combined languages, and the techniques used

How prevalent is multi-language development ? In which context is it preferred over mono-language development ? What techniques are available to facilitate the interoperability of multiple languages ? In order to find answers to these important questions, we conduct a systematic literature review to highlight the state-of-the-art of multi-language development. A systematic literature review is an effective methodology to explore the new dimensions of multi-language development from researchers' perspective and also to facilitate identifying and discovering new facts.

We address four objectives : (1) understanding the motivation behind opting for multiple languages, (2) classifying contexts where multiple languages are used, (3) categorizing different combinations of programming languages used in multi-language systems, and (4) identifying procedures, *i.e.,* techniques that present a bridge to combine the languages, that allow for the integration of various programming languages.

One of our main results highlighted that Java/C(++) has been found as the most common combination of programming languages via the use of the Java Native Interface technique. We also found during our systematic literature review (Chapter 4) that change impact analysis is one of the main challenges of multi-language systems.

Both researchers and developers within the multi-language development community will benefit from the results of this study. They can use this literature review to further explore the diversity and complexity of multi-language frameworks. Furthermore, this study will empower them to propose more practical and feasible solutions to facilitate the integration of complex links and bridging techniques for multiple programming languages.

3.1.2 Studying the best practices in multi-language development

One of the major findings from the first study was that Java/C(++), through the JNI technique, is the most common combination of languages used within the multi-language context. JNI is the Foreign Function Interface that allows communication between Java and C/C++ code ; functions written in C(++) or assembly can be called within Java code, and vice-versa. It allows Java code running inside an instance of the Java Virtual Machine (JVM) to invoke native code to call Java code [27].

In this chapter, we identify the JNI usage and the practices followed in 100 open source multi-language systems collected from Open-Hub (with around 8k of source code files combined between Java and C/C++, including the Java class libraries part of the JDK v9). The JNI practices are identified by semi-automatically and manually analyzing the source code (the JNI part) following the data flow propagation between Java and C(++). This qualitative analysis identified 11 JNI practices that are primarily related to loading libraries, implementing native methods, exception management, return types, and management of local/global. Based on these practices, we suggest a set of recommendations to developers to help them in debugging JNI tasks.

This chapter presents an opportunity to thoroughly investigate the data flow propagation between languages (in the case of Java and C(++)) ; we studied the practices used by developers to ensure better communication and dependency synchronization between Java and C/C++ components.

3.2 Part 2 : How can we overcome the challenge of change impact analysis in multi-language development ? (Chapters 6 and 7)

The systematic literature review presented in Chapter 4 showed that change impact analysis in multi-language systems is one of the most discussed topics. An in-depth investigation showed that multi-language development posed a new challenge to the maintenance of software systems. The challenge comes from the fact that when a multi-language system is changed, we should take into consideration the different rules of each programming language (*e.g.,* source code syntax, data conversion, synchronization, exception management, etc,) that govern dependency analysis. Safe change impact analysis in the multi-language context allows to identify the propagation of a change between two or more programming languages. This kind of detection is necessary to ensure a safe change and keep the consistency between the source-code entities written in different programming languages.

3.2.1 Investigating the change impact analysis from developers' perspective

In this first study in Chapter 6, we aim to understand how developers manage their changes in multi-language systems. Using a technical survey comprising 30 questions, we first examine the developers' motivations for the change impact analysis. Next, we investigate how developers deal with changes in multi-language systems. In particular, we investigate the tools, techniques, and methodologies used by developers. We also inquire about the difficulties that they face during their analysis. We do not only investigate and record the challenges faced by the developers while performing their analysis, but also discuss the impact of a lack of change impact analysis on system quality and system security. We categorise the consequences of a lack of change impact analysis in multi-language systems and present recommendations to guide and help industries and developers in their change impact analysis of multi-language systems.

The survey was sent to 200 developers using the LinkedIn platform. It was completed by 69 different developers (34,5%), from eight countries, with diverse backgrounds and work positions. We also invited the participants for a 20-minute follow-up interview. The purpose of the interview was to ask more precise and interactive questions to make sure that they understood each question accurately and responded accordingly. Eight out of 69 participants agreed to participate in the interview.

From the survey and the interview, we find that dependency analysis is the most challenging problem within the change impact analysis of multi-language systems and there is a major need for an approach or tool to support such analyses. Surprisingly, we found that developers typically perform implicitly the change impact analysis before applying changes to multi-language systems. We found that they use testing methods without a deep analysis of the dependant component that may be impacted by the change. Most surprisingly, developers are aware of the importance of change analysis but they do not know how to implement it in the context of multi-language development, as the dependency analysis process is totally different from that of mono-language development. Furthermore, we identified that multi-language systems are more prone to be negatively impacted by bugs and vulnerabilities through changes. In general, developers lack of tool support to perform correctly change impact analysis within multi-language systems.

3.2.2 Investigating the inter-language dependencies in multi-language systems

Based on the main findings from our previous studies, we perform an empirical study on 10 Java Native Interface (JNI) open-source multi-language systems to identify the multi-language dependencies and their impact on software quality and security. Moreover, to answer the developers' needs detected in the technical survey study, we introduce two approaches, which we applied to the ten JNI systems :

— S-MLDA (Static Multi-language Dependency Analyzer) that performs a static dependency analysis using heuristics and naming conventions to detect direct multi-language dependencies between two multi-language files.

— H-MLDA (Historical Multi-language Dependency Analyzer) that performs historical dependency analysis based on software co-changes to identify the indirect multi-language dependencies that could not be detected by static analysis.

Our main results show that multi-language dependencies *i.e.,* dependencies between multi-language components involve a higher risk of introducing bugs and vulnerabilities to the software than dependencies between mono-language components. The percentage of bugs within multi-language dependencies is three times higher than within mono-language dependencies, and the percentage of vulnerabilities is two times higher in the case of multi-language dependencies.

3.3 Part 3 : What is the impact of multi-language development in machine learning vs. traditional systems ? (Chapter 8)

Machine Learning, a major sub-field of Artificial Intelligence, has introduced the automation concept. SE is yet another specialised field ready to use the machine learning (ML) techniques for solving diverse problems ranging from software development to maintenance [77].

Machine Learning frameworks are complex and require skilled developers for efficient use. Therefore, the SE community has been trying to implement user-oriented solutions in order to make ML applications simpler. The research community, for many years, has been focusing on multi-language development in this regard. The software developers have been leveraging the use of multiple languages in a single AI system. To be more precise, developers have been opting for the Python-C-Extension with highly performant C code for frequently executed low-level learning algorithms, as required, for example, by the gaming industry [67], multi-agents [68], and so on, instead of using python alone. In this domain, Python requires consequential CPU time for interpretation, and lacks efficient performance required for high-frequency real-time predictions [66].

As multi-language development presented new challenges in traditional systems, it is likely to present some challenges in machine learning frameworks as well. Therefore, Chapter 8 empirically analyzes the ten largest open source multi-language machine learning frameworks, comparing them with ten large open source traditional systems. Furthermore, we recorded the variation between challenges faced while applying multi-language in ML frameworks as compared to traditional systems. We analyze the correlation between the existence of multi-language development with machine learning. Finally, we analyse the impact of multi-language development on the acceptance rate of open-source code contributions and their relation to bug introduction.

The primary result of our analysis highlighted that multi-language presents a challenge for machine learning. Furthermore, it was also found that ML frameworks are more error-prone compared to traditional systems, therefore, multi-language machine learning software contributions take longer to be accepted than the traditional systems.

3.4 Chapter Summary

This chapter presented a detailed overview of the thesis and the link between the hypothesis and the respective chapters. We briefly described the different studies made during this thesis, their methodologies, and summaries of the major results.

CHAPTER 4 WHY AND HOW IS MULTI-LANGUAGE DEVELOPMENT USED IN PRACTICE ? (Sub-hypothesis 1)

4.1 Chapter Overview

Most modern systems are developed through the integration of multiple programming languages [78]. According to Tomassetti *et al.* [79], 96% of their 15000 studied GitHub systems use at least two different languages, with 50% using three or more. Rather than trying to solve all problems with a single language, developers adopt the languages that are best suited for their needs.

However, multi-language development presents different development challenges. Developers have to decide which programming languages best complement one another, determine the best mix of programming languages to adopt, consider what is the best existing mechanism or technique to use, and what are the best developing practices that they should follow. As we hope to support multi-language development users to deal with the complexity of such systems, we aim to highlight first the state of art of the multi-language development and then the state of the practices of the multi-language development.

We present, in this Chapter, a systematic literature review aimed at establishing the state-of-the-art of multi-language development and addressing the following objectives : (1) understanding the motivation behind the use of multi-language development in literature, (2) categorising the main topics and contexts of use of multi-language development in literature, (3) identifying the different sets of programming languages in literature, and (4) identifying the mechanisms used to link between the programming languages.

A Systematic Literature Review (SLR) is considered as an effective research methodology [17] to identify and discover new facts about a research area and to publish primary results to investigate specific research questions [80, 81]. This study aims to answer the following research questions :

RQ4.1. Are there articles that deal with multi-language development in literature?

RQ4.2. What are the different contexts of the use of multi-language development in literature?

RQ4.3. What are the different sets of programming languages used by researchers in the multi-language systems?

RQ4.4. What are the existing mechanisms used for the integration between the different programming languages in literature?

During this study, we followed Kitchenham *et al.* 's approach [17]. Figure 4.1 presents the different steps followed to collect and analyse the data.

4.2 SLR Design

We performed an SLR covering the multi-language development studies published from 2010 up to 2020. We performed an automated search on Engineering Village [1] for relevant multi-language development papers. Engineering Village is an information discovery platform that is connected to several trusted engineering electronic libraries. Specialized in engineering, it offers many options to refine the search queries, exclude and include criteria, offers the flexibility to choose the period of time, language, venues, and authors. This platform gives to users also the ability to search in all recognised journals, conferences, books, and workshop proceedings together with the same search query [82].

We assumed that the main keywords to create the search query are : **Multi-language**, **Software**, and **Analyse**. We used keywords, synonyms, and truncation to build our query in order to ensure a complete collection of papers. Following are the respective three main keywords combined with AND/OR in the search queries.

— "Multiple language", "Multiple languages", "Multi language", "Multi languages", "Multi-language*", "Mixed language", "Mixed languages", "Mixed-language*", "Heteregenous language", "Heteregenous languages", "Polylingual", and "Polyglot".

— Software : "Software*", "Program*", and "System*".

— Analysis :"Analys*" and "Analyz*".

Now, we present the search query and the different refinement steps applied to get the relevant desired input set of papers for our analysis. We searched for desired papers basing on the title, abstract and keywords. This option is expressed by *wn KY* in the Engineering Village.

The following is the general query that does not include any exclusion criteria. The query returned **3964** papers. In order to refine the output and get more relevant papers, we performed two exclusion process *i.e.,* one automatic and one manual analysis.

Query 1 :
((((((((((((((((("multiple languages" OR "multiple language" OR multi-language* OR "multi language" OR "multi languages" OR mixed-language* OR "mixed language" OR "mixed languages" OR "heterogeneous language" OR "heterogeneous languages" OR polylingual OR polyglot)wn KY AND (software* OR Program* OR Analys* OR System*)wn KY))))))))))))))))))

1. `https://www.engineeringvillage.com/search/quick.url`

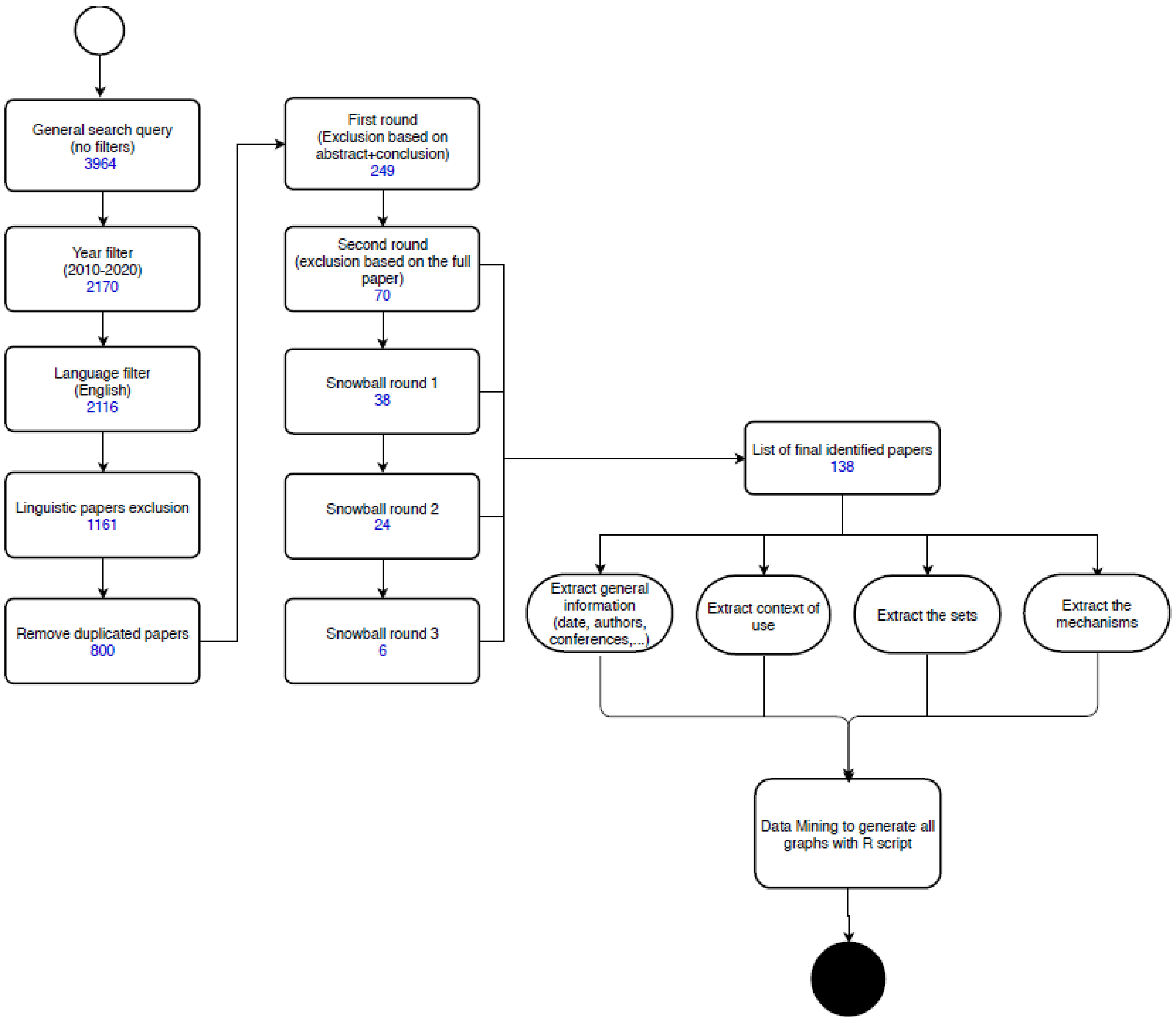


Figure 4.1 Overview of the literature review process

4.2.1 Automatic analysis

The first exclusion set contains four steps (presented below).

— First, we excluded papers published before 2010. The new query returned **2170** papers down from 3964.

Query 2 :
((((((((((((((((((("multiple languages" OR "multiple language" OR multi-language* OR "multi language" OR "multi languages" OR mixed-language* OR "mixed language" OR "mixed languages" OR "heterogeneous language" OR "heterogeneous languages" OR polylingual OR polyglot)wn KY AND (software* OR Program* OR Analys* OR System*)wn KY))))))))))))))) AND ((2020 OR 2019 OR 2018 OR 2017 OR 2016 OR 2015 OR 2014 OR 2013 OR 2012 OR 2011 OR 2010) wn YR))))

In this study, we limited the studied period to ten years (2010 until 2020). We believe that ten years is a good and sufficient period in which we can find the most modern and updated works. Especially, in the Software engineering field, technologies are moving quickly so referring to papers older than ten years will not provide up to date information.

— Second, we excluded papers not written in English. The query returned **2116** papers.

Query 3 :
((((((((((((((((((("multiple languages" OR "multiple language" OR multi-language* OR "multi language" OR "multi languages" OR mixed-language* OR "mixed language" OR "mixed languages" OR "heterogeneous language" OR "heterogeneous languages" OR polylingual OR polyglot)wn KY AND (software* OR Program* OR Analys* OR System*)wn KY))))))))))))))) AND ((2020 OR 2019 OR 2018 OR 2017 OR 2016 OR 2015 OR 2014 OR 2013 OR 2012 OR 2011 OR 2010) wn YR)) AND (english wn LA))

— Third, we observed that the `Multi-language` keyword causes confusion. It is used to describe the `Multi-language development field` but on the other hand, it also matches with the `Multi-language linguistic field`. From the obtained papers, we found that many papers turn around plagiarism, speech recognition, translation, and linguistic software tools. Thus, we used the options offered by Engineering village to exclude them : Classification code (presented by the keyword `wn CL`) and Controlled vocabulary (presented by the keyword `wn CV`). Engineering village groups the papers in different groups according to their main field, each group is presented by a code (*e.g.,* 751.5, 802.3, 932.1) as we see in the following search query. Hence, we excluded those papers considered as out of scope. The new query resulted in **1161** papers, down from 2116.

Query 4 :
(((((((((("multiple languages" OR "multiple language" OR multi-language* OR "multi language" OR "multi languages" OR mixed-language* OR "mixed language" OR "mixed languages" OR "heterogeneous language" OR "heterogeneous languages" OR polylingual OR polyglot)wn KY AND (software* OR Program* OR Analys* OR System*)wn KY) NOT (computational linguistics WN CV) NOT (cp OR ip OR ds) wn DT)) AND (english wn LA)) NOT ((751.5 OR 802.3 OR 932.1 OR 931.3 OR 801) wn CL)) AND (2020 OR 2019 OR 2018 OR 2017 OR 2016 OR 2015 OR 2014 OR 2013 OR 2012 OR 2011 OR 2010) wn YR)) NOT (linguistics OR language translation OR speech recognition OR ontology OR speech synthesis OR natural languages OR speech processing OR sentiment analysis OR teaching OR speaker recognition OR character recognition OR visual languages OR neural nets OR text detection OR behavioral research OR vocabulary OR knowledge representation OR emotion recognition) wn CV))

— Last, using the option provided by Engineering Village, we removed the duplicated papers within the three data-banks (Inspec, Compendex, and Knovel) used by Engineering Village. The output gave a total of **800** papers *i.e.,* 621 from Compendex, 177 from Inspec, and only two from Knovel.

4.2.2 Manual analysis

The second set of exclusions was dedicated to the manual analysis of the 800 papers previously obtained. Two researchers were involved in this set of exclusion to ensure a high confidence in the validity of the papers excluded.

— First, we analyzed, for each of the obtained papers, their title, abstract, and conclusion to eliminate the papers out of scope and to keep only the relevant ones. Papers considered out of scope are : papers not directly related to multi-language development ; papers that did not go in depth in analyzing the multi-language development, for example, papers that enumerated only examples of multi-language systems ; papers that integrated multi-language development in databases and services (did not involve the programming languages).
However, papers considered in scope are the papers that discussed multi-language development challenges, mechanisms used, multi-language tools used or developed or extended, papers that classify the combined programming languages, etc. Thus, we eliminated 551 and kept **249** to be analyzed in the next step.

— Second, we analysed the content of the papers obtained in the previous step, since

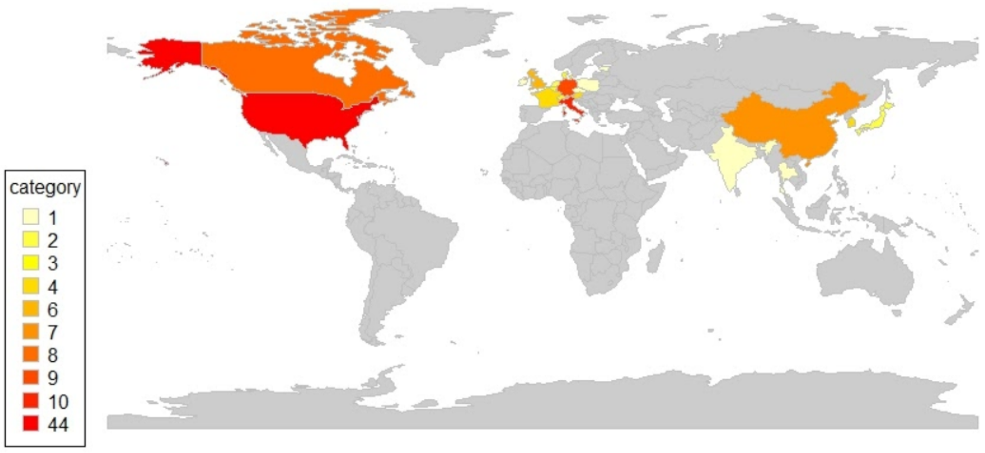


Figure 4.2 Author affiliations (countries)

using only the title, abstract, and conclusion does not provide enough information about the content of the paper when we want to investigate specific information. We went in depth in each paper, read it and understand its contribution in order to identify the most relevant ones needed for this study and also to extract the needed information for the RQs. At the end of this step, we kept **70** papers.

— Finally, we performed a snowballing search technique to ensure high recall. For this, we run through papers' references and identify, if any, papers that were missed by the search method/queries for any reason. We did three snowballing rounds based on the references of the 70 papers.

The first snowballing round returned **38** papers, the second round returned **24** where as the third one returned **six** multi-language development papers. We stopped the snowballing process at the third round as we obtained convergence in the results. As we see in figure 4.1, the four last steps resulted in a final list of **138** papers. We completely and deeply analyzed the 138 identified papers to extract the following information :

— General information : author affiliation, publication countries, publication years, and conference/journal names.
— Contexts of multi-language development use.
— Sets of combined programming languages.
— Mechanisms used to combine and provide the interface between the different programming languages.

Table 4.1 Number of papers published according to the respective conferences/journals

Conferences/Journals	Occurrences
PLDI	13
ICSE (full research papers)	11
SCAM	9
OOPSLA	6
SANER	6
PPPJ	6
ASE	5
IWST	5
WCRE	5
SIGAda	4
Europlop	2
DLS	2
ESEM	2
EASE	2
FSE	2
POPL	2

4.3 SLR results

We present now the SLR report step *i.e.,* the last step in Kitchenham' methodology [17], which consists of reporting the findings.

RQ4.1. Are there articles that deal with multi-language development in literature?

Motivation : The goal is to identify relevant multi-language development research papers. Based on that, we extract different information that characterizes the use of multi-language development nowadays *e.g.,* evolution in time, conference/journal publishers, etc.

Results : We found 138 relevant papers that deal with multi-language development in the literature. We report, in the following, general information about them.

— **The main researchers are from USA followed by Italy, Germany, and then Canada**. We present in Figure 4.2 a world map with countries colored by the number of papers published in multi-language development.

— **PLDI, ICSE, and SCAM are the main conferences where multi-language development papers are published**. We collected the main conferences/journals that published the multi-language development papers identified previously. It is considered as a helpful result as it will assists the researchers to target in

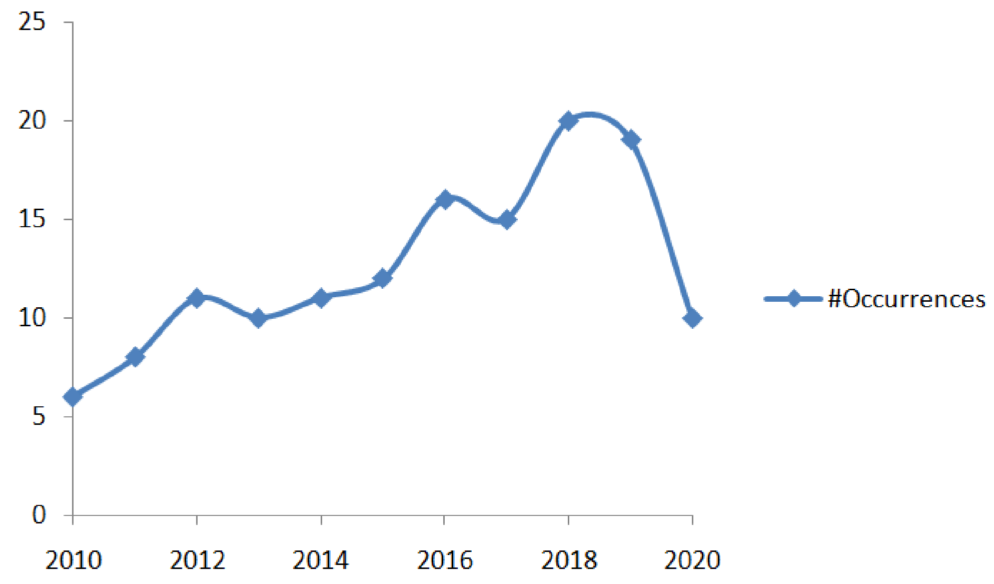


Figure 4.3 Multi-language development over time

fact, any software engineering conference/journal is suited. We present the results in Table 4.1. We found that 56 conferences/journals published one paper each, for the sake of the space, we did not add them in the Table.

— **The number of published multi-language development papers is increasing linearly over time**. We analysed the evolution over time of the identified papers to investigate how multi-language development were considered over years. We show in Figure 4.3, the evolution regarding the publication years. In figure 4.3, the results indicates that multi-language development domain is a growing research domain, however, the results for 2020 are misleading since it is incomplete year.

RQ4.2. What are the different contexts of the use of multi-language development in literature?

Motivation : We are interested in understanding when multi-language development should be used. Is there any specific situation where the use of multi-language development is necessary ? In which context, multi-language development presents an added value to the software system ? and are the most common challenging situations of multi-language systems ?

Results : "Software analysis" in multi-language software systems is the most commonly discussed topic in the literature where **"software changes" and "development practices" are the main challenging tasks.** We present in the following, the different steps followed for the coding. To ensure a high confidence of

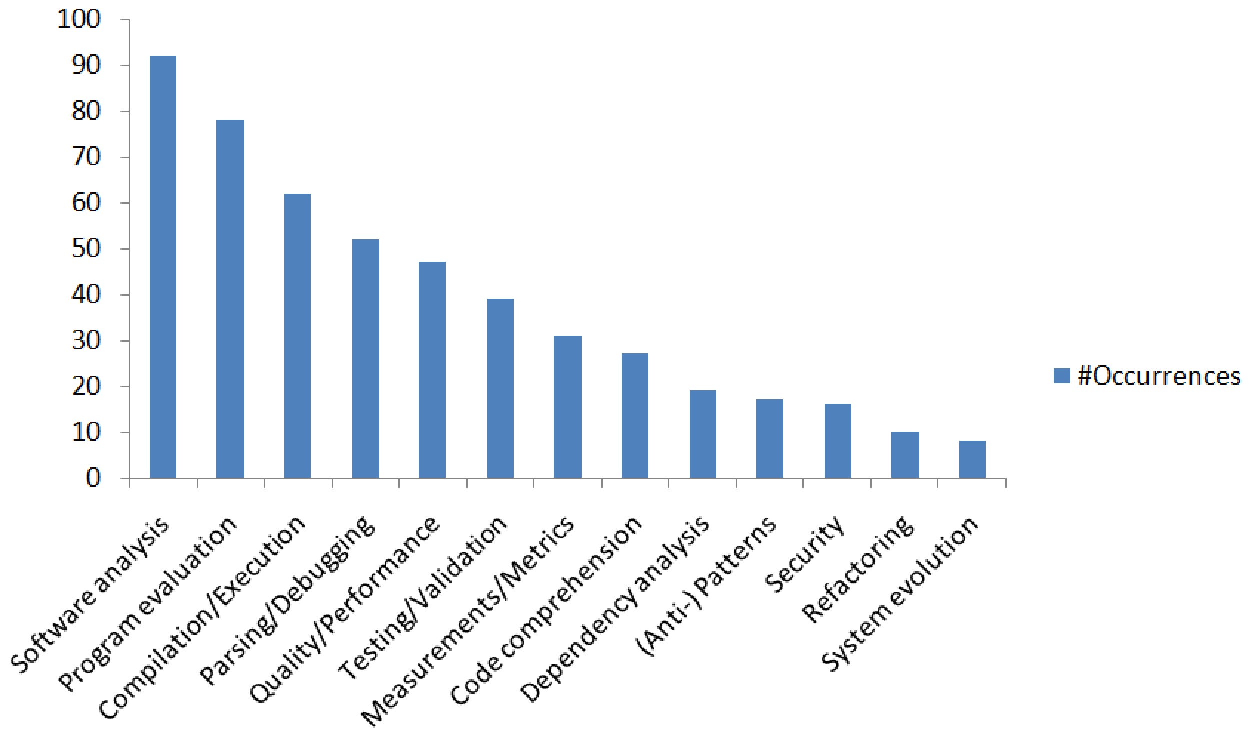


Figure 4.4 Main topic categories

the validity of coding, two researchers involved in this research work did the process and then contrasted the obtained results.

— First, we read all the papers to understand their contexts, objectives, and motivations.
— Then, we identify the keywords/sentences that could identify the multi-language development studied topics (*e.g.,* code analysis, compilation, tool evolution, metric measurement, debugging, etc.). On the other hand, we identify the papers that reported challenges and difficulties of multi-language development based on keywords such as : challenge, hard, difficult... to debug,test,develop, etc.
— Third, we perform a coding process based on the identified keywords to generalise the extracted keywords and create different topic categories. The process lasted two days and performed on three steps [18] : *Open Coding*, *Axial coding*, and *Selective coding*.
— Lastly, we verify the matching of the papers with their respective categories. During this step, a validation of the obtained categories was performed between the two researchers implicated in this coding process to reduce subjectivity of the coding methodology.

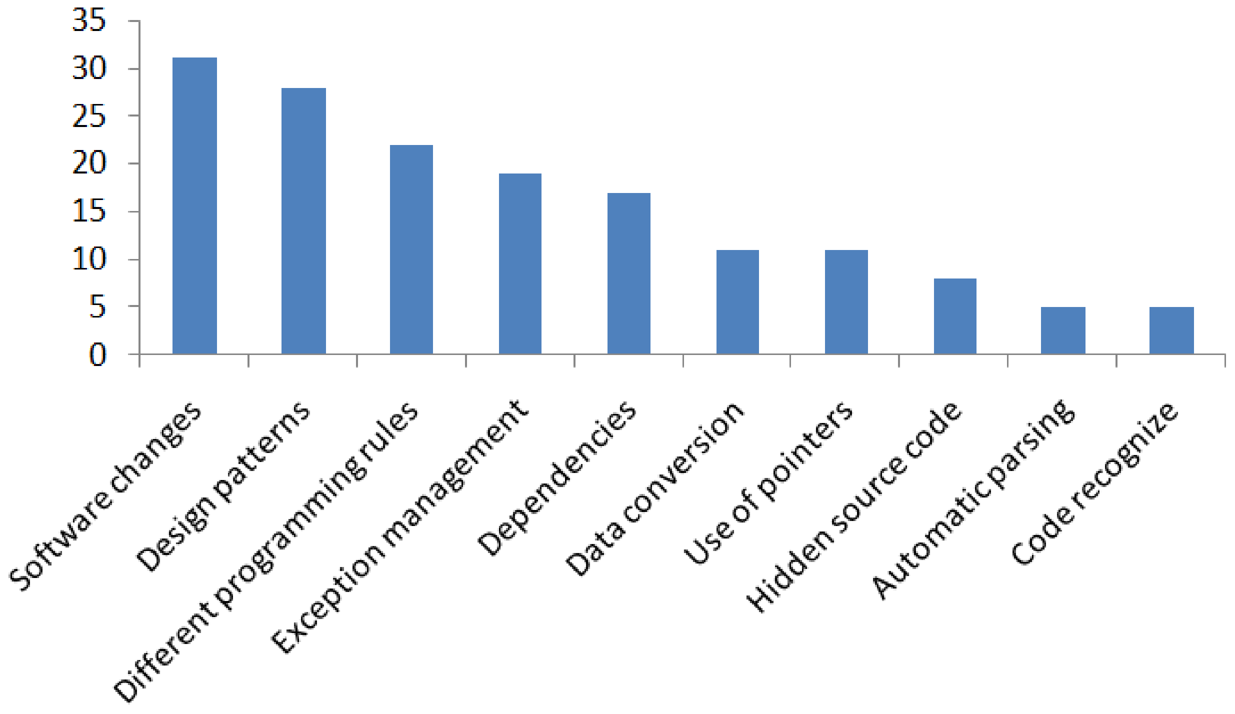


Figure 4.5 Major challenges of multi-language development

Figure 4.4 summarises the 13 general obtained categories. One paper can belong to more than one category. We found that "Software analysis in multi-language software systems" is the most common discussed topic in the literature. Researchers are preoccupied by proposing solutions and facilitate the analysis of multi-language systems. This category could contains any analysis method such as static analysis, dynamic analysis, slicing, graph-based, code review, etc.

Regarding the challenging situations, reported by researchers, that developers face during their development, we find that change impact analysis (reported in 31 papers) and design patterns/code smells (reported in 28 papers) are the major challenges of multi-language development as they are related to the core of software maintenance *i.e.,* they touch daily actions such as bug fix, new requirements to add, etc. Figure 4.5 shows the multi-language challenges as reported by researchers.

RQ4.3. What are the different sets of programming languages used by researchers in the multi-language systems?

Motivation : Multi-language development is the combination of two or more programming languages. Thus, we aim through this RQ to identify the existing combination of programming languages studied in the literature. All the multi-language deve-

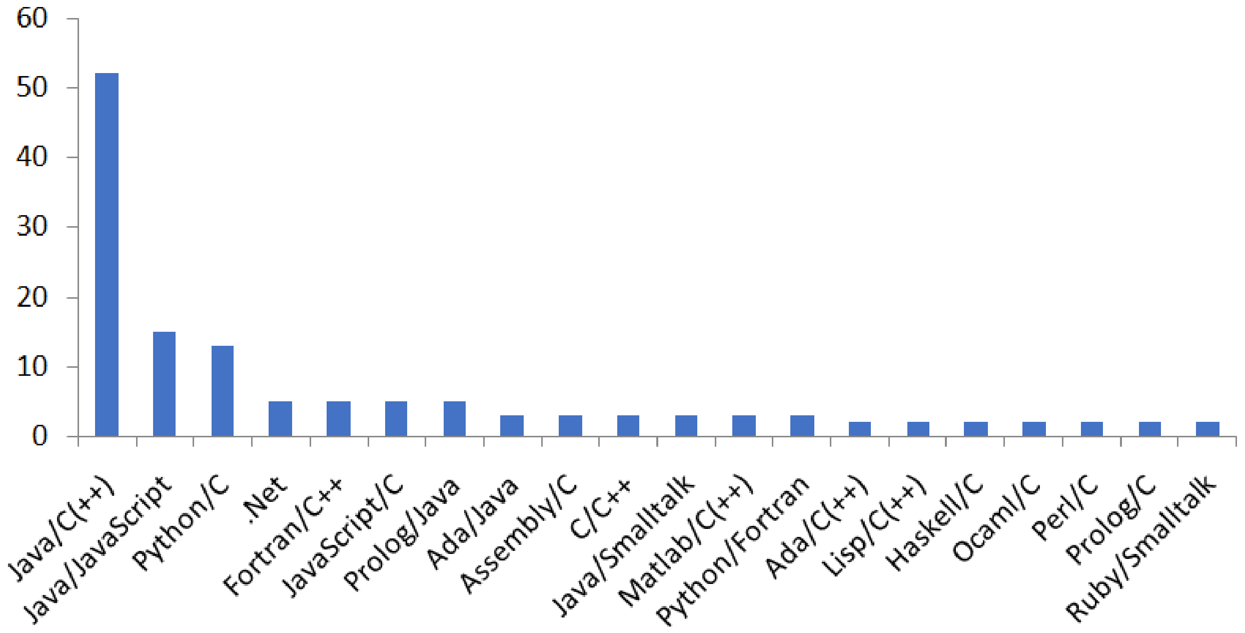


Figure 4.6 The 20 most common programming language combinations discussed in literature

lopment papers (138 papers) belonging to the 13 identified categories were analyzed to extract the information needed about the most common language combinations.

Results : The most used set in the literature is Java combined with C/C++, reported in 52 papers (37.68%). We found 60 sets (combinations of programming languages) that we present in figure 4.6. One paper can include more than one set. We limited the figure to the top 20 sets for clarity and visibility of the figure. The results show three main sets that are discussed the most in literature. The integration of Java with C/C++ takes the first place with 52 papers (37.68%). Java/JavaScript takes the second place in 15 papers, followed by Python/C, which was discussed in 13 papers. The remaining sets were presented in less than 5 papers (3,62%).

RQ4.4. What are the existing mechanisms used for the integration between the different programming languages in literature?

Motivation : Similar to any development methodology, integrating languages follows specific techniques or mechanisms that ensure the communication between languages. Our goal is to investigate the existence of such mechanisms and their prevalence in multi-language development.

Results : The most discussed mechanism is Java Native Interface (JNI) with 51 occurrences (36.95%). We identified from the papers the different multi-language development mechanisms discussed by the researchers. These mechanisms

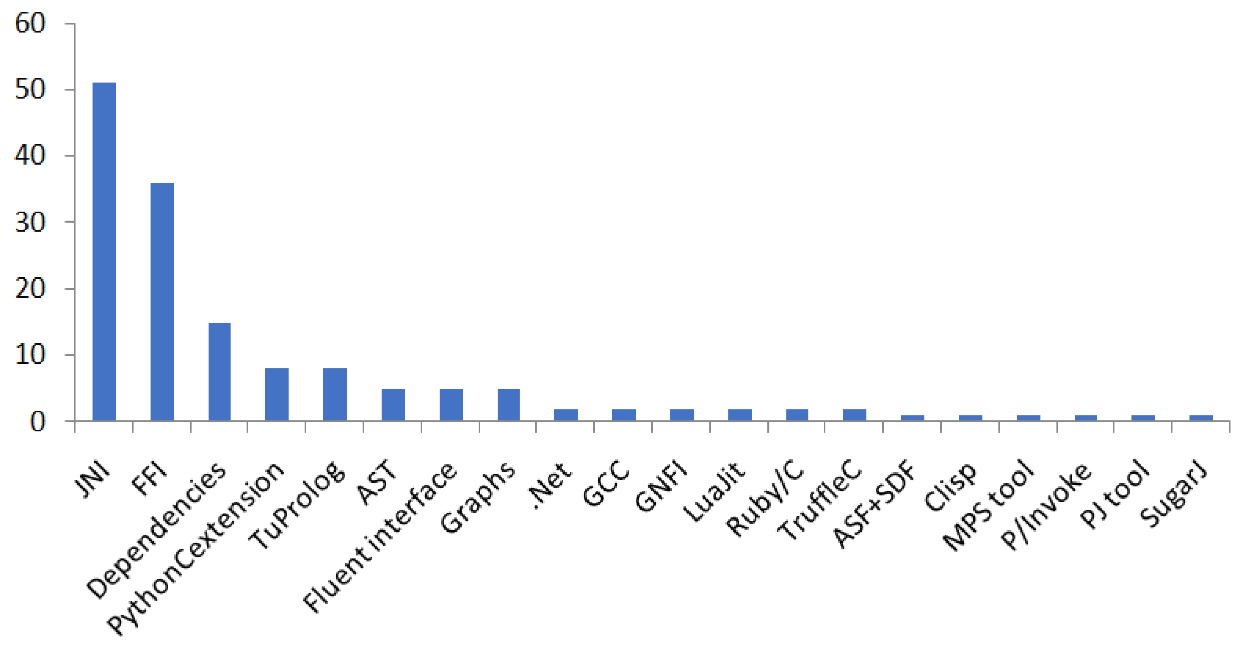


Figure 4.7 Techniques used for the integration of languages

could be :

— Techniques for helping developers within their multi-language development analysis such as : abstract syntax tree, graphs, etc.

— The existing interfaces to combine the languages *e.g.,* JNI, Python/c, etc.

— Existing tools already developed or being developed providing services for : testing methods, dependency analysis, quality metrics, debugging tasks, etc.

— New tools/algorithms being developed by researchers and discussed in their papers.

We present the results in Figure 4.7, limited to the top 20 mechanisms for the clarity and the simplicity for the reader. One paper can introduce more than one mechanism (method, tool, technique, etc.). We found that 32 papers (21.73%) did not report any details about the used mechanisms. In total, our analysis identified 90 mechanisms where the most used technique is Java Native Interface (JNI) with 51 occurrences (36.95%), which confirms the major results of RQ4.3 *i.e.,* the integration of Java with C/C++ is the most common set of languages used. Then, Foreign Function Interface (FFI) that takes the second place with 36 occurrences (26.08%). In these 36 papers, researchers discussed FFI in general without precising any specific types such as JNI or Python-C-extension. This, dependency analysis technique was reported in 15 papers (10.86%), followed by eight papers reporting the use of TuProlog tool and Python/C (5.79% each). Fluent interface, abstract syntax tree (AST), and graph techniques were reported by five papers each (3.62% each). GNU Compiler Collection (GCC), GNFI, LuaJIT tool, Microsoft .Net, RubyC, and TruffleC were presented by two papers each (1.44% each). The remaining mechanisms were discussed each in only one paper.

4.4 Discussion

Nowadays, researchers are becoming more interested in studying multi-language development, with several papers reporting its benefits and advantages. The inherent complexity and challenges associated with multi-language development has been a major motivation for further research in this field. Several other researchers are investigating multi-language development and proposing new solutions to help developers to overcome these challenges.

We observed from the systematic literature review that researchers are focusing most on the combination of Java/C(++). We explain it by the fact that the Java Native Interface (JNI) *i.e.,* the interface that allow the combination of Java and C(++), is already developed by the JDK team and available since 1996 [27]. Few years ago, Java and C++ were among the top five programming languages used in the world. At that time researchers started trying to make the combination between Java and C(++) more simple by providing better and more updated versions of JNI. Furthermore, Java is based on bytecode, thus, many languages have been ported on top to interact with a rich catalog of APIs for Java. On the other hand, JavaScript and Python are becoming the new trend and rank ahead of Java and C(++) terms of language popularity. We believe that focusing multi-language development on these new top programming languages will help the research community stay up to date with the industry trend.

Identifying multi-language development techniques and mechanisms was challenging. We found it difficult to identify them as many of the studied papers do not describe in detail their methodologies. We succeeded to identify the techniques used in some papers based on the documentation provided by the papers, references, and deeper investigation on the internet *i.e.,* developers blogs, GitHub, etc.

4.5 Threats to validity

Threats to construct validity : Threats that concern the design of the study. In this chapter, we presented a systematic literature review where the findings rely on our own evaluation on the papers collected and on our criteria used to exclude papers deemed out of scope. To ensure a high confidence in the validity of the papers excluded, two researchers involved in this research work did the process and then contract the obtained results. Then, the same two researchers did twice the process related to papers analysis and information extraction (Programming language sets, mechanisms, categories, etc). Regarding the validation of the categories presented in Section 4.1, we performed it on three steps. The first round was dedicated to reading in depth all the papers and trying to extract some relevant keywords that can help to understand the scope of the paper (Code analysis, Compilation, Software quality, etc.). Then, we generalised the keywords extracted and created categories. Finally, we validate the final results. To reduce subjectivity in our coding methodology, we validated it with two different researchers.

Threats to internal validity : We manually validated the inclusion and exclusion criteria of the papers selection in the systematic literature review study. We manually extracted the needed information from the 138 papers by reading and analysing them twice. We also performed many discussions with the researchers involved in the study to decide about the keywords used for the query search. We also identified from the literature the main challenges of multi-language development *i.e.,* change impact analysis and design practices. We assumed that these challenges are the challenges of developers as reported by researchers, but we also surveyed developers about that directly through a survey and an interview in Chapter 6.

Threats to external validity : Threats that concern the generalization of the results. During the systematic literature review, our methodology was firstly based on favouring Recall over the Precision to be sure that we had all papers and then from the obtained ones, we worked on the precision to keep only the relevant ones to the scope of the paper. We may forget or exclude some papers that discussed and analysed multi-language systems between 2010 to 2020. We accept this threat, as exclusion criteria vary depending from a researcher to other. However, to mitigate this threat, we did three snowballing rounds in which we collected papers that was not selected by the research queries because of the keywords used. We diversified the keywords, and their synonyms, used in the queries to cover as much as possible the published papers.

4.6 Chapter Summary

Multi-language development has become prevalent and despite its benefits, multi-language development introduces new challenges. In order to direct research efforts and help developers deal with the multi-language development challenges, we investigated through this chapter, the use of multi-language development in the literature. We performed a systematic literature review of the use of multi-language development.

Our analysis of 138 papers showed that the most commonly used set of programming languages in the literature is Java/C(++) combined through the Java Native Interface (JNI). One of the most important results concerns the identifications of the challenges that developers face during their daily multi-language development. Hence, we find that change impact analysis and the need for design patterns/code smells presents the main challenge of multi-language systems according to the literature.

CHAPTER 5 IDENTIFICATION OF THE PRACTICES OF JAVA NATIVE INTERFACE DEVELOPMENT (sub-hypothesis 1)

5.1 Chapter Overview

One key finding from the previous chapter was that the implementation of multi-language systems presents a huge challenge to developers. In this chapter, we analyze the practices used by developers of multi-language systems with the aim of identifying a set of best practices that can help developers overcome the challenges associated with multi-language development.

We perform a case study on 100 systems written in Java/C(C++) and using the Java Native Interface. These 100 systems, *i.e.,* 99 open-source GitHub software systems and the JDK v9 from OpenJDK, are from different application domains with different percentages of JNI native methods. We included the JDK because it is a large code base that includes many usages of JNI that has been written by the inventors of JNI, thus presumably an appropriate repository of good practices. We qualitatively studied those systems and catalogued the practices.

5.2 Study Design

This section details the design of our case study.

5.2.1 Data collection

We used OpenHub to collect open-source JNI systems. OpenHub provides reports about the composition and the activities of the systems. To create our data-set including the relevant open-source systems in GitHub, we used Python scripts to extract systems that contain Java and either C or C++ code. Then, for each software system, we calculated the number of JNI method declarations using `grep` (searching for the keyword "native") and followed by manual filtering. Based on that, we grouped the initial set into three groups : high, medium, and low number of JNI native method declarations and we randomly selected 99 JNI systems as follows :

— 20 projects with a number of native method declarations superior to 1,000

— 39 projects between 1,000 and 100

— 40 with a number of native methods between 100 and 10

We ignored systems with a number of native methods lower than ten, because our manual filtering showed that these methods are very short, usually delegators. Thus, we obtained a representative set of JNI open-source systems.

We also considered the source code of JDK v9 from OpenJDK. We decided to analyze it in addition to the 99 collected systems since it is developed and maintained by the implementors of JNI as it appeared firstly with the JDK v1.1 [1, 83]. We believe that best practices of JNI may have been implemented in that system. We downloaded the OpenJDK source code of the JDK v9 ; the reason behind this choice resides in the fact that it is the official implementation reference of Java SE since version seven [1].

5.2.2 Data analysis

First, we measured the main metrics of the collected software systems to have an initial overview of our data-set. We measured, using the Understand tool [2], the total number of methods (NOM) and the total number of code lines (LOC). We also identified and counted metrics related to the quantity of JNI in the systems *i.e.,* the number of JNI method declarations and number of JNI implementation functions using `grep` commands. Then, we manually validated the output and cleaned the data. Table 5.1 presents the metrics of top-20 studied JNI systems as an example. Then, using a `Python` script, we extracted JNI implementation functions from C/C++ files. These are functions starting with *JAVA_* or *JAVACRITICAL_*. We manually checked the implementation function names to validate the output. Identifying the native method declarations with their respective JNI implementation functions is a necessary step to facilitate the analysis process for the practices extraction.

These steps helped to understand the entire functionality of the systems and the implication of the JNI on them. We extracted JNI components *i.e.,* JNI method declarations and JNI implementation functions, to understand the coupling between them and how the data is propagated in order to identify developers' habits.

5.2.3 Practices identification

Once the JNI software systems are collected, we investigated the practices followed by developers when they use the Java Native Interface. The goal is to document and establish *common* and *recurring* practices of JNI usage to help and guide future JNI developers.

A best practice is a practice that respects the JNI specifications, the programming language rules to ensure the software quality, the design patterns, that was used in JDK, and potentially

1. `http://jdk.java.net/java-se-ri/7`
2. `https://scitools.com/`

Table 5.1 Metrics of the top 20 of the collected JNI systems

JNI systems	JNI methods (Declaration + Implementation)	LOC	NOM
libgdx	8417	577,986	60,248
JDK	2789	1,925,782	147,098
Google toolkit	2755	729,918	78,898
Openj9	1771	1,050,418	77,251
Rocksdb	839	209,054	18,205
JMonkeyEng	720	208,909	20,411
OpenVRML	324	653,777	5,256
Realm	314	83,426	8,162
Conscrypt	276	49,993	5,067
JavaSMT	237	20,460	2,641
Jna	213	97,161	8,4617
CeylonComp	160	268,853	27,581
ReactNative	155	79,283	8,394
Telegram	114	525,663	27,9
OpenCV	93	745,57	60,612
Tenserflow	81	460,926	29,811
JatoVM	64	59,384	4,964
SQLite	54	19,297	2,463
Frostwire	45	161,630	16,810
Godot	23	867,822	53,819

used in most of the collected JNI software systems. A practice could be any methodology of the implementation of JNI functions, a recurrent or a common JNI piece of code, a recurrent security check, a common use of specific attributes, etc. We report a practice when it was identified in ten or more of the JNI systems. We present in the following the steps followed to identify the practices and verify their usage in the collected JNI software systems. Figure 5.1 shows a general overview of these steps.

— First, we relied on the literature review study and on the JNI programming rules [83] to identify the essential and critical categories of JNI (*e.g.,* JNI library loading, exception management, JNI return complex types, etc,). We used these categories to limit and guide our process in searching for JNI programming practices first in JDK and then in the 99 open-source JNI software systems. The categories identified and used in this study are : JNI library loading, exceptions management, return types management, local and global references management, and String uses.

— Then, for each category identified from the previous step, we analyzed the source

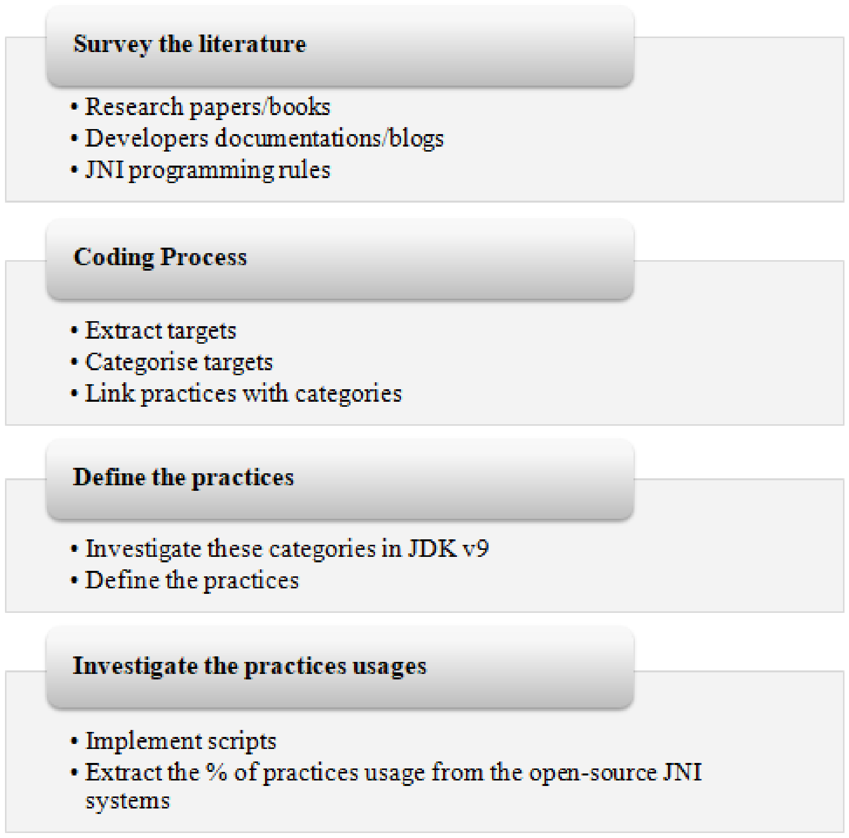

Survey the literature
• Research papers/books
• Developers documentations/blogs
• JNI programming rules
Coding Process
• Extract targets
• Categorise targets
• Link practices with categories
Define the practices
• Investigate these categories in JDK v9
• Define the practices
Investigate the practices usages
• Implement scripts
• Extract the % of practices usage from the open-source JNI systems

Figure 5.1 An overview of the steps followed to identify the JNI practices

code of JDK v9, considered as our pilot, to find how these categories are implemented and used. We investigated mainly the JNI part *i.e.,* JNI methods declaration in the Java side and their respective JNI implementation functions in the C side. Based on that, we defined the practices that we will investigate their usage in the other 99 JNI software systems.

— Lastly, we investigated the presence of JDK practices in the others systems. We identified the systems that are/aren't following JDK's development ways and that are/aren't following JNI specification rules in general.

For example, regarding the category of JNI library loading, we extracted the respective pieces of code in JDK, then we identified the involved keywords such as : static, system.load(), etc. Last, using scripts, we searched for these keywords in the other 99 JNI software systems and compared how the native libraries are loaded there. Each category was analyzed differently. For example, in the case of JNI exception management, we had to identify the data propagation of the exception object from both the Java and C side and to verify if the same implementation used in the other systems.

5.3 Study Results : Catalogue of identified practices

By analyzing the 100 JNI systems source code, we extracted 10 best practices and habits of JNI developers. These practices were identified based on the common JNI mistakes as defined in the bug reports. We found that six of these practices have been mentioned also in the IBM developers blog [3] which provides evidence that we followed an accurate approach to identify the best practices to support developers. This set of 10 practices are the first of a set of practices that can be extracted from the Java and C/C++ source code. We believe that developers should be aware of them and try to follow these common guidelines and practices when using JNI. In future work, we plan to enforce these practices with tools/IDEs to provide developers with an adequate JNI programming environment.

1. Practice Name : `AccessController to load a native library`.

   `AccessController` presents a safe way to load a native library because it ensures that the library cannot be loaded without permissions. We analyzed the source code of the JDK and observed that libraries are loaded in static blocks, wrapped in a call to `AccessController.doPrivileged` as shown in Figure 5.2. JDK developers use `doPrivileged` to ensure security and prevent undesirable access to the system [4] [1]. Similarly, we analyzed the source code of the other systems to look for *System.loadLibrary*. We found that 85 out of the 99 systems, *i.e.,* 85%, do not use the safe way to load libraries.

2. Practice Name : `No hard-coded library name`.

   Java is written to be used everywhere *i.e.,* the same code is expected to run on all platforms but there should be different native code libraries for different platforms that have to be loaded according to the target OS. Loading the library in a way to take care of the OS ensures that all platforms are covered and those missing libraries can be easily identified. This practice was ignored by 34% of JNI systems that we studied. We show in Figure 5.3 an example extracted from `Frostwire` system where it follows this good practice.

3. Practice Name : `Hiding Library Loading`.

   To load the native libraries, depending upon how the code is being compiled, developers should create a class to load the correct library. Then, they should call it from

3. `https://developer.ibm.com/technologies/java/articles/j-jni/`

4. `http://download.java.net/jdk7/archive/b123/docs/api/java/security/AccessController.html`

```
static { AccessController.doPrivileged(
         new PrivilegedAction<Void>() {
         public Void run() {
         System.loadLibrary("osxsecurity");
         return null; }
         }  ); }
```

Figure 5.2 Safe loading library (OpenJ9) [1]

```
/*for Windows*/
if (OSUtils.isWindows() && OSUtils.isGoodWindows()) {
if (OSUtils.isMachineX64()) {
System.loadLibrary("SystemUtilitiesX64");
} else { System.loadLibrary("SystemUtilities");}
/*for Mac OS*/
public final class GURLHandler {
System.loadLibrary("GURLLeopard");
public class MacOSXUtils {
System.loadLibrary("MacOSXUtilsLeopard"); }
```

Figure 5.3 Load library for different OS (Frostwire) [1]

another class to hide the actual library loading as in the case of the `Conscrypt` system shown in Figure 5.4. We found that 13% of JNI systems are not following this practice.

4. Practice Name : `Using Relative Path`.

   In JNI software systems we usually need to access or integrate foreign libraries or API. For that, when using a native library, we need to specify the relative path to access the library and allow it to be installed anywhere. We found that 32 of the studied systems load the external library by only specifying the name of the library without providing the full path, which can impact the reuse of code or maintenance of the library. We show in Figure 5.5 an example of the right way to perform it.

5. Practice Name : `Use List of Probable Paths`.

   When we deal with different libraries and operating systems, a list of probable loading paths can be generated based on the OS being used as illustrated by `JNA` system. This is a good practice when the native library is in a JAR, however, it seems that this approach is not used by the majority of the developers [5]. Figure 5.6 shows an example of the JNA system that creates a list of probable paths where the library can be found.

5. `https://www.adamheinrich.com/blog/2012/12/how-toload-native-jni-library-from-jar`

```
class NativeCryptoJni {
public static void init() {
  System.loadLibrary("gmscore");
  System.loadLibrary("conscrypt_gmscore_jni");
  System.loadLibrary("conscrypt_jni");}}

class NativeCrypto {
public static void main(String[] args) {
   ...
   NativeCryptoJni.init();  ...  }}
```

Figure 5.4 Abstracting library loading (Conscrypt) [1]

```
public class JNITest extends TestCase {
 static {System.load("./test/functional/jni/ libjnitest .so"); }
```

Figure 5.5 Using relative path (JatoVm) [1]

We found that 62% of the studied JNI systems are not following this practice.

6. Practice Name : `Assuming safe JNI return values`.

   Exceptions allow developers to report and handle exceptional events that require special processing outside the actual flow of an application. However, their support is not available for C programming language. JNI is designed to be a universal solution to facilitate the integration of native modules into Java applications, hence it uses exceptions, but in the C-way. Typically, we need to access and transfer data and information between different languages. We usually pass and return values from one language to another, the JNI API functions rely on their return values instead of indicating any errors during the execution of the API call.

   As a good practice, developers should never assume that it is safe to use a value returned by a JNI API call, it should always be checked to make sure that the JNI API call was successfully executed and the proper usable value is returned to the native function. Figure 5.7 shows a good example of the code extracted from Libgdx where Figure 5.8 shows a counter-example. Our results show that 40% of the studied JNI systems do not follow this practice.

7. Practice Name : `Take care of the strings`.

   JNI handles Java strings as reference types. These reference types are not null-terminated C char arrays (C strings). JNI provides the necessary functions to convert such Java

```
private static NativeLibrary loadLibrary(...) {
boolean isAbsolutePath = new File(libraryName).isAbsolute();
List<String> searchPath = new ArrayList<String>();
int openFlags = openFlags(options);
String webstartPath = Native.getWebStartLibraryPath(libraryName);
if (webstartPath != null) {if (Native.DEBUG_LOAD) { System.out.println("Adding web start
    path " + webstartPath);} searchPath.add(webstartPath);}
List<String> customPaths = searchPaths.get(libraryName);
if (customPaths != null) { synchronized (customPaths) { searchPath.addAll(0,
    customPaths);}}
if (Native.DEBUG_LOAD) {
System.out.println("Adding paths from jna.library.path: " +
    System.getProperty("jna.library.path"));}
searchPath.addAll(initPaths("jna.library.path"));
String libraryPath = findLibraryPath(libraryName, searchPath);
```

Figure 5.6 Use list of probable paths (JNA) [1]

```
jclass clazz;
clazz = env->FindClass("java/lang/String");
if (0 == clazz) { /* Class could not be found. */
} else { /* Class is found, we can use the return value. */}
```

Figure 5.7 Assuming safe return value of JNI (libgdx) [1]

string references to C strings and vice-versa, as shown in Figure 5.9. When a Java string is converted to a C string, it becomes simply a pointer to a null-terminated character array. It is the developers' responsibility to explicitly release the arrays, allocated on heap, using the `ReleaseString` or `ReleaseStringUTF` functions, as shown in Figure 5.10. Memory leaks can occur if the developers forget to do so. Our results show that only 8% of the studied systems are not following this practice.

8. Practice Name : `Never Cache Local References`.

   The lifespan of a local reference is limited to the native method itself. The JVM garbage collector's boundaries are limited to the Java space only, so the JVM garbage collector cannot free the memory that the application allocates in the native space. It is the developers' responsibility to manage properly the application memory in the native space. Otherwise, the application will cause memory leaks. To reuse a reference, the developer must explicitly create a global reference based on the local one using the *NewGlobalRef* JNI API call, as shown in Figure 5.11. The global reference can be

```
staticvoid  nativeClassInitBuffer (JNIEnv ∗_env){
 jclass  nioAccessClassLocal= _env−>FindClass("java/nio/NIOAccess");
 nioAccessClass = (jclass) _env−>NewGlobalRef(nioAccessClassLocal);
 bufferClass  = (jclass) _env−>NewGlobalRef(bufferClassLocal);
 positionID = _env−>GetFieldID(bufferClass, "position", "I");
```

Figure 5.8 Assuming safe return value of JNI (Libgdx) [1]

```
jboolean isCopy;
str  = env−>GetStringUTFChars(javaString, &isCopy);
if  (0 != str)  {...}
```

Figure 5.9 Converting a Java String into a C String (Telegram) [1]

released when it is no longer needed using the `DeleteGlobalRef` function. Only 22 JNI software systems are not following this practice.

9. Practice Name : `Always Check for Java Exceptions caught in native code`.

   Exceptions behave differently in the JNI than they do in Java. In Java, when an exception is thrown, the virtual machine stops the execution of the code and goes through the call stack in reverse order to find an exception handler that can handle the specific exception type. The VM clears the exception and transfers the control to the exception handler block.

   In contrast, JNI requires developers to explicitly implement the exception handling flow after an exception has occurred. Java exceptions can be caught in native code using the JNI API call *ExceptionOccurred*. This function queries the JVM for any pending exception, and it returns a local reference to the exception Java object, as shown in Figure 5.12. It will not block the execution of the native code. As the actual exception does not leave any traces behind, it is hard to debug.

   Checking whether a Java exception has been thrown is considered a good practice by the developers. Developers must check whether a Java exception has been thrown after invoking any Java methods that may throw an exception. Upon handling the exception, it should be cleared using the `ExceptionClear` function to inform the JVM that the exception is handled and JNI can resume serving requests to Java space. 80% of the JNI systems do not follow this practice, Libgdx is an example of that.

10. Practice name : `Use of proper caching of classes, methods, and field IDs`.

    JNI does not expose the fields and methods of Java classes directly in the native code.

```
str = env−>GetStringUTFChars(javaString, &isCopy);
if (0 != str) {env−>ReleaseStringUTFChars(javaString, str);
str = 0; }
```

Figure 5.10 Releasing the C string (Telegram) [1]

```
jobject globalObject = env−>NewGlobalRef(...);
if (0 != globalObject) {/*we can cache and reuse globalObject*/}
```

Figure 5.11 Obtaining a global reference from a local reference [1]

It provides a set of APIs to access them. For example to get the value of a field of a class : we can (1) obtain a reference to the class object through the `FindClass` function, (2) obtain the ID for the field that will be accessed through the `GetFieldIDfunction`, (3) obtain the actual value of the field by supplying the class instance and the field ID to the `Get<Type>Field` function.

These functions go upon the inheritance of classes and methods to identify the right ID to return. Neither the Class object, the Class inheritance, nor the fieldID can be changed during the execution of the system. These values are cached in the native layer for subsequent accesses. The return type of the `FindClass` function is a local reference so developers must create a global reference first through the `NewGlobalRef` function, when it is needed. The return value of `GetFieldID` is `jfieldID`, which is an integer that can be cached as it is.

As a good practice, developers should focus on caching both the field Ids and method IDs that are accessed multiple times during the execution of the application. This practice improves the execution time of the system. Caching classes, methods, and fields present an impact on the application's run time. We found that all the JNI systems that we analyzed are following this practice. Figure 5.13 shows an example of this practice extracted from JatoVM.

5.4 Discussion

Table 5.2 summarizes the lessons learned from the practices qualitative study to support developers within their JNI development, while in the following, we discuss our findings.

First, we investigated the safe library loading and we found that JDK developers use `AccessController.doPrivileged` to load the native library. However, in the other 99 systems, we found that

```
env−>CallVoidMethod(instance, throwingMethodId);
ex = env−>ExceptionOccurred(env);
if (0 != ex) { env−>ExceptionClear(env); }
```

Figure 5.12 Native exceptions using ExceptionOccurred [1]

```
static jfieldID JNI_GetFieldID(JNIEnv ∗env, jclass clazz,...)
   { fb = vm_jni_common_get_field_id(clazz, name, sig);
         if (!fb) {return NULL;}
         if (vm_field_is_static(fb)) {return NULL;}
         return fb; }
```

Figure 5.13 Use proper way of caching FieldIDs (Jatovm) [1]

85 out of 99 do not use the same way to load their libraries *i.e.,* `AccessController.doPrivileged`. Thus, we investigated 40 StackOverflow Q&A entries and we found that most of the developers use unsafe library loading as well. This practice faces software systems to more vulnerabilities *i.e.,* security breaches from outside[6]. The `AccessController` class is used to : (i) determine whether access to a critical system resource should be allowed or denied, (ii) mark the code as “privileged” thereby affecting subsequent access determinations, and (iii) obtain a “snapshot” of the current call context so that the access control decisions from a different context can be made with respect to the recorded one. We believe that developers must be aware of this practice to ensure safe use of the JNI and avoid vulnerabilities.

Regarding the software exceptions, the languages do not have the same way to manage exceptions as it depend on the language itself. Thus, in the context of multi-language systems, we can not rely only on the exception provided by the other language, it is necessary to implement the exception handling. Same in the case of JNI, developers have to explicitly implement the exception handling flow when an exception occurred. A mis-handling on JNI exception may result in software issues (quality and security issues) as unchecked exceptions will introduce faults in the system that will be hardly debugged or retraced to the origin of the bug.

Another practice concerns the check of the JNI Return Values where we suggest to developers to never assume that it is safe to use a return value without any additional verification. As well, developers should always check the return values to make sure that JNI API call was

6. `https://docs.oracle.com/javase/7/docs/api/java/security/AccessController.html`

Table 5.2 Lessons learned

	Lessons learned	**Brief description**
1	*Be aware of library loading*	Developers should be aware of the security of the system when loading native libraries. We recommend the use of *AccessController* for a safe load.
2	*Define the library for each OS*	Developers have to use a clean way to load the library by handling all targeted OS on which the library is available to ensures the code readability by making the libraries easily defined for each operating system.
3	*Hide the library loading*	Developers are recommended to abstract the access to the native library loading by hiding its implementation in a separate class and then call it through that class.
4	*Use relative Path to load the library*	In order to ensure the reusability and maintainability, developers are recommended to use relative path when loading the native library as it will easily located by maintainers and accessible from anywhere.
5	*Generate a list of library probable paths*	It is better to generate a list of probable loading paths of the native libraries When we manage different libraries through different operating systems.
6	*Always check JNI return values*	Never assume it is safe to use the return value of a JNI API call as is. Developers should always check the return value to make sure that the JNI API call was successfully executed and the proper usable value is returned to the native function.
7	*Take care of the Strings*	JNI handles Java strings as reference types that are not directly usable as native C strings, so JNI provides the necessary functions to convert these Java string references to C strings and back.
8	*Never cache local reference*	Developers should use a global reference, but they should take care of the memory.
9	*Check Java Exceptions*	JNI requires developers to explicitly implement the exception handling flow after an exception has occurred.
10	*Use Proper way of caching classes, methods, and fields*	JNI does not expose the fields and methods of Java classes directly in the native code. Instead, it provides a set of APIs to access them indirectly.

successfully executed and that the native code will receive the correct return value. Developers also need to take care of strings as they are converted from a pointer to a character array. These references will keep Java objects from being garbage collected. As good practice, we believe that it is the developers' responsibility to release the character arrays explicitly using the `ReleaseString` or `ReleaseStringUTF` functions to avoid issues related to memory leaks because JNI cannot manage the memory allocation automatically. From another survey study performed in [84], we found that checking multi-language return values, even if it is mostly followed by developers, is not given the same degree of a priority than checking the exceptions. Also, the survey reported that only 27.96% of the surveyed developers check multi-language return values very often, while the others reported that even they did not check these return values, they rarely experienced issues related to this practice.

The next interesting finding is about local and global reference management in JNI. Local references become invalid when the execution returns from the native method in which the local reference is created. Therefore, a native method must not store a local reference and expect to reuse it in subsequent invocations. So whenever the state is to be maintained during JNI calls, global references are a must where JNI global references are prone to memory leaks because they are not automatically garbage collected, and the developer should explicitly free them.

5.5 Threats to validity

Threats to construct validity : We ensured a diversity of systems where we analyzed the source code of the JDK v9 and 99 JNI systems and downloaded the source code of JNI systems from GitHub and the JDK from the OpenJDK Web site.

Threats to internal validity : We manually validated our choice of the JDK, the 100 JNI systems, our methodology, and tools. We believe that the JDK is an example of good JNI usage. We choose the JDK v9 as it is the last stable version of the JDK. We also analyzed 99 open source JNI systems collected from GitHub. We use the Understand tool, Git commands, and python scripts to identify JNI method declarations and implementation functions. We semi-automatically validated the results of our scripts.

Threats to external validity : We identified 10 common JNI practices. We believe that these practices are some of many other practices that can be found by analyzing more systems. On the other hand, while we cannot generalize our results to all JNI systems, the choice of major categories of the JNI development and the use of JDK as a study pilot provides confidence that the identified practices may be applied on many other JNI systems. Also, we mitigate the threat by evaluating the manual analysis by two of the implicated researchers to ensure a high accuracy.

5.6 Chapter Summary

From the previous chapter, we concluded that JNI (a sub-type of FFI) is the most used and studied technique in literature. Researchers are interested in analysing the combination of Java with C(++) in different contexts such as JNI practices [1, 78] and JNI challenges [85]. In this chapter, we further investigate the Java Native Interface from the developer's perspectives. JNI proposes structures and an idiomatic syntax to declare, implement and call native methods. However, the actual state of the practice of JNI usage was unknown so far. Thus, we performed a qualitative analysis to extract facts about the usage of JNI in the source code of 100 systems (around 8k Java and C(++) files) that use JNI from GitHub. Our main results show 10 JNI practices mainly related to library loading, exceptions management, return types management, local and global references management, and String uses.

CHAPTER 6 HOW IS CHANGE IMPACT ANALYSIS PERFORMED IN THE CONTEXT OF MULTI-LANGUAGE DEVELOPMENT ? (Sub-hypothesis 2)

6.1 Chapter Overview

The systematic literature review conducted in chapter 4 identifies change impact analysis as one of the main challenges faced by developers during multi-language development. Thus in this chapter, we analyze in depth the concept of change impact analysis with the goal of helping the multi-language developers to overcome this challenge.

Change-impact analysis (CIA) is the technique to analyze the impact of changes on source-code entities. It is an essential technique for software maintenance and evolution, as it computes the impact associated with any kind of changes, as well as their consequences [86], *i.e.,* the source-code entities that must be changed after a bug fix, a client's new requirement, a migration to a new technology, etc. When a change is applied to a system, the dependency between its entities may be broken *i.e.,* one of the entities may risk being no longer coherent with others or the new change may require further changes to some entities on which it depends [87]. A lack of change impact analysis may lead to introduce several bugs and security vulnerabilities on the software system [3]. Thus, performing change-impact analysis on a system is vital for the success of the system as it consists of verifying the propagation of any change through the dependant entities [58].

Many researchers have studied CIA in mono-language systems *i.e.,* systems developed with a single programming language [39, 40, 45, 88]. Different causes of changes in the scope of mono-language systems have been enumerated in the literature, *e.g.,* requirement uncertainty, market demands, changing environment, new clients' requirements [54]. Nowadays, developers are often using multi-language development and multi-language systems are thus becoming more prevalent [19, 76, 89]. However, investigating CIA in those systems presents a new challenge due to the complexity of such systems in terms of dependency analysis, source-code entity interaction, change propagation, different programming language rules *e.g.,* lexical, semantic, and syntactical, different mechanisms to combine languages *e.g.,* Java native interface, Python C extension, etc. [8].

Our objective in this chapter is to understand how developers manage changes in multi-language systems. First, we examine the developers' motivations for change-impact analysis. Next, we investigate how developers deal with changes in multi-language systems, in particular, we investigate tools, techniques, and methodologies followed by developers in industry. We also inquire about the difficulties that they face when conducting their analysis and the impact on system quality and system security. Lastly, we present recommendations to guide and help industries and developers in their change-impact analysis in multi-language systems. Thus, we answer the following two research questions :

RQ6.1. What are the challenges of change-impact analysis in multi-language systems? On the one hand, we investigate the main reasons behind source-code changes. On the other hand, we investigate the challenges and issues that developers face when changing their multi-language systems. So, through this general research question, we aim as well to answer the following sub-question :

- What are the consequences of a lack of change-impact analysis in multi-language systems ?

RQ6.2. What are the existing means for change-impact analysis in multi-language systems? We investigate how developers analyze and make changes to their multi-language systems. We investigate the existing means that developers use to perform their change-impact analysis on multi-language systems. Thus, we answer the following sub-questions :

- Do companies put into place specific means to conduct a change impact analysis in multi-language systems ?
- What are the different steps followed by developers when conducting their changes in multi-language systems ?
- What are the requirements for a professional change impact analysis means for multi-language systems ?

To achieve these objectives and answer the research questions, we developed a survey with 30 questions that we sent to 200 developers. The questionnaire was completed by 69 different developers (34,5%), from eight countries, with mixed backgrounds and mixed work positions. We present the questions in the appendix of this thesis.

A survey is the adequate method to answer these research questions because it allows collecting information from different stakeholders to describe and compare their knowledge and behavior [90]. There are many ways to conduct a survey : either directly by asking participants or indirectly by reviewing written or visual records. In this study, we opted for a direct survey. Thus, first, we shared the questionnaire using LinkedIn platform. Then, to validate the answers, we asked the participants, via the last question of the survey, for a follow-up of a 20-minutes interview so we can ask more precisely and interactive questions. Eight out of 69 agreed to participate.

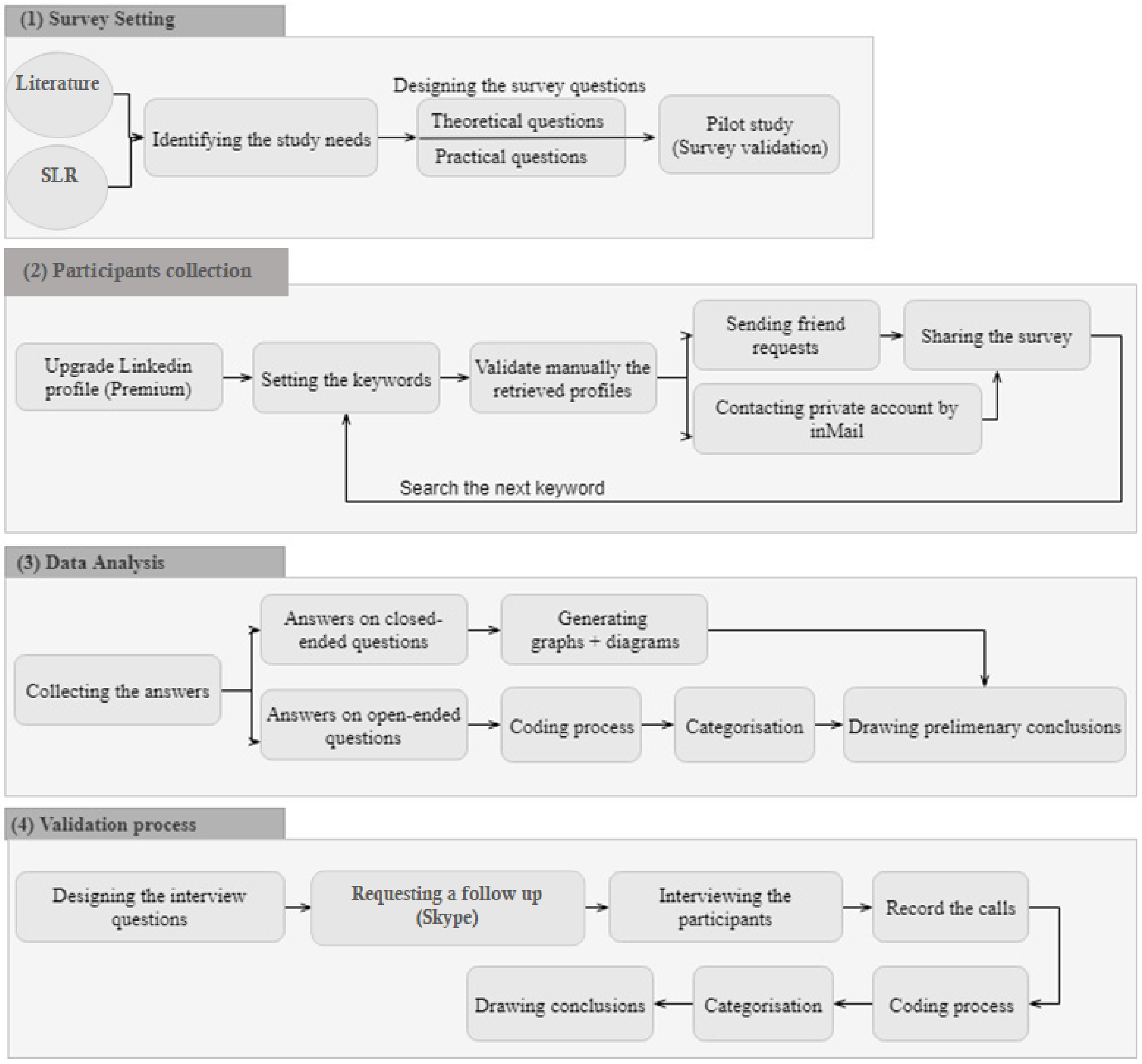

(1) Survey Setting
Literature
SLR
Identifying the study needs
Designing the survey questions
Theoretical questions
Practical questions
Pilot study (Survey validation)
(2) Participants collection
Upgrade Linkedin profile (Premium)
Setting the keywords
Validate manually the retrieved profiles
Sending friend requests
Sharing the survey
Contacting private account by inMail
Search the next keyword
(3) Data Analysis
Collecting the answers
Answers on closed-ended questions
Generating graphs + diagrams
Answers on open-ended questions
Coding process
Categorisation
Drawing prelimenary conclusions
(4) Validation process
Designing the interview questions
Requesting a follow up (Skype)
Interviewing the participants
Record the calls
Coding process
Categorisation
Drawing conclusions

Figure 6.1 An overview of the followed methodology

6.2 Study design

In this section, we present the design of this study. We show in Figure 6.1 an overview of the different steps followed.

6.2.1 Questionnaire Design

The survey questions[1] were prepared based on prior literature [36, 45, 54] as well as software blogs, websites, and developer discussions. We used the online professional survey platform `CheckMarket`[2] that possesses a free trial version and offers an unlimited number of questions. Our survey contains close-ended questions with potential predefined answers and open-ended questions, to give the participants the freedom to provide all the information that they consider relevant. We present the questions used in this survey in Appendix Section.

6.2.2 Questionnaire implementation

As a basis for any questionnaire, we started by the policy of the study where we confirmed that participants' identities were kept anonymous both for the data and the results. Then, we detailed the objective of the study in which we presented a short definition of multi-language development and change-impact analysis concepts, to help participants understand the context of the study and the research questions.

We split the questionnaire into three parts, where the first part is dedicated to participant's information and work experience in general. The second part is dedicated to their experience with change-impact analysis in multi-language systems, and the third part presents a short case study where participants are asked to answer three analysis questions related to a given source code scenario.

The first part included nine questions related to information about the participants' context of work *e.g.,* education level, work positions, years of experience, company information, competencies, programming language knowledge, etc. All the questions in the survey contain the references from where we extracted the information.

The second part included 18 questions mixed between multi-language development and change-impact analysis knowledge. The questions included experience in these two topics, change impact analysis means being used, challenges faced, etc.

The third part included one multi-language source code scenario. We provided a small pro-

1. `https://s-ca.chkmkt.com/?e=132567&h=EC01C1B95F9F3F8`
2. `https://www.checkmarket.com/`

gram written in (Perl, Java, and C) with a descriptive text where the developer needs to apply changes in the Java part of this code source (example of code extracted from [3]). we asked them what would be the impact in the C part and what are the methodologies that they would use. In total, we asked the participants three open-ended questions from which we aimed to extract information about the behavior of a developer in such situation. The 30 questions in the survey were conditional. Regarding Yes/No questions, if we suppose that the answer is No then our survey will ignore the following related questions and bring the respondent to the next, relevant one.

The survey is a mix of close-ended and open-ended questions. 24 questions are close-ended with predefined answers where participants select their best answers. An open choice box “Other (please specify)” is available, if the participants’ answer does not match with any of the predefined answers. Six questions are totally open-ended and contain empty fields to give to the participants the ability and freedom to describe and explain their ideas. The open-ended questions were created especially where we need participants’ opinion about the benefits and improvements of specific change-impact analysis mechanisms.

6.2.3 Interview Sessions

We performed interview sessions with eight participants who kindly accepted our invitation. We believe that an interview *i.e.,* one-on-one conversation between the participants and us is the better method to validate and evaluate the survey answers in an interactive way. Performing these interviews allowed to assess the reliability of the answers in the survey by looking for contradictions, spurious, and unreasonable answers.

During the survey process, we asked the participants to provide us their email addresses if they are interested in a follow-up interview. Thus, we contacted them by email to fix upcoming appointments. We chose to carry the interview session using video-conference (Skype) as some of them were outside Canada. We performed the interview on three consecutive days according to the availability of our interviewees.

We performed open-ended qualitative interviews. The questions were flexible. We adjusted them depending on the interviewees’ answers, in order (1) to clarify the responses, (2) to follow promising new lines of inquiry, and (3) to probe for more details. Thus, during the interview sessions, we asked the interviewees :

- First, to validate their answers. This step allowed them to rectify some of their answers or to ask for more details about the questions that maybe were confusing for them.
- Then, to elaborate their answers. This step helped us to understand the interviewees’ intentions, since they had the opportunity to discuss in detail the difficulties due to

a lack of means, their expertise, etc., and the problems faced with changes in multi-language systems.

— Lastly, to resolve contradictions among the survey answers. With the interviewees' feedback, we were able to clean the data *i.e.,* the survey answers, to correctly categorize and code the answers, and to easily draw conclusions.

6.2.4 Participant Selection

`LinkedIn`[3], as a professional social media site, seemed like a good way to identify and make contact with potential participants for this study. It is a social network designed specifically for career and business professionals. We decided to use LinkedIn because it offers an advanced search with multiple filters to facilitate the discovery of relevant profiles.

We identified 200 developers by searching for them with specific criteria and keywords that we detail below. We also used the inMail contact available in the premium version of LinkedIn, to contact developers that can only be reached with a premium account. First, we defined the list of keywords that will be used to select the practitioners in LinkedIn. Then, we upgraded our LinkedIn profile to a premium version to take advantage of its advanced options available such as contacting people using inMail. Next, in the LinkedIn search bar, we started inserting our keywords one by one. And last, for each profile list returned by each keyword, we dedicated half of a day to check the online profiles and look for target persons, and then moved to the next keyword and new profile list.

Our search keywords for the developers' competences were "Software developer/Software development", "Software engineer", "Software analyst/Software analysis", "Code analysis", "Static analysis/Dynamic analysis", "Software tester/ Software testing", "Software quality", "Foreign function interface/FFI". Then, we manually checked the profiles retrieved, and contacted those who satisfied the criteria detailed above. "Multi-language" and "Change-impact analysis (CIA)" keywords as well as their synonyms are not used as competences in LinkedIn. Thus, it was impossible for us to know in advance and without checking the profiles if developers have competences in multi-language systems or change-impact analysis. We were interested in the profile of developer in general who had used several programming languages during their career.

The selection of the right profiles was challenging. Manual verification of the target participants was made by consulting the online profiles. We selected developers based on their online resumes as well as their actual or previous jobs. Some LinkedIn users add details to

3. `https://www.linkedin.com/uas/login`

their online profiles about old or in-progress projects. This information was very helpful to target the right participants.

After identifying potential participants, we started sending formal invitations via LinkedIn's message service and sharing the survey with a descriptive message of the study and a public link towards the questionnaire.

To get more participants, we also shared the survey by posting it in discussion groups specialized in software engineering and programming languages. We asked people to contact us if they needed more information or details about the study. We can not estimate the exact number of answers received from developers of these groups, because there is no way in linkedIn to know who and how much people in these groups had read our message or open our survey link. The only number that we can confirm is the 200 developers that we contacted directly and personally via LinkedIn messages. From the participants contacted, we received responses from 69 participants (34.5%).

6.3 Data Analysis

6.3.1 Answer Collection

CheckMarket offers the option to download the results of the survey in Excel, CSV, or PDF formats. In our case, we chose to save locally the data in a CSV file and to build our analysis and hypothesis based on the data interpretations.

Regarding the interview results, we recorded videos of the different sessions with the eight interviewees after getting their official approval. We committed to keeping the interviewees' identities anonymous and will never publish the interviews in order not to risk exposing personal information.

6.3.2 Coding Method

We extracted the responses to our open-ended questions *i.e.,* the survey's answers and the transcripts of the individual interviews, then, we encoded them. Close-ended questions did not need to be encoded because their answers are already predefined.

We followed the process of Bluff [18] to transform the qualitative data for analysis. The process implies three main steps :

— *Open Coding* to name and give meaning to the data, then, to link the similar meanings together, and renamed them as categories.

— *Axial coding* to build connections between categories and sub-categories. These connections are created by determining causes, contexts, consequences, covariances, and conditions.

— *Selective coding* to link all categories and sub-categories to the core category in order to make the “story-line".

In more details, the coding process in this study lasted six days, three for each of the survey's answers and the transcripts of the individual interviews. The detailed process performed as following :

— We regrouped the survey answers together for each open-ended question, and we did same for each individual interview transcript.

— In the first round, for each question separately, we extracted the keywords and the targets from each answer. Targets are keywords that provide enough information about the question. Examples of targets are presented in bold in Table 6.1 and described below. Some questions that asked for a description of a sequence of events/steps were coded differently. For example, Q28 required that we go through the answers and analyze each one in depth, there were no target words to look for. Instead, we looked for a logical sequence of events or steps.

— Then, for the second round, we categorized the targets according to the purpose and the goal of each question. We regrouped similar targets into one. One answer can be related to more than one target and one target can include more than one answer.

— In the third round, we went through the answers and their corresponding categorization, to verify and validate the matching.

Table 6.1 presents examples of three questions chosen randomly, to illustrate our encoding process. We present their answers and the three rounds of the coding that were performed. As can be seen from Question 15 “Have you faced, in your department/team, problems caused by a lack of change-impact analysis in multi-language systems ?” and question 16 “What were these problems ?” in Table 6.1, we extracted the main keywords from the answers in the first round, then we categorized them according to the reported consequences of a lack of change-impact analysis. The third round is a matching validation step. We discussed all ambiguities in the collected answers to generate the final categories.

Regarding question 27 “Imagine that a change-impact analysis software is being developed, what aspects should this software consider for a good change analysis in a multi-language system ?” in Table 6.1, the first round was similar to the first round of Q16, but the categorization was different because the goal and the context of the question are not the same. We regrouped the targets in three categories in terms of the realisability of such tool characteristics : easily realisable, medium realisable, and hardly realisable.

The last example presented in Table 6.1 concerns question 28 “What steps will you follow when applying this change ?”. The first round followed was the same as in the methodology of the previous questions. The categorization was made based on the way that a developer performed the changes, *i.e.,* considering whether a change-impact analysis was made *implicitly* or *explicitly*. Implicit change-impact analysis occurs when developers followed the steps required to do a CIA using a manual approach, or without knowing that they were doing a change-impact analysis, or they missed the step of analyzing dependencies, etc. Explicit change-impact analysis is when developers applied necessary means for a CIA such as checking the dependency between components, identifying the potentially impacted artifacts, applying specific testing for a multi-language system, etc.

6.4 Results

The following section presents the survey and the interview results, summarizes the findings, and provides recommendations. Each subsection shown below corresponds to one section of the survey and the last one concerns the interview sessions.

6.4.1 Demographics

We surveyed a total of 69 participants from eight countries as shown in Figure 6.2 : 44 were from Canada (63.77%), nine from France (13.04%), eight from Tunisia (11.59%), three from the USA (4.35%), two from Morocco (2.90%), and one from Jordan, Ukraine, and United Kingdom, each (1.45% each).

From question Q1 to Q8, we report the following results.

1. “What is your highest level of education ?”
   Figure 6.3 presents the findings. 39 participants among 69 had a Master’s degree, 20 (28.98%) had an engineering degree, seven (10.14%) had a Ph.D. degree, and three (4.34%) had a bachelor degree.
2. “What is (are) your current position(s) ?”
   Respondents hold different work positions in different enterprises, one developer can hold more than one position. We show the results in Figure 6.4. 46 (66.66%) participants are software engineers, nine (13.04%) postdoc position, eight (11.59%) web developers, seven (10.14%) system analysts, six (8.69%) project managers, six (8.69%) software testers. There were five (7.24%) technical consultants, three (4.34%) business analysts, two (2.89%) technical support, and only one developer (1.44%) was a network engineer.

Table 6.1 Illustration of our Data Encoding process

	Answers	**First Round**	**Second Round**
Q16	"**Incompatibility** between code components"	Incompatibility between components	Incompatibility
	"Completely **restart** over a feature because the client had a **new** awesome **idea**"	Clients new requirements	Rework
	"**Platforms issues** like Windows vs Linux or Unix"	Incompatibility between the OS	Incompatibility
	"We had to **work** on feature that were **supposed to be done** since something changed..."	New change appeared	Rework
Q27	"It should be **integrated** in an **IDE**"	Be part of an IDE	Easily realisable
	"Ensure the **behavior** of the program remains **unchanged** after modifications"	Ensure coherent results	Medium realisable
	"The **evaluation** of the many **risks** associated with the change and dependencies analysis"	Evaluate the risks	Hardly realisable
Q28	"Add jint parameter in C program. Change Java program to add parameter in functions. Add parameter in Perl program when we call Java program"	Changes were made step by step according to dependent classes identified visually without any CIA means	Implicit CIA
	"I will start making those changes from top to bottom (Perl to C). I will add a new question so that we can get the days in Perl. Then add the parameters in both print methods in Java. And finally i will implement the logic in C"	Changes were made step by step according to dependent classes identified visually without any CIA means	Implicit CIA
	"Look at the Java method **dependencies** (both what it calls and what calls it), synch up, and down the **changes requirement**. Compile and **test behavior**"	Some change-impact analysis techniques were used (Dependencies analysis, behavior testing)	Explicit CIA

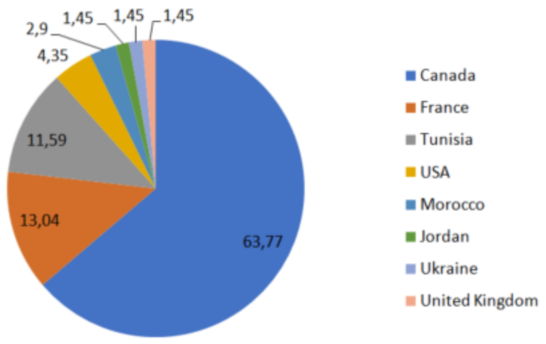


Figure 6.2 Participants countries

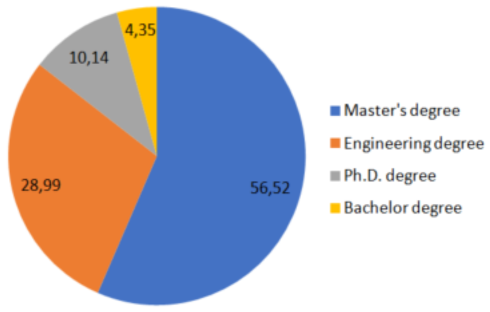


Figure 6.3 Education level

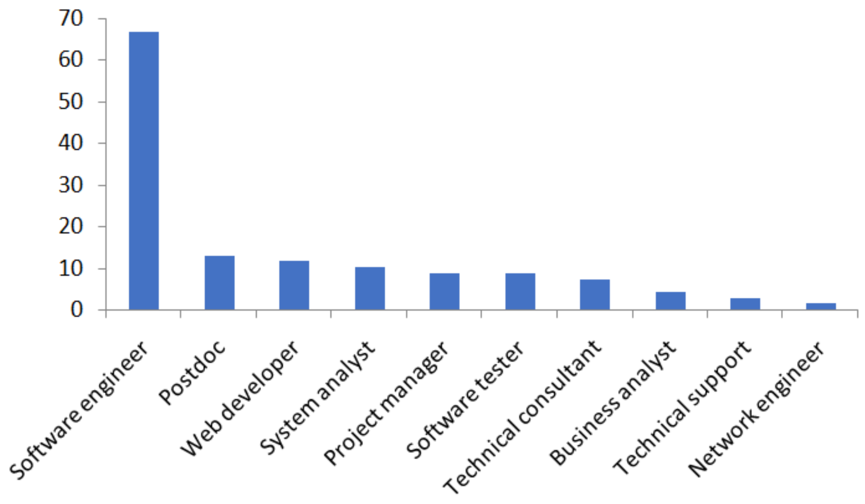


Figure 6.4 Work positions

3. "How many years of work experience do you have in software development ?"
   The majority of participants had a solid work experience as shown in Figure 6.5. 29 participants had between two and five years (42.03%) of experience, 33 participants had between six and ten years (47.82%), four participants (5.80%) had between 11 and 15 years of experiences, one participant (1.45%) had between 16 to 20 years of experience, and two (2.90%) participants were very highly experienced with more than 20 years.

4. "What is the field of the developed software in your company ?"
   Figure 6.6 shows that companies' domains were mixed as well : 22 (31.88%) of participants are working in the business and IT services. Then, 20 (28.98%) in research and development department where they are implicated in research activities undertaken by the company in developing new services/products, or improving the existing

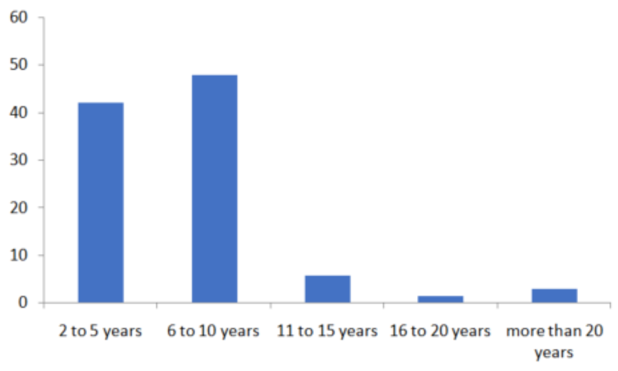


Figure 6.5 Work experience years

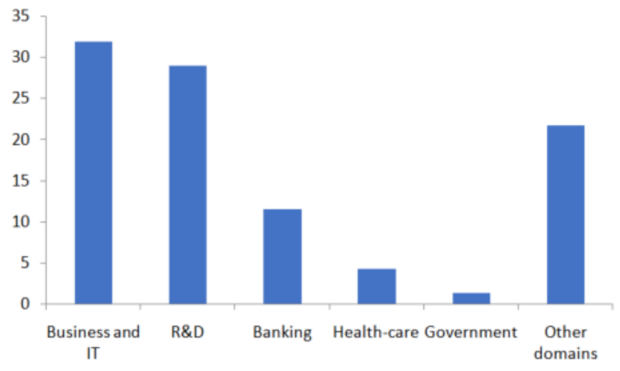


Figure 6.6 Companies field

ones. Eight (11.59%) in banking and insurance, three (4.35%) in health-care, and one (1.45%) in government and defense. 15 (21.74%) of the participants mentioned other domains like transport, finance, social media, telecommunications, and cyber-security.

5. "What is the size of your company (number of developers) ?"
   The size of these companies in terms of developers numbers is different from one company to another. 32 (46.37%) developers reported that their company holds less than 50 developers, eight (11.59%) reported between 51 and 250 developers, seven (10.14%) reported between 251 and 500 developers, and 22 (31.88%) affirmed that their company contains more than 500 developers. The results are presented in Figure 6.7.

6. "What is the number of software projects undertaken by your department/team per year ?"
   We asked the developers as well about the number of software being developed per year in their team/department inside these companies (Figure 6.8) : 45 (65.21%) affirmed that their team is developing less than ten software projects, 13 (18.84%) reported between 11 and 20, eight (11.59%) reported between 21 and 50, and finally three (4.34%) affirmed that more than 50 software projects are being developed yearly in their team.

7. "What kind(s) of software development methodologies does your company follow ?"
   The participants' companies follow different development methodologies as presented in Figure 6.9, one company can follow more than one development methodology. 65 (94.20%) affirmed using Agile while 38 (55.07%) affirmed using Scrum, nine (13.04%) reported using Waterfall, one (1.44%) affirmed using Spiral and XP.

8. "What programming language(s) do you use in your company ?"
   Programming language competences were also mixed. We asked a close-ended question

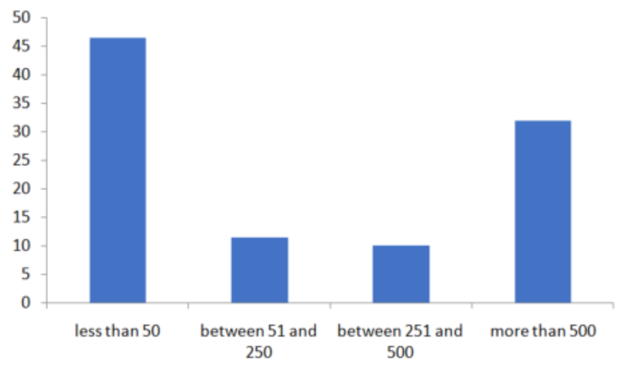


Figure 6.7 Companies size

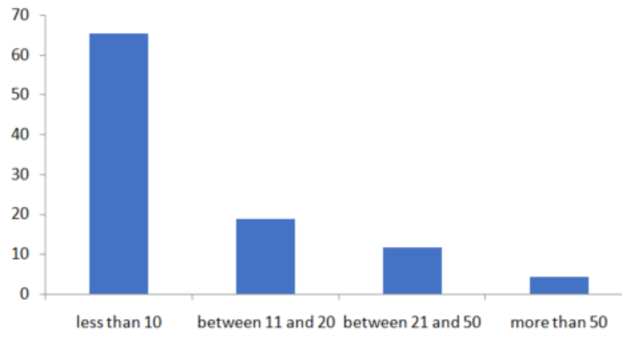


Figure 6.8 Projects developed per year

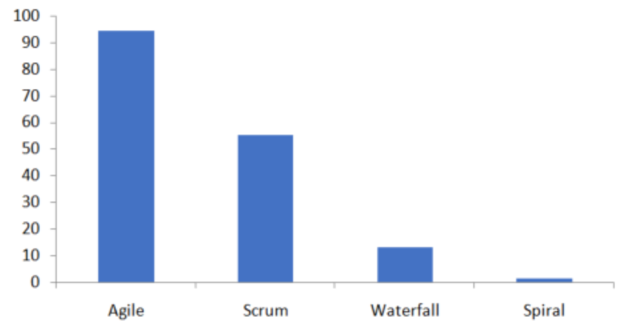


Figure 6.9 Software development methodologies

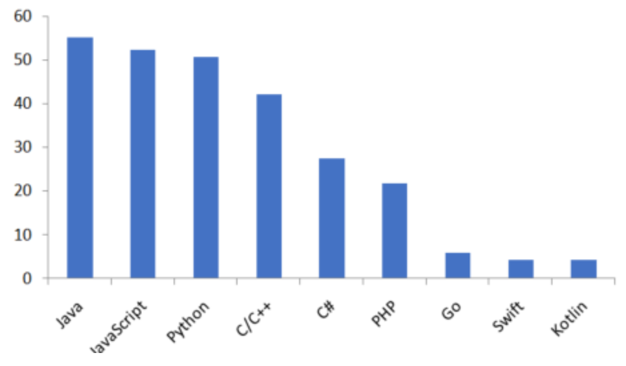


Figure 6.10 Developer's top ten programming language competences

with predefined answers extracted from the list of top 10 programming languages[4]. The participants were allowed to select more than one programming language competence. We used the same list of programming languages in the design of the last survey section, more detailed in the short program given to the participants. Results are summarized in Figure 6.10. 38 (55.07%) of the participants use Java, 36 (52.17%) reported using JavaScript. 35 (50.72%) use Python, 29 (42.02%) use C/C++, 19 (27.53%) use C#, and 15 (21.73%) use PHP. Four (5.79%) participants reported using Go, and only three (4.34%) reported using Kotlin as well as Swift. 18 (26.08%) participants mentioned that they are using other programming languages in addition to those in the top 10 list. They mentioned for example : Cobol, Erlang, F#, Objective C, Perl, Shell.

Among these surveyed participants, we performed an interview sessions with eight out of the 69 participants to assess the reliability of the answers in the online survey and to validate our

4. https ://simpleprogrammer.com/top-10-programming-languages-learn-2018-javascript-c-python/

Table 6.2 Interviewees' demographics

Interviewees	Education level	Current position	Work experience	Company field
I1	Ph.D. degree	Software engineer and researcher	5 years	Research and Development
I2	Master's degree	Software engineer	5 years	Business and IT
I3	Master's degree	Software engineer	5 years	Banking and insurance
I4	Master's degree	Software engineer	6 years	Business and IT
I5	Master's degree	Software engineer	6 years	Business and IT
I6	Ph.D. degree	Project manager	7 years	Research and Development
I7	Engineer's degree	Software engineer	7 years	Business and IT
I8	Engineer's degree	Project manager	9 years	Business and IT

recommendations. Table 6.2 presents their demographics. All interviewees have strong experience with multi-language systems as well as change-impact analysis. Their work experience varies between five and nine years. They have as well a professional software background where six of them hold software engineer positions, and two of them are project managers.

6.4.2 Multi-language in Companies

Multi-language development is more and more used today in the industry. Most of the participants in the survey are dealing with this kind of systems. The results of Q10 "Have you used multi-language development in your past projects ?" show that 65 (94,2%) participants used multi-language development during their development. The following answers concern only the 65 participants that answered Yes to Q10.

From Q11 "What is the number of software systems your company worked on per year, that involved developing multi-language software systems ?". we observed that participants were involved in many multi-language systems in their companies. From Figure 6.11, we report despite the prevalence of multi-language development that more than 40% of participants reported that their companies deal with less than 25 multi-language systems per year (the minimum is two projects per year that can reach up to 25). In future work, we plan to investigate the reasons behind this small number of software systems that may be related to the difficulties of developing a multi-language system, but it could also that such development requires more experience with more than more programming languages and a good manipulation of the technique used to integrate the desired languages. In this question, we chose 25 as a selection threshold for the number of multi-language software systems being developed, because it is nearly equivalent to two completed software systems per month.

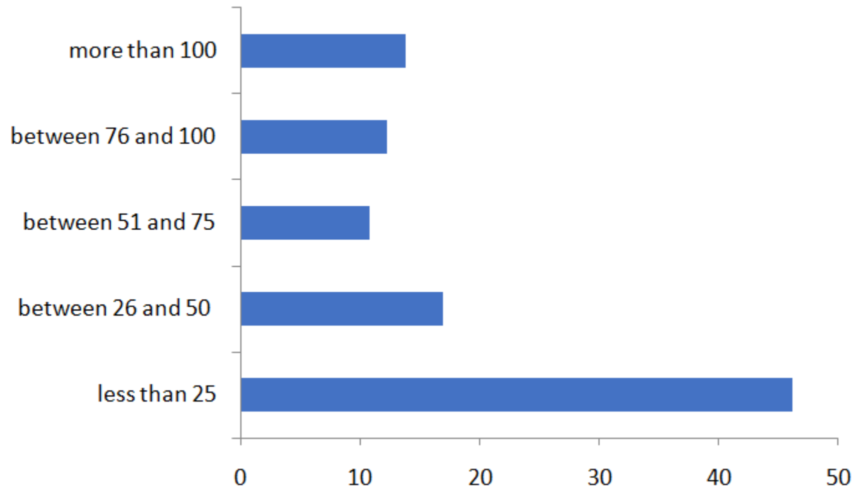


Figure 6.11 Number of Multi-language systems per year

Based on these quantitative numbers, we interviewed the eight developers directly about their opinions regarding the number of multi-language systems being developed in the industry nowadays. All of them noticed that the number is increasing quickly. They mentioned that nowadays, almost all systems developed within their current companies are multi-languages which was not the case in their first job. I2, I3, I5, and I6 said that, during their development tasks, they usually prefer to include external libraries written in different programming languages. They believe that this kind of mix allows them to not reinvent the wheel and to respect the deadline. However, I1, I4, I7, and I8 mentioned that if they had the choice, they would choose to work with a single programming language. Because, according to them, dealing with communication between different parts of the system is a challenging task. Moreover, multi-language development implies having excellent skills in many programming languages. I1 added that all the software systems in the company where he currently works involve more than one programming language. His tasks are in general related to the maintenance or the update of the existing multi-language software systems *e.g.,* a given platform written in Java where it is required to add new functionality for which Python (and Python libraries) seemed to be a more appropriate language.

Second, we inquired about the percentage of the increase in the multi-language systems that are developed within their companies. Only I8 was able to give us a rough rate because he is managing a development team for nine years. He said that this year (2019) there is an increase of nearly 60% compared with 2010 when he started holding his position. Moreover, our interviewees confirmed that indeed applying changes in multi-language systems is almost a daily task. The software systems within their companies are updated and maintained

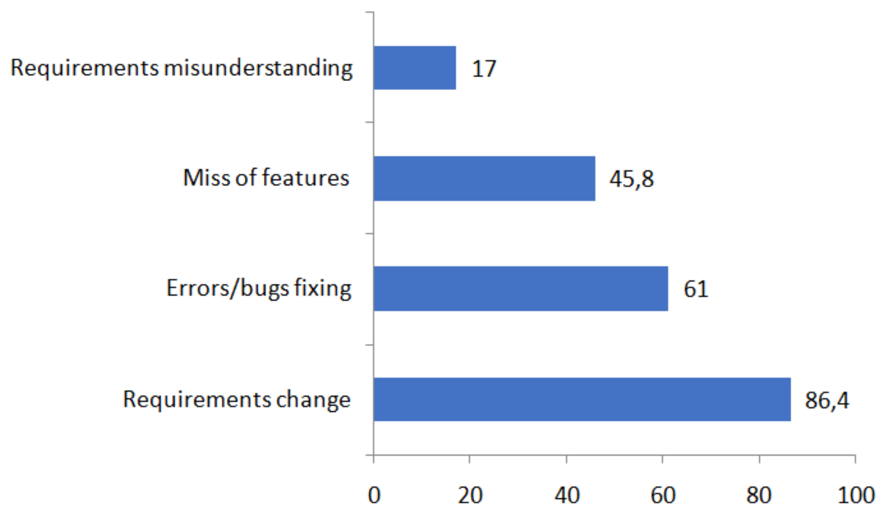


Figure 6.12 Percentage of Multi-language code change reasons

continuously before and after the delivery.

6.4.3 Challenges of change-impact analysis in multi-language systems

"Requirements change" (86.4%) presents the main recurrent reason for a multi-language change. The maintenance phase presents the most challenging activity during change impact analysis in multi-language systems.

We asked in Q13 "What are recurrent situations that you faced and that demanded changes be applied in multi-language systems ?". Multiple answers per participant are allowed in this question. Figure 6.12 shows that "requirements change" (86.4%) takes the first place. For example : a new functionality must be added/edited/deleted, new legislation or rules, new customers' needs. The second one is related to "faults/errors/bugs fixing" (61%) with changes required to improve the code, correct the bugs, and improve the quality. "Missing features" takes the third place (45.8%) ; this situation happened mainly when there is no communication between the teams working on the same software, when the testing steps are ignored or even the last validation of the requirements is missed. Finally, "misunderstanding of requirements" (17%) takes the fourth place. Misunderstanding of requirements can be represented in the following situations :

— A lack of understanding of the client's requirements.

— An underestimate of change requirements.

— The wrong choice of the methodology that does not satisfy the requirements.

We also asked in Q14 "How do you evaluate the challenges of change-impact analysis for the

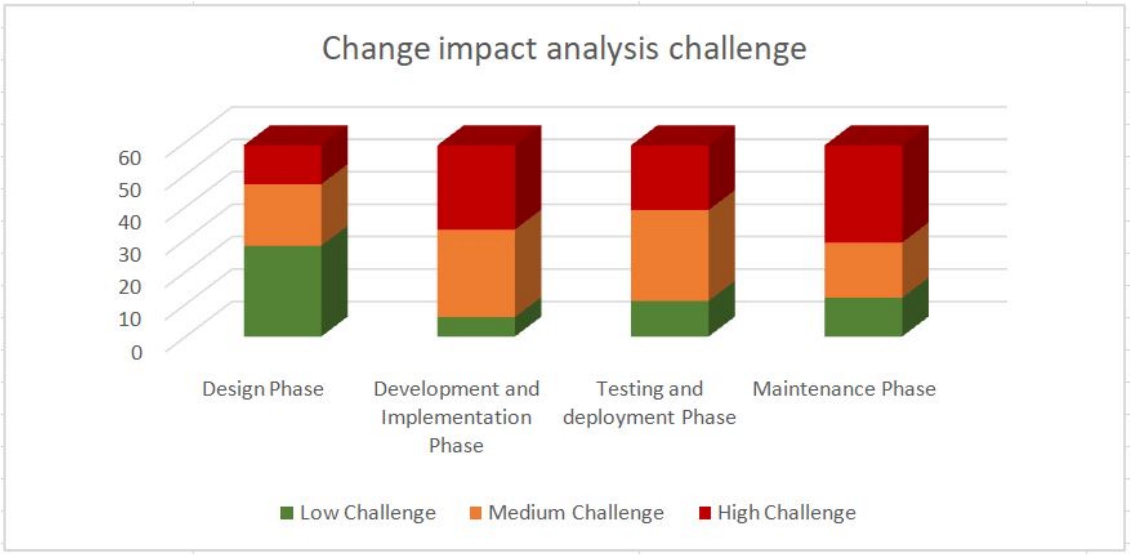


Figure 6.13 Multi-language change impact analysis challenge

following four phases ?" From the answers, we observed that change impact analysis makes the maintenance phase more challenging than the other phases. In the second position came the implementation phase. Then, the testing phase takes the third position while the design phase is less challenging according to the surveyed developers as illustrated in Figure 6.13.

Through the interview, we want to ask directly the interviewees more about the difference between change impact analysis in multi-language systems and change impact analysis in mono-language systems according to the interviewees' experiences. All the interviewees confirmed that CIA in multi-language systems is more challenging and risky. The results show six differences :

— Testing a change in multi-language system is hard. P2 said : "testing and debugging changes in multi-language systems is harder than mono-language because you need to consider how the data is called and used especially when the data is common between files written in diverse programming languages".
— Maintaining a multi-language system requires knowledge in diverse programming languages at the same time. P3 affirmed : "More languages in a system means having to know more about the whole architecture to know how a change could render another language interaction broken or vulnerable. Also, most tools are good to analyze basic vulnerabilities in a single language, but cross languages validation is much harder to predict".
— Managing the incompatibility of many languages in the same software is challenging. P5 said : "The compatibility of the parameters/variables that are exchanged

can become obsolete after changes that impacted many components written in many languages".

- Multi-language systems propagate more quickly the changes and impact more entities compared with mono-language. The impact of a tiny change in multi-language systems can be vast and propagate between other components written in different programming language's where it will be hard to detect the impacted artifacts *i.e.,* there is no dedicated tool for that. In mono-language systems, the impact is more limited because it touches only components written in the same language where it is more easier to detect the impacted parts even using traditional means.
- Multi-language systems are more vulnerable to cyber-security issues compared with mono-language systems. One of the interviewees affirmed that "interfaces between languages to communicate and transfer information introduces potential entry points and vulnerabilities".
- Multi-language systems are facing more quality problems compared with mono-languages *i.e.,* bugs. One of the interviewees said : "programmers are not often experts in all employed languages. The language that we have less knowledge/experience with increases the chances to create breaches or mistakes".

6.4.4 Methods used for change-impact analysis in Multi-language systems

Testing methods and manual checks are the main techniques used to analyze the impact of changes in multi-language systems. The use of dedicated tools or IDEs is not common. From Q17, Q18, Q22, Q23 presented below, we surveyed the specific means that developers used during their change-impact analysis in multi-language systems. The number of answers is higher than the number of participants because participants were allowed to put more than one answer. For example, when implementing a change on a system, a developer may use manual checks in addition to a specialized testing method.

- Q17 : What important method(s) do you consider before changing a line of code on a particular programming language embedded in a multi-language software system ?
- Q18 : What important method(s) do you consider after changing a line of code on a particular programming language embedded in a multi-language software system ?
- Q22 : Have you used any specific means to make your changes and to measure its impact in multi-language systems ?
- Q23 : If no, please specify your means being used.

Our results, which are summarised in Figure 6.14, show that 32 (49.23%) out of the 65 participants are using testing methods (*i.e.,* system and integration testing methods to test

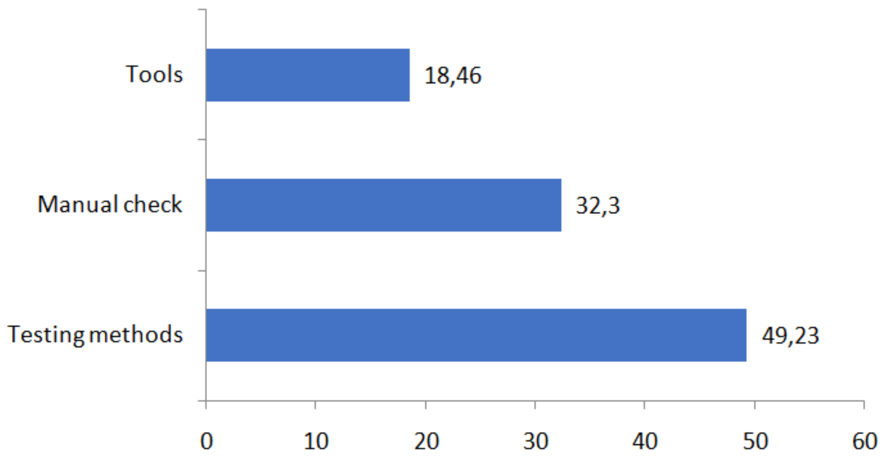


Figure 6.14 Percentage of Means followed by developers on their CIA

the integration of multi-language components), while 21 (32.30%) are using manual checks. Only 12 (18.46%) participants are using special methodologies and tools to analyze the impact of their changes that are developed within the company. Participants refrained to provide the names of their tools because of legal restrictions in the company. We observe that the use of specific tools to perform a change impact analysis is limited. Developers are using standard means as testing methods followed by manual verification. Thus, we were interested to investigate more about the change impact analysis tools, hence, we asked in Q25 “Does your company put into place specific means to conduct change-impact analysis in multi-language systems ?” and Q26 “Please describe them.”, to know if the developers’ companies provide some specific methodologies or tools to guide them and to ensure the success of CIA. Only 10 participants (15.38%) affirmed that their companies put into place specific CIA means that are developed within the company to conduct their change analysis. These means are mixed between a specific list of testing methods, code review guidelines that developers should follow, and some internal tools. We elaborate more on these answers in the interview section.

During the interview session, we confirmed the survey results by asking developers about their strategies to perform a change-impact analysis in multi-language systems. We identified from the answers three categories :

— Internal tools : I1, I4, I6, and I8 reported using four different internal tools that were developed within the industry (for the information confidentiality, we will name them X1, X2, X3, and X4). All these tools have in common the following options : dependency call graphs, testing methods, and detection of the coupled files. I1, I4, and I6 argued that their internal tools (X1, X2, and X3) have a limitation in terms of the number of programming languages that they support. Until today, these tools are updated and maintained according to the company new requirements. I8 mentioned that X4 supports only Java Native Interface systems (JNI) *i.e.,* the combination between Java and C/C++, and according to him the tool provides him all that he needs to make his changes in multi-language systems since he and his teammates work only with JNI systems. He also said that X4 is even able to measure specific metrics to JNI systems and ready to give percentages related to the quantity of the changes in a given JNI system.

— Manual analysis and testing methods : I2, I3, I5, and I7 confirmed using manual code analysis and code review in addition to testing methods implementation. They argued that manual code analysis is an alternative solution to reveal the different dependant system artifacts. However, all of them believe that these methods are not enough for the case of multi-language systems because they can not be exhaustive, and it is impossible to cover all the parts of a given multi-language system. Also, manual code review and manual testing are always likely to have many human errors compared with automatic testing or tools use.

— Documentation and knowledge share : I6 and I8 mentioned that they relied a lot on the documentation shared between teammates when a change should be applied in a given multi-language system. Given the complexity to analyze the dependency and the coupling between components written in different languages, they try to ensure good communication and to share information amongst teams in order to facilitate the migrations of tasks from one developer to another. They indeed have the internal tools that support multi-language systems but also believe that sharing knowledge is a very important step *i.e.,* each system developed within the company should have its documentation that should include information related to the software system hierarchy, languages used, dependant parts, as well as information related to the different versions, commits, bugs, new requirements, problems faced, solutions applied, etc.

Furthermore, we asked the interviewees about the frequency of the use of the CIA research tools in their companies. All of them (I1, I2, I3, I4, I5, I6, I7, I8, and I9) affirmed that they never used any academic tools in their change-impact analysis in multi-language systems or even in mono-language systems. They added that in general, they do not have enough time to search for an existing academic tool, to test it, then apply it in the change tasks. I1 and I6, who have an academic background *e.g.,* researchers, argued that there is a rupture between industry and research. I4 and I8 reported that in their companies, they do not have the choice to choose their preferred CIA means, so they need to follow the instructions of project managers to do the job.

6.4.5 Consequences of a lack of change impact analysis in multi-language systems

51 of the participants (78.46%) experienced issues in their company, because of a lack of change-impact analysis, and these issues negatively impacted the project progress, its quality, and its security. We surveyed the consequences related to a lack of change-impact analysis in multi-language systems. Results of Q15 “Have you faced, in your department/team, problems caused by a lack of change-impact analysis in multi-language systems ?” and Q16 “What were these problems ?”, shows that 51 participants (78.46%) experienced issues in their company, because of a lack of change-impact analysis, and these issues negatively impacted the project progress, its quality, and its security. We coded these answers and we categorized them into four groups. We summarize the result below :

— Decrease in quality : 20 participants reported that, in their previous projects, a lack of change-impact analysis in multi-language systems let to a decrease in code quality. From the received answers, we cite the following as example : *e.g.,* “Delay in the release of a product due to avoid evaluating the impact of a change request".

— Introduces of security vulnerabilities : Five participants reported that, according to their personal experience, multi-language systems are more susceptible to vulnerabilities. One of the participants believes that it will mainly touch the segmentation ; he said : “... segmentation faults principally". Another developer said : “... entry points such as code injections that become exploitable because of the need of the languages to exchange data". The others affirmed facing cyber security issues caused by a change applied in multi-language systems which mostly were : code injections, memory dump, or incorrectly released memory.

— Increase in software costs : Eight participants reported that, because of a lack of CIA in their multi-language systems, they had to restart some parts of a project from the beginning and this caused an increase of the software cost. An example of the received answers : “staying as general as possible to include all possible outcomes and taking into account the augmented Budget".
In some situations, a lack of analysis of the impact of changes can negatively impact the whole functionality of the multi-language system or affect some components and their coupling. It may force the developer to restart developing the component from the beginning with another methodology and with consideration of the change. Restarting some part of a project will always lead to an increase in the development budget as cited by one of the participants : “We had to rework a feature that were supposed to be done since something changed in the back end side and we weren’t expecting these

types of changes".

Furthermore, they reported being involved in maintenance activities that impacted the cost negatively. Migration to new technologies also led to cost increase, *e.g.,* “changes of used technologies...".

— Introduction of bugs : Nine developers explained how ignoring change-impact analysis in multi-language systems led to the introduction of issues in their system. They commented that a lack of tests, analysis, team communication, CIA experience, and knowledge in multiple programming language may result in bugs being introduced in the system. One of the participants said that : “Not enough regression testing during the maintenance phase...Not understanding the scope of a change".
— Introduction of incompatibilities : Nine participants discussed issues related to the introduction of incompatibilities in the system. They explained how they faced some incompatibility issues between modules, components, languages, and platforms because of a lack of CIA in multi-language systems. One of the participants reported a faced problem as : “Code desynchronisation and incompatibility between code components".

Our interactive interview led to discuss more in depth the issues faced by developers due to a lack/miss of change-impact analysis, and highlighted by them during the survey. We had a long discussion with I6, where he told us a story when he was a developer for almost six years before being promoted as a project manager. At that time, there was an application written mainly with Objective-C that contained a component written in JavaScript. During one year, several changes had been made to the application to update it and to add missing features to meet the need of their client. However, the whole team did not realize that there was a part in the application written in JS that must be considered during the changes ; otherwise, the new changes may break some features. Indeed, they delivered the application with the latest changes well tested and analyzed, but they never investigated the impact of this change by analyzing the coupling between the different components written in different languages other than Objective-C *e.g.,* the part written in JS. Thus, this error led the client to lose data and then, it led the team to rework on it one more month to fix it because the changes made were not compatible with the component written in JS. This lack/miss of the analysis of the impact of the change between parts written in different languages led the company to lose future projects with that client.

Table 6.3 Requirements needed for a multi-language CIA tool

Easily realisable	**Medium realisable**	**Hardly realisable**
1- Analyse dependencies	1- Detect bugs	1- Include business impact
2- Measure metrics	2- Propose tests	2- Include change cost
3- Integrate in an IDE	3- Detect vulnerabilities	3- Evaluate risks
4- Support different OS	4- Perform coherent changes	4- Propose potential changes automatically
5- Ensure traceability	5- Ensure usability	
6- Identify components to be changed	6- Ensure scalability	
7- Rollback a false change	7- Propose design conception	

6.4.6 Requirements for a multi-language change-impact analysis approach

The majority of the participants call for the development of dedicated CIA tools for multi-language systems where dependency analysis option consists of the major requirement. Despite the numbers of participants (*i.e.,* 51) who reported having faced problems caused by a lack of change impact analysis, 62 participants expressed a need for change-impact analysis tools that support multi-language development (even developers who did not face issues with CIA in multi-language systems, need a supportive tool to help them manage their impact analysis in an multi-language systems). From Q27 "Imagine that a change-impact analysis software is being developed, what aspects should this software consider for a good change analysis in a multi-language system ?", we extracted a list of requirements for a CIA tool for multi-language systems. We summarize these requirements in Table 6.3, organizing them in three groups based on their feasibility : "Easily realisable", "Medium realisable", and "Hardly realisable". We discussed our assessment of feasibility with other researchers from the community. We found that 53 out of 62 (85,48%) asked firstly for a support within the dependency analysis to help them in tracking the propagation of their changes in the hidden source code parts.

In addition to the survey results, we were interested in getting the opinions of the eight interviewees regarding the requirements for a CIA tool that support multi-language systems, collected from the online survey' answers. I1, I2, I3, I4, I5, and I7 thought that among all the proposed tool requirements : (1) *analyzing dependencies e.g.,* building call graph dependency between component written in different languages, extracting the dependency hierarchy, detecting the potential impacted dependent artifacts, and (2) *supporting different OS platforms* are the most critical requirements for them as developers. I6 and I8 mentioned that evaluating the risks, the change cost, and analyzing the business impact before performing any changes to a multi-language system, are excellent options, and for a project manager, it matters a lot. The interviewed people believe that considering these requirements when developing a new CIA tool for multi-language systems is necessary. These requirements reflect the real need of developers. Hence, we strongly encourage researchers and practitioners to consider them when designing change-impact analysis tools for multi-language systems.

6.4.7 Scenario of change-impact analysis in multi-language systems

35 (62.5%) participants used change-impact analysis implicitly. System testing is the main test (21.57%) used by developers to test a multi-language change.

The last section of the survey presents a short case study where the scenario of a change in a multi-language system is presented. We proposed a simple case study where the developer does not need to do a hard analysis or invest a lot of effort in order to have more chance that the participants answer all the questions related to this scenario and to not ignore them. This case study contains three programming languages (Java, C, and Perl) that interact together in the same program as shown in Figure 6.15. Our questions were limited to the JNI part (the interaction between Java and C functions). We also provided participants with a tutorial of JNI development in case. The program was accompanied by a descriptive text explaining the role of each method. The provided program uses two methods printTemperature() and printHumidity() to display temperature and humidity rate. Their respective signatures in C are the following :

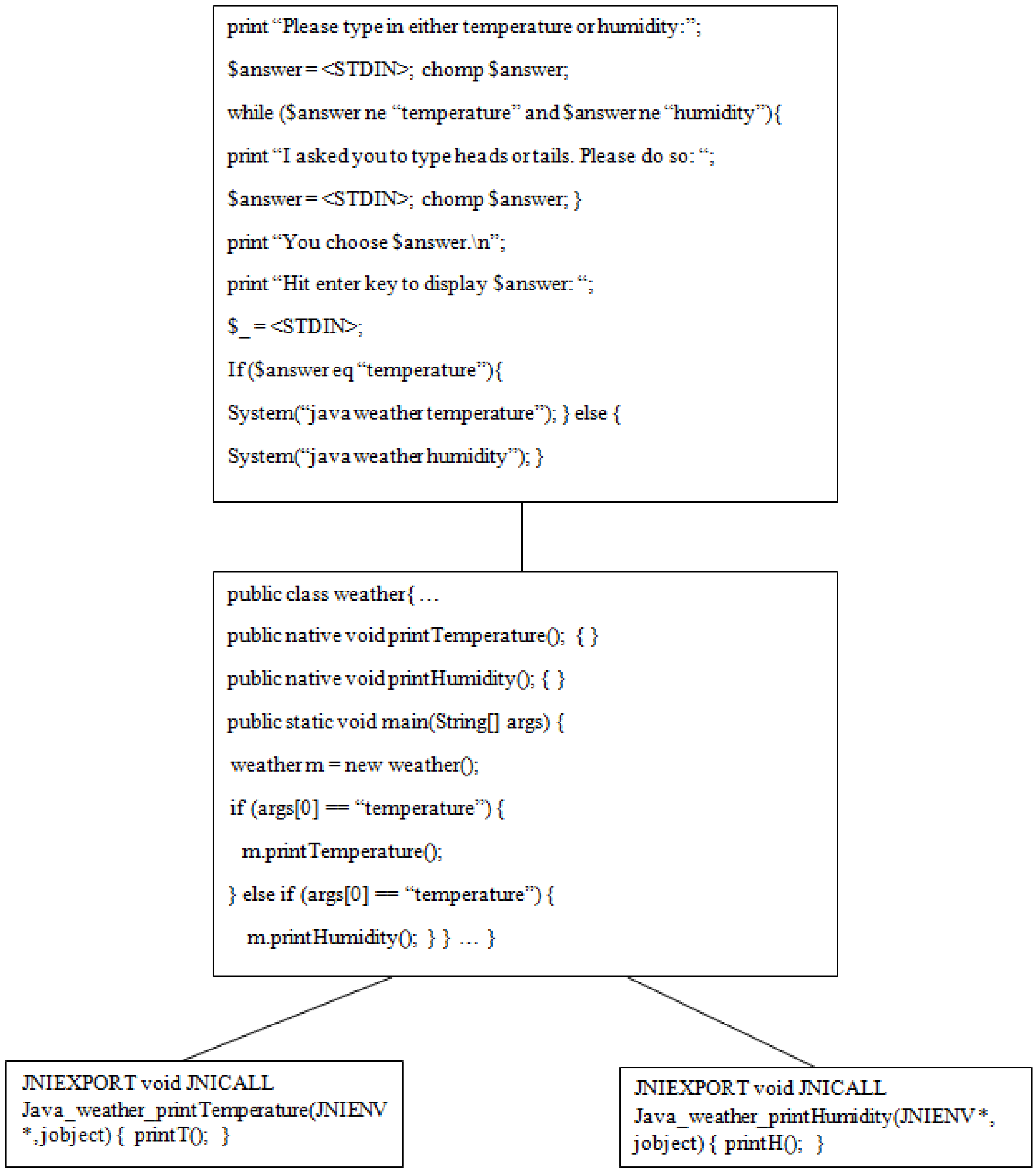
```
JNIEXPORT void JNICALL Java_weather_printTemperature (JNIENV *, jobject);

JNIEXPORT void JNICALL Java_weather_printHumidity (JNIENV *, jobject);
```

We asked developers to make a change in the Java method printTemperature() that had dependencies with C source code. The change involved the signature of the method by adding a new integer argument, to identify weekdays, and analysing the impact of this modification. So, the native method signature should become :

```
JNIEXPORT void JNICALL Java_weather_printTemperature (JNIENV *, jobject, jint);
```

We asked the following three open-ended questions to participants, corresponding to Q28, Q29, and Q30 in the survey :

— Q28 : What steps will you follow when applying this change ?
— Q29 : What are the different tests that you would implement and execute in this case ?



```
print "Please type in either temperature or humidity:";
$answer = <STDIN>; chomp $answer;
while ($answer ne "temperature" and $answer ne "humidity"){
print "I asked you to type heads or tails. Please do so: ";
$answer = <STDIN>; chomp $answer; }
print "You choose $answer.\n";
print "Hit enter key to display $answer: ";
$_ = <STDIN>;
If ($answer eq "temperature"){
System("java weather temperature"); } else {
System("java weather humidity"); }
```

```
public class weather{ …
public native void printTemperature();  { }
public native void printHumidity(); { }
public static void main(String[] args) {
 weather m = new weather();
 if (args[0] == "temperature") {
   m.printTemperature();
} else if (args[0] == "temperature") {
   m.printHumidity();  } } … }
```

```
JNIEXPORT void JNICALL
Java_weather_printTemperature(JNIENV
*,jobject) {  printT();  }
```

```
JNIEXPORT void JNICALL
Java_weather_printHumidity(JNIENV *,
jobject) {  printH();  }
```

Figure 6.15 Multi-language scenario [3]

- Q30 : How will you identify and analyze the impact of this change (*i.e.,* tools, techniques, or methodologies that would be used) ?

Regarding the first question, we received 56 answers that we categorized in three categories :

- "Implicit change-impact analysis" where CIA has been implicitly made. It means that developers followed the steps required to do a CIA using a manual approach or without knowing that they were doing a change-impact analysis.
- "Explicit change-impact analysis" where the mechanisms of the change analysis have been clearly used.
- "Do not provide information" where answers are too general and–or do not provide enough information to analyze.

An example of this categorization is given in Table 6.1 from Section 6.2. Results show that 35 (62.5%) participants used change-impact analysis implicitly, while nine (16.07%) used it explicitly. 12 (21.42%) did not provide detailed answers.

Regarding the second question, 49 answers were received. Table 6.4 presents a summary of the results. We ignored seven of the answers (13.73%) that did not provide specific test methods. In their answers, participants could provide more than one testing method. 3,92% of the answers were not enough detailed, developers declared using functional testing in general which can be any of the testing techniques presented above. We present in the following the top 3 testing methods identified by the participants :

- System testing was the main test (21.57%) that a developer would implement to test a change. This test validates the complete integrated software components. System testing is important in the case of multi-language systems as it ensures that multi-language components are correctly connected together and that the dependency is not broken.
- Unit testing takes the second position (19.61%). Unit testing is essential as it validates that each component (in our case multi-language components) of the software system performs as expected.
- Integration testing takes the third position (13.73%). Within the context of multi-language development, this test is very important as it allows to test the interaction between integrated multi-language components.

Regarding the third question, we categorized the means followed by the 49 participants who answered this question. 19 participants (38.77%) said that they would start by applying some test methods, 12 participants (24.48%) by analyzing the code (static analysis, dependencies analysis, etc.), and eight (16.32%) by using an IDE. Six (12.24%) participants stated that the change could be identified manually, while four (8.16%) of participants reported that they would use metrics and code review techniques to ensure the safety of the changes. In general,

Table 6.4 Different tests used to analyze the impact of changes

Test methods	**Percentage**
System test	21.57%
Unit test	19.61%
Integration test	13.73%
Manual test	7.84%
Regression test	5.88%
Validation test	5.88%
Functional test	3.92%
Dependency test	1.96%
Interface test	1.96%
Sanity test	1.96%
Alpha test	1.96%

we can observe that developers identified diverse needs ; (1) tools such as : Visual Paradigm, LTTNg ; (2) techniques like : dependency analysis, slicing, graph use ; and (3) methodologies as : Static analysis, Code review, Testing. These results support our previous findings of testing methods as the main techniques used by developers when conducting a change.

6.5 Discussion

We discuss the technical survey findings per research question.

RQ5.1. What are the challenges of change-impact analysis in multi-language systems? The first result shows that multi-language development is common in the industry despite its low coverage in the literature. We argue that many open-source software systems are multi-language systems, but in this survey, we focused in industrial systems and not open-source systems which might not all be representative for industry software. However, CIA in multi-language systems is challenging especially in the maintenance phase, which is greatly impacted by a change.

One of the unexpected results is the fact that many of the surveyed developers did not know the term of change-impact analysis. **The majority of these developers are making change-impact analysis implicitly in their multi-language development**, *i.e.,* without knowing that they are doing a real analysis of the impact ; meaning that they do not have a good understanding of the existing means of change-impact analysis especially in the context of multi-language systems. We believe that performing the change-impact analysis implicitly in multi-language systems is more critical than mono-language systems. This lack of understanding of the benefits of the change-impact analysis in general and in multi-language systems in particular makes their performed changes risky (*i.e.,* subject to faults, bugs, security vulnerabilities), more complicated, and more challenging.

Furthermore, **Many participants did not report using any specific (either external or internal)** tool to prevent their changes from negatively affecting the software quality, however **they rely on testing to assess the impact of their changes in multi-language systems**. From the answers to our open-ended questions, we observed that developers in a large majority, use System test, Integration test, or Interface test to ensure a coherent behavior of their systems, after a change. We conclude that developers are aware of the importance of the use of specific testing methods dedicated to CIA in multi-language systems. However, we believe also that companies have to be aware of the risks of a lack of change-impact analysis in multi-language systems. In particular its potential negative impact on the quality of the systems, as well as on the software's budget.

RQ5.2. What are the existing means for change-impact analysis in multi-language systems? One unexpected result is the high rate of participants who reported using manual checks and testing methods compared with the small percentage who use specific IDE or tools. This outcome may suggest that existing testing methods and manual validation are enough for the type of CIA that our survey participants conducted in their activities. However, the need to support multi-language development, to automate this manual checking, and to have a tool that saves time, is real in companies, given the large proportion of participants who requested it. We believe that the automation of these activities will help developers improve the reliability of their changes as well as the overall quality of their systems.

Also, as we see in Table 6.4, manual testing is ranked fourth ; before some necessary testing methods for multi-language systems such as "dependency testing", "interface testing", etc. We believe that using manual testing in the scope of multi-language systems that contain different source-code entities written in different programming languages, is not a good practice. Manual tests can ignore or forget dependency testing between these entities. Hence, it is not enough to track and test the change propagation between the different entities level in a software, it can generate as well errors because of the different languages available in the same software system, or it might not be adequate for some big software systems. We recommend developers to prioritize automated testing rather than manual testing, following recommendations from the literature [91–93].

Nowadays, CIA in mono-language systems has progressed substantially, many static code analysis tools have been put into place to cover these challenges such as `Rigi`, `Modisco`, and `Jtracker` tools. Moreover, analyzing files written in the same language is easier than files written in different languages because the developer has to deal with each programming language rules *e.g.,* lexical, semantic, and syntactical, which lead to a challenging analysis of the association links between them.

We also observed that **the few existing multi-language research tools/approaches are not commonly used in the industry**. These tools have not been read or cannot be read by industry, so they cannot re-implement or build commercial products based on them.

6.6 Threats to validity

Threats to construct validity : The survey collects and analyses the developers' open-ended answers to learn about their personal experiences and behavior in a given scenario. More specifically, it relies on our evaluation of developers' reported opinions, experiences, and solutions. We started by following a coding methodology to code the different developers' answers (in the online survey and during the interview sessions). We then categorized the target keywords. To ensure high confidence in the validity of this analysis, we did a coding process [18], thus, we performed it in three rounds and validated the results through discussions with other researchers. The first round was dedicated to read all the answers and try to extract the keywords, then the answers, one by one, were assigned to a target group. The last round was dedicated to the validation of the final result.

Threats to internal validity : We manually validated the criteria of the participants' selection, the survey and the interview questions, and possible answers in the case of close-ended questions. We relied on the literature to extract the information needed to build the survey content regarding the scope of this study. We also performed a pilot with some researchers and students from our research team. We improved the survey based on the results of this pilot study. We believe that the use of LinkedIn also is an adequate choice, as the majority of the practitioners have a LinkedIn account. LinkedIn, allows specifying keywords to select participants. We validate the keywords used to select the participants that satisfy the requirements of this study. We guaranteed to the survey participants as well as the interviewees their anonymity and we highlighted that all the answers would be kept only for research use.

Threats to external validity : Our results from the survey may not be representative of all software practitioners, since we only received responses from 69 participants. We accept this threat as participants' answers could vary depending on different criteria. However, to mitigate this threat, we have shared the survey via LinkedIn, which includes a huge variety of people from different domains and different countries. The respondents are from different backgrounds and skills. We have also diversified the keywords to contact different profiles and to broaden the scope of our participant selection. The case study used in this survey may also present a threat to validity to generalise the behavior of developers when analysing a change-impact due to the size of the experiment. Regarding dedicated change impact analysis tools, we believe that it may be possible that some surveyed developers are not aware of these tools.

Threats to conclusion validity : We discussed all websites, platforms, and tools used in our methodology, such as CheckMarket, LinkedIn, and Microsoft Excel. We are aware that other websites could be used to perform the survey and for this reason, we explained and supported our choices within the methodology to create the questions, collect the participants, and share the survey.

From all the survey, we focus only on asking and interviewing participants about the change impact analysis within multi-language development context and we did not investigate their difference with mono-language development. Some of the findings, could be also applied for both mono- and multi-language development, however, we tried in the interview to ask about examples, languages used, emphasize about dependencies through components written in different languages and not same language, etc.

Threats to reliability validity : To mitigate the reliability threat, we provide all the needed information to replicate this study (the questions are provided in the appendix). We used online and mostly free websites such as CheckMarket to create the survey and LinkedIn to reach participants. The list of the participants could be different from one researcher to another. Searching for people on LinkedIn depends on different criteria such as : friends in common, location country, number of contacts for each profile, etc. We also provided the different steps followed during the study either to collect data or to analyse the findings. We assessed as well the reliability of the survey answers by performing an interview with eight participants to validate their answers and to mitigate possible contradiction answers.

6.7 Chapter Summary

To investigate change impact analysis in multi-language systems, we performed a survey that contained 30 questions and that we sent to 200 developers in eight countries. We received 69 participants' answers (34.5%) and we interviewed eight of them. We selected participants from different backgrounds and with different experience years to diversify the population under study.

Our study shows that there is a high need for CIA tools dedicated to multi-language systems in industry ; most developers currently are using testing methods and manual checks to perform their change impact analysis. This lack of formalized CIA processes makes multi-language systems more susceptible to be negatively impacted by vulnerabilities through changes. We believe that our results may help researchers and companies to improve the quality of their multi-language development research in order to better cope with their inherent complexity.

CHAPTER 7 EMPIRICAL STUDY ON THE INTER-LANGUAGE DEPENDENCIES IN THE JAVA NATIVE INTERFACE (Sub-hypothesis 2)

7.1 Chapter Overview

As discussed in the previous chapter, it is necessary and important to perform change impact analysis during the software system maintenance while introducing changes to the software [94]. Change impact analysis relies mainly on the dependency analysis between the component being changed and the potentially impacted components. Our results from the survey and interviews discussed above show that dependency analysis is the most requested CIA feature by developers. They argue that identifying the dependant components (written in different programming languages) is a challenging task. Hence, in this section we perform an empirical study with the goal to help developers overcome the dependency analysis in multi-language development.

The dependencies within a system reveal the entities potentially impacted by a maintenance task, assist developers in their maintenance activities, and allow tracking the propagation of their changes [52, 95]. Since each programming language has its own rules (*i.e.,* lexical, semantic, and syntactical), dependency analysis becomes difficult. This is because generic dependency analyses are no longer able to follow direct and indirect function calls in order to determine dependencies, i.e., developers need to understand the specific calling convention between, for example, Java and C++ [8,38]. In contrast to the dependency analyses in mono-language systems that have been studied extensively [47, 56, 88] using a variety of techniques such as static code analysis and mining software repositories, dependency analysis for multi-language systems is not well established yet [8] and is still subject to further research [55].

Thus, we empirically study ten Java Native Interface (JNI) open-source multi-language systems, to identify the inter-language dependencies (dependencies between components written in different programming languages), analyze their prevalence in multi-language systems, and their impact on software quality and security. We also introduced two approaches based on historical dependency analysis (H-MLDA) and static dependency analysis (S-MLDA). We focus, in this study, on the Java Native Interface since we found in Chapter 4 that JNI is the technique used the most by developers. Consequently, we address the following research questions :

RQ7.1. How common are direct and indirect inter-language dependencies in multi-language systems?

The goal of this research question is to measure the prevalence of direct and indirect inter-language dependencies. Unlike direct dependencies, indirect dependencies are difficult to identify using static analysis approaches. The goal of this RQ is to augment static analysis with historical analysis to enable us measure (and compare) the prevalence of both direct and indirect inter-language dependencies. We aim to compare the prevalence of static direct dependencies to indirect dependencies (potentially hidden from the static analysis).

RQ7.2. Are inter-language dependencies more risky for multi-language software system in terms of quality?

Dealing with dependencies in multi-language systems is a challenging task and requires specific effort as different programming languages are involved. Through this research question, we aim to understand if bug introduction is frequent in these kinds of dependencies (a case study of JNI) and its consequence on the system quality.

RQ7.3. Are inter-language dependencies more risky for multi-language software system in terms of security?

A major concern of practitioners and researchers nowadays is security vulnerabilities especially since the emblematic "Heartbleed bug", which is a security flaw that exposed millions of passwords and personal information. As today almost the software systems are multi-language systems, security vulnerabilities in those systems became a priority for developers [78]. This research question aims to identify the impact of inter-language dependencies on the security of multi-language systems. For this, we present the relationship between the inter-/intra-language dependencies with vulnerabilities.

7.2 Study Design

A dependency is a relationship link between entities inside the same software software. In this study, we consider the following types of dependencies :

— Inter-language dependency (inter-LD) : a relationship between files, written in different programming languages, where the dependency identification relies on the (third party) technology used to integrate different programming languages (*e.g.,* Python - C extension). Inter-language dependencies could be direct or indirect.

Table 7.1 List of the studied multi-language software systems.

Systems	Commit start date	Commit end date	#SLOC	#Commits
Conscrypt	2008/12/18	2019/08/27	64,6K	3860
RethinkDB	2013/04/10	2019/08/26	1,15M	19898
JatoVM	2005/09/24	2014/05/13	339K	4135
Libgdx	2010/03/31	2019/08/23	830K	13580
Lwjgl	2002/08/09	2016/12/08	88,9K	3884
Openj9	2017/08/31	2019/08/27	1,47M	6219
React-native	2015/01/30	2019/03/29	283K	16298
Realm	2012/04/30	2019/07/21	107K	8172
Seven-Zip	2007/12/19	2015/10/07	300K	909
VLC	2010/11/10	2019/05/26	96,1K	11866

  — Direct inter-language dependencies (DILD) : an inter-language dependency ensuring a direct communication between two files according to the multi-language conventions (*e.g.,* a change in native Java class requires changing the native C/C++ function). We use S-MLDA to identify them.
  — Indirect inter-language dependencies (IILD) : an inter-language dependency that is hidden from the static code analysis (*e.g.,* a change in Java class propagated to the native C/C++ function, that in turn impacted foreign C/C++ and/or Java files). We use H-MLDA to identify these dependencies.

— Intra-language dependency (intra-LD) : a relationship between files written in the same programming language. There is no specific (third party) technology used to ensure the communication between them.

We used the OpenHub API [1] for querying all OpenHub's systems to get the list of the systems that have at least two programming languages, in particular Java and C/C++ (they could have others in addition). We chose OpenHub because it provides the list of the programming languages involved in each open source system. From the obtained results, we took the first 100 systems sorted by "Rating" (an option provided by Openhub). We further used the system status to exclude inactive or abandoned systems from the 100 selected software systems. We analysed different systems based on their size (number of lines of code). We picked four systems $\leq 100k$ LoC (*i.e.,* Conscrypt, Lwjgl, VLC, and Realm), four systems between 100k and 1M LoC (*i.e.,* Seven-Zip, React-native, Libgdx, and JatoVM), and two systems $\geq 1M$ LoC (*i.e.,* Openj9 and RethinkDB).

Figure 7.1 shows, for each of the ten selected systems, the programming languages used with

1. `https://www.openhub.net/`

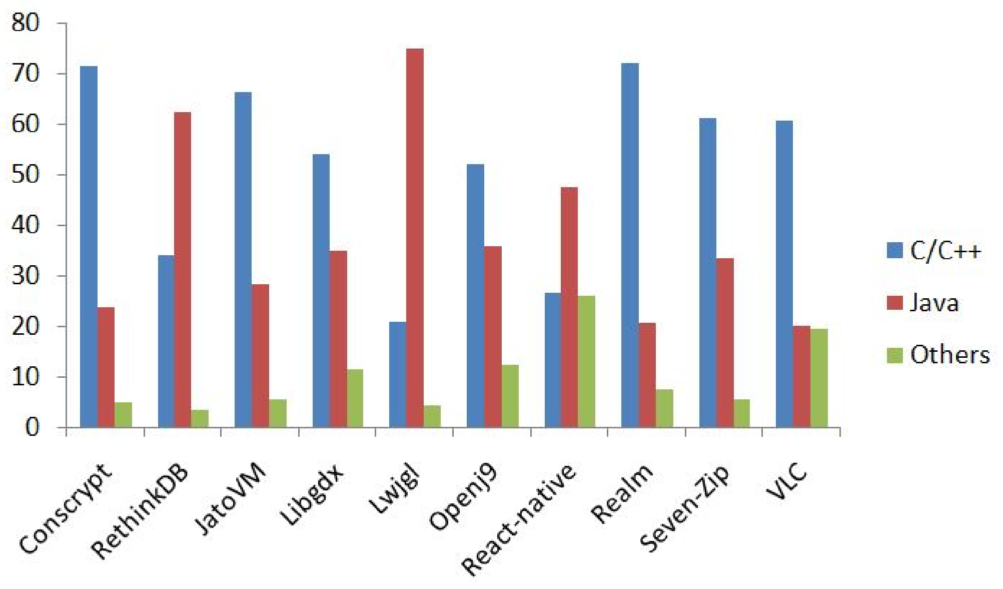


Figure 7.1 Percentage of programming languages used in each software system.

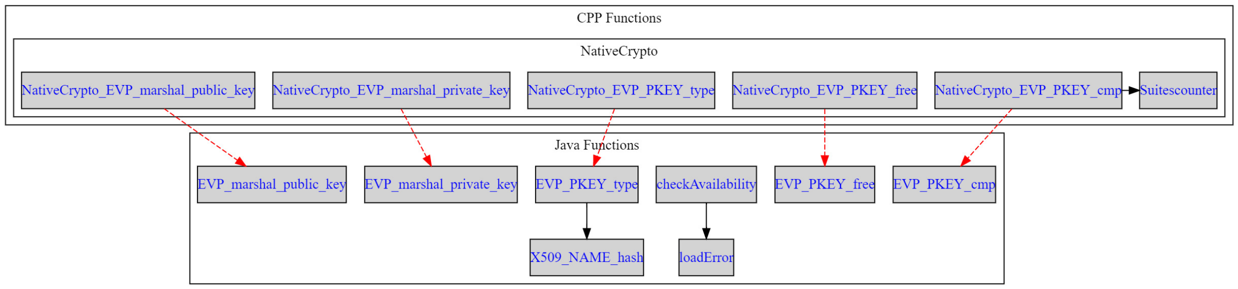


Figure 7.2 Dependency call graph of a part of Conscrypt generated by *S-MLDA*.

their respective proportions. We limit the presentation of our analysis in Figure 7.1 to three languages per system. The “Others” category combines the rest of programming languages. We present an overview of the collected systems in Table 7.1.

We use static and historic code analysis to identify the inter-language dependencies. We designed a first approach called S-MLDA based on JNI’s rules as defined by Liang [27] to identify the direct inter-language dependencies. We designed a second approach called H-MLDA to identify the indirect inter-language dependencies that are hidden for static code analysis using the changes history.

7.2.1 Static dependency analysis

Overview : S-MLDA is a static analyzer written in Java and based on PADL (Pattern and

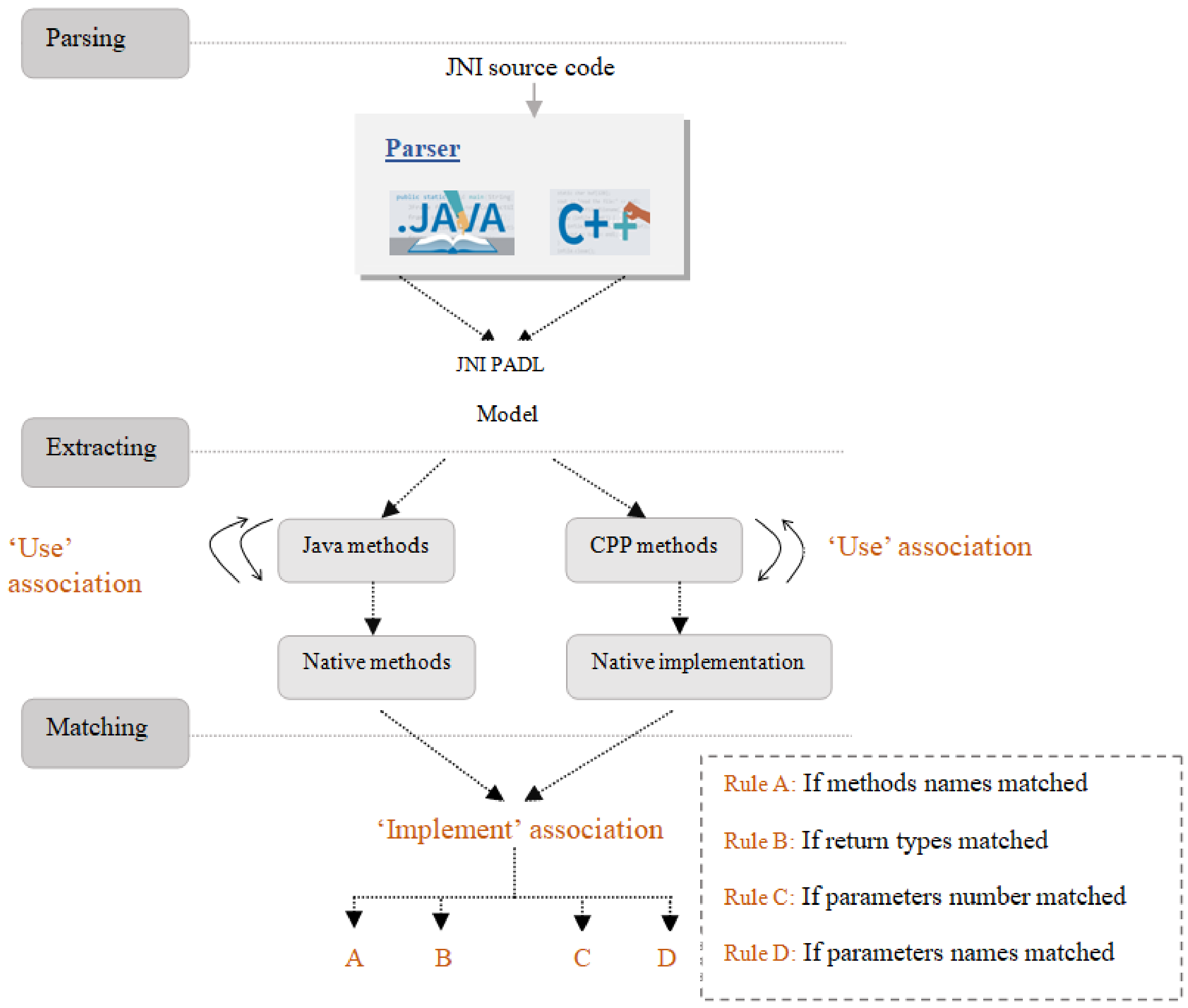


Figure 7.3 S-MLDA approach.

Abstract-level Description Language) [96]. It takes as input a set of multi-language files and statically analyzes their source code using an algorithm based on JNI rules. It provides as output a set of data, i.e., sets of files involving direct inter-language dependencies, presented in a dependency call graph showing the general relationships between files and the specific ones between methods. Figure 7.2 illustrates an example of S-MLDA output.

Motivation : To the best of our knowledge, there is no existing static analysis tool that analyzes the inter-language dependencies for JNI systems and generates a dependency call graph output [55]. Thus, S-MLDA is the first static analyzer able to report static dependencies between artefacts written in different languages.

Approach : S-MLDA consists of the following steps as shown in Figure 7.3 :

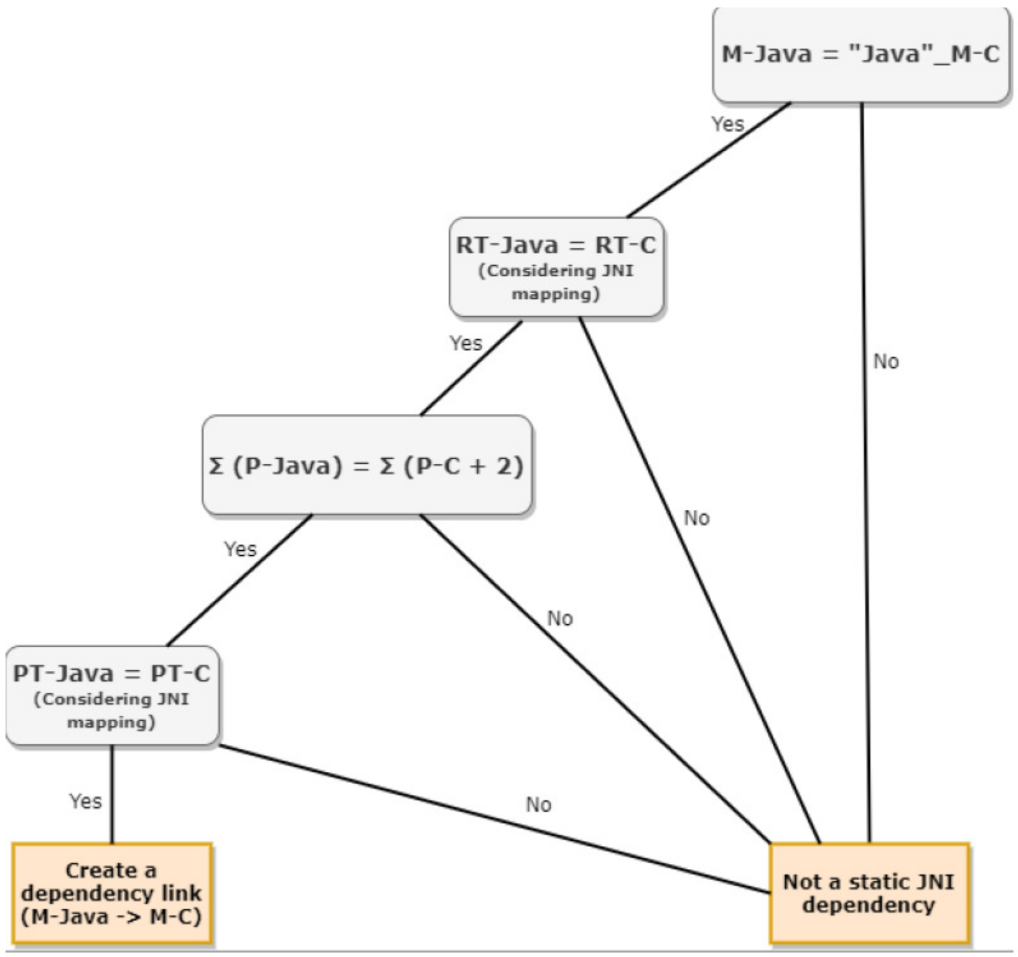

M-Java = "Java"_M-C
Yes
No
RT-Java = RT-C
(Considering JNI mapping)
Yes
No
Σ (P-Java) = Σ (P-C + 2)
Yes
No
PT-Java = PT-C
(Considering JNI mapping)
Yes
No
Create a dependency link (M-Java -> M-C)
Not a static JNI dependency

Figure 7.4 S-MLDA matching rules.
(M-Java : Java method name ; M-C : C function name ; RT-Java : Java method return type ; RT-C : C function return type ; P-Java : Java method parameters ; P-C : C function parameters ; PT-Java : Java method parameter types ; PT-C : C function parameter types)

1. Parsing : We parse a given JNI system to create a model that contains all constituents of that system *e.g.,* packages, classes, methods, parameters, fields, and relationships (inheritance, implementation, etc.).
2. Extracting : We identify the Java methods and C++ functions. From the obtained Java methods, we identified the JNI native methods *i.e.,* methods that contain the keyword "native" in their signatures, and from the obtained C++ functions, we identified the native implementation functions, *i.e.,* functions that implement the native methods.
3. Matching : We identify the matched Java native methods with the respective implementations in C++ files based on the JNI rules that we illustrate in Figure 7.4 :
   — Rule A : we verify if the Java method names (M-Java) match with the C++ function names (M-C) using the JNI naming convention.

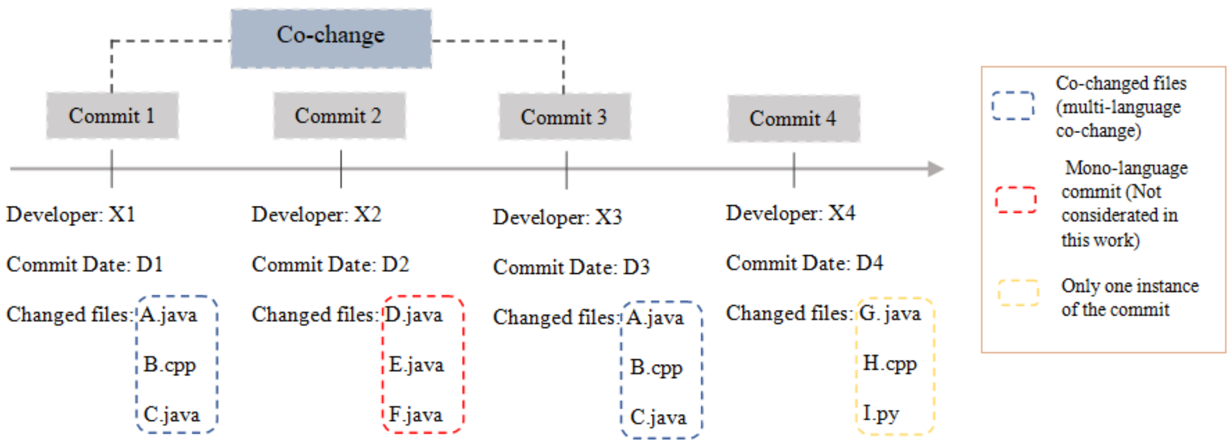

Co-change
Commit 1
Commit 2
Commit 3
Commit 4
Developer: X1
Commit Date: D1
Changed files: A.java B.cpp C.java
Developer: X2
Commit Date: D2
Changed files: D.java E.java F.java
Developer: X3
Commit Date: D3
Changed files: A.java B.cpp C.java
Developer: X4
Commit Date: D4
Changed files: G. java H.cpp I.py
Co-changed files (multi-language co-change)
Mono-language commit (Not considerated in this work)
Only one instance of the commit

Figure 7.5 Mono/multi-language co-changes analysis.

— Rule B : we verify the return types (using the JNI mapping types) from the obtained Java (RT-Java) and C++ methods (RT-C) from the previous step to keep only the matching types.

— Rule C : we verify the number of parameters. We consider the matching when the number of parameters of the C++ function (P-C) equals the number of parameters of the Java method (P-Java) plus two. In JNI, the C++ function implementing the native method contains two more JNI parameters *e.g.,* JNIEnv, jobject.

— Rule D : Last, we verify the parameter types of the Java method (PT-Java) and the C++ method (PT-C). We consider the methods/functions when we found that the mapping of the parameter type in both methods is matching.

We built, using the obtained relationships, a dependency call graph (example shown in Figure 7.2) with different hierarchy levels : classes level and file level. In each class, nodes are the methods and the edges are the dependencies. Edges between nodes across two sub-graphs are inter-language dependencies at methods level. We consider two files having direct inter-language dependencies if they involve at least one edge across the two sub-graphs. From the obtained sets, we remove the redundant dependent files.

7.2.2 Historical Dependency Analysis

Overview : H-MLDA studies the change history of a multi-language system's Git repository to reveal the indirect inter-language dependencies. It identifies the multi-language co-changes (involving multi-language files), converts them into sets of multi-language files, and removes the sets in common with the output of S-MLDA (example is shown in Figure 7.7). These steps allow retrieving the indirect inter-language dependencies not detected by S-MLDA (*i.e.,* potentially hidden for static analysis).

Motivation : Historical code analysis is one of the common methodologies followed to analyze dependencies [56, 97, 98]. In particular, the concept of co-change is a useful method to recommend dependent files potentially relevant to a future change request when they are detected by static code analysis. The purpose in this analysis is to identify a set of files changing together over time often enough (within commits) to derive an assumption that these files could be historically dependent. Several studies such as Abdeen *et al.* [37] and Hassan *et al.* [99] studied the dependency structure as an predictor and metric for co-changes in software systems.

Approach : H-MLDA consider a co-change to be a commit involving source code files that have been observed to change together [100] (set of files changing together) to exhibit some form of logical coupling, *i.e.,* a temporal relationship among files changed over time [101]. Let's consider Figure 7.5, which shows four different co-changes in a given multi-language system. Commit 1 and 3 are instances of the same changed three files *i.e.,* a co-change, involving the same three files. Commit 2 is a mono-language co-change *i.e.,* involving mono-language files. Commit 4 presents the case of a multi-language files with only one instance *i.e.,* a set of files changed together only once. To reduce the number of false positives, we did not consider the cases where files are co-changed accidentally without a significant reason *i.e.,* files appeared changing together (at commit level) only one time.

Figure 7.6 presents a summary of the main parts of H-MLDA. We query Git repository for extracting all the commits and analyze the data obtained. We split the co-changed set of files in two groups : one for the inter-language co-changes involving Java and C/C++ files and the second one for the intra-language co-changes involving Java or C/C++ files.

To allow a logical comparison at files level between S-MLDA and H-MLDA outputs (since, by default, H-MLDA concerns commits, while S-MLDA concerns files) and to identify the indirect inter-language dependencies, we convert the multi-language co-changes into sets of multi-language files as shown in Figure 7.7 in step (1). Then, we remove the ones in common with S-MLDA sets *i.e.,* the direct inter-language dependencies as presented in step (2) in Figure 7.7, and last, we consider the remaining sets the indirect inter-language dependencies. This is illustrated in step (3) in the same Figure 7.7.

7.2.3 Quality Issues and Security Vulnerabilities

We aim to understand if the inter-language dependencies in multi-language systems introduce more bugs and/or security vulnerabilities than intra-language dependencies. We also study

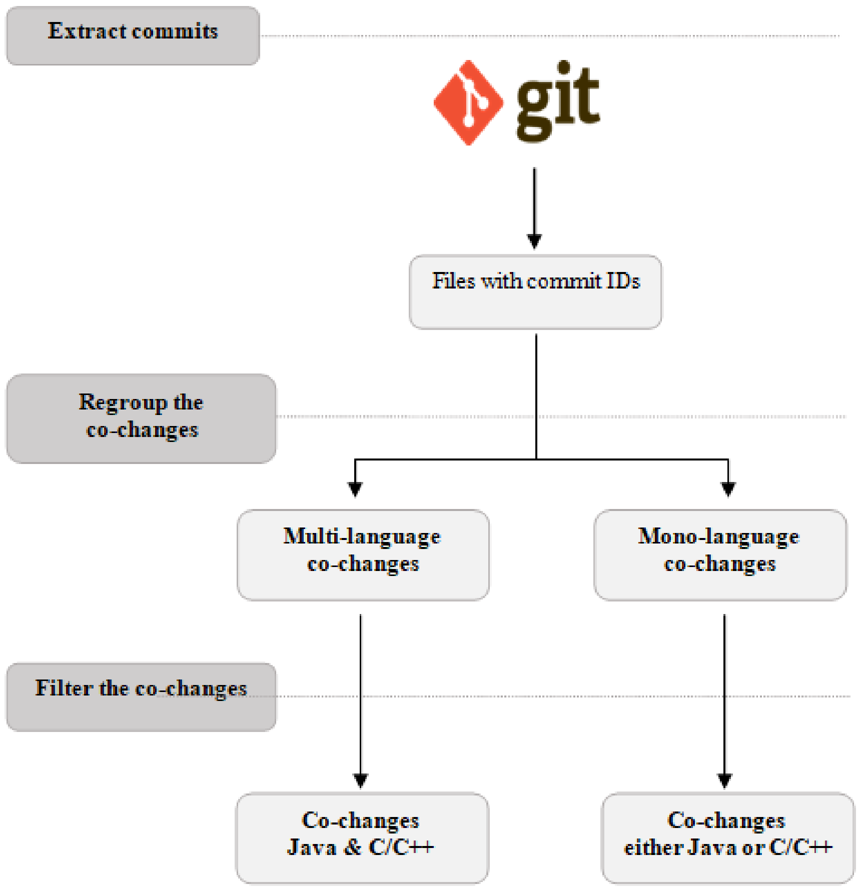


Figure 7.6 H-MLDA approach.

the relation between DILD and IILD with the system quality (introduction of bugs) and the system security (introduction of security vulnerabilities). To achieve this, we collected the bug reports of the studied systems (during the collection step of the systems, we ensured that their respective bug reports as well as their commit messages are accessible).

We take advantage of the existing SZZ algorithm [100] to identify bug-introducing dependencies. The SZZ algorithm identifies changes that are likely to introduce issues, it uses the issue report information to find such bug-introducing changes. Using the results of the SZZ algorithm, we prepared an issue data-set that contains essential information about the bug such as its bug ID, files affected, when it was reported and fixed.

Regarding security issues, we analyzed five vulnerabilities : memory faults, null-pointer exceptions, initialization checkers, race conditions, and access control problems [102]. We implemented a script where we used these vulnerabilities as search keywords (in the collected bug reports and commit messages) in addition to the following keywords : threat(s), vulnerabi-

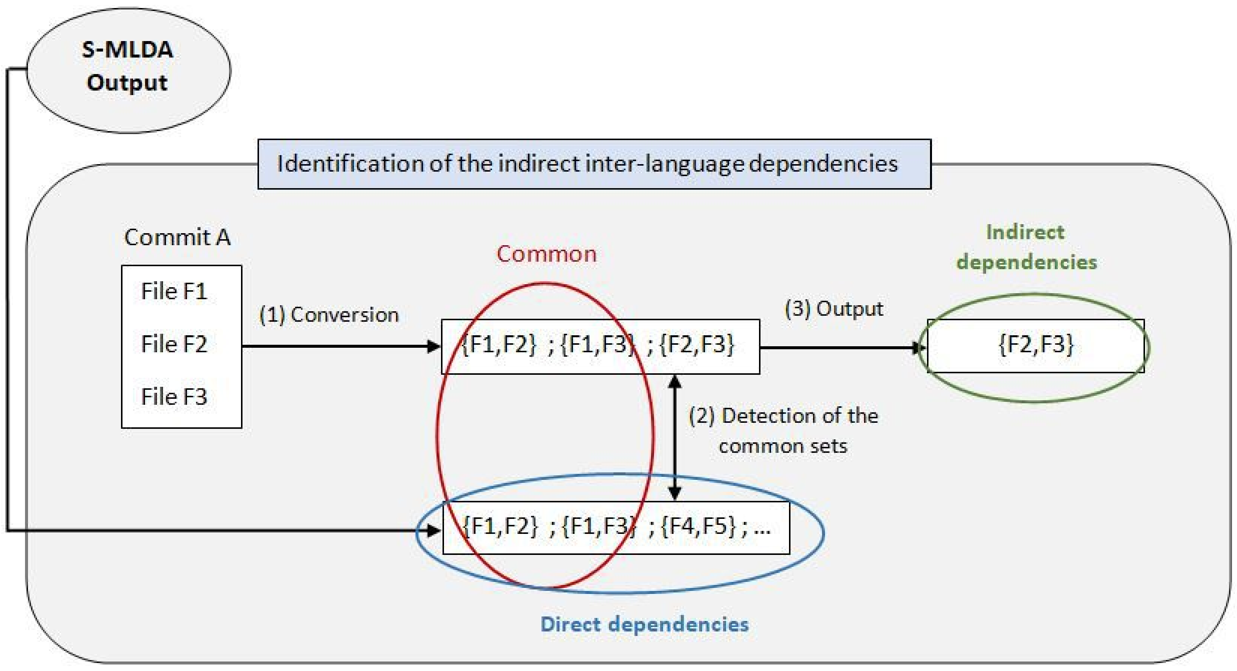

S-MLDA Output
Identification of the indirect inter-language dependencies
Commit A
File F1
File F2
File F3
(1) Conversion
Common
{F1,F2} ; {F1,F3} ; {F2,F3}
(3) Output
Indirect dependencies
{F2,F3}
(2) Detection of the common sets
{F1,F2} ; {F1,F3} ; {F4,F5} ; ...
Direct dependencies

Figure 7.7 Identification of indirect dependencies.

lity(ies), and security. It should be noted that, to increase the confidence in our results, none of the bug reports identified by the vulnerability search are present in the aforementioned issue data-set (the two data-sets are completely disjoint.

7.3 Results

The following section presents our results and summarizes them per research question.

RQ7.1. How common are direct and indirect inter-language dependencies in multi-language systems?

Approach : We present in Table 7.2 the results related to the commits identified by H-MLDA. We show the total number of the multi-language commits and the total number of the mono-language commits, with their respective percentage out of all the commits. The results of the inter-LD and intra-LD are illustrated in Table 7.3, where the two columns show the percentage of inter-LD and intra-LD identified out of the total number of multi-language commits. Table 7.4 shows (i) the number of DILD (identified by S-MLDA) and (ii) the number of IILD (generated via H-MLDA). Lastly, we evaluate the performance of S-MLDA and H-MLDA in order to validate their precision and recall.

Table 7.2 Number of Mono-/Multi-language Commits.

	Systems	#Total Commits	#Multi-language Commits	#Mono-language Commits
1	Conscrypt	3860	2952(76,47%)	908(23,53%)
2	RethinkDB	19898	13,002(65,34%)	6896(34,66%)
3	JatoVM	4135	2996(72,45%)	1139(27,55%)
4	Libgdx	13580	11,332(83,40%)	2248(16,60%)
5	Lwjgl	3884	2108(54,27%)	1776(45,73%)
6	Openj9	6219	4992(80,27%)	1227(19,73%)
7	React-native	16298	6772(41,55%)	9526(58,45%)
8	Realm	8172	5893(72,11%)	2279(27,89%)
9	Seven-Zip	909	432(47,52%)	477(52,48%)
10	VLC	11866	8676(73,11%)	3190(26,89%)

Table 7.3 Proportions of identified inter-LD and intra-LD.

	Systems	%Inter-LD (out of multi-language commits)	%Intra-LD (out of multi-language commits)
1	Conscrypt	70,59	29,41
2	RethinkDB	73,33	26,67
3	JatoVM	83,34	16,66
4	Libgdx	68,74	31,26
5	Lwjgl	51,61	48,38
6	Openj9	71,99	28,01
7	React-native	55,55	44,45
8	Realm	62,07	37,93
9	Seven-Zip	63,63	36,36
10	VLC	48,27	51,73

Results : We observed that **multi-language co-changes are common in multi-language systems, with values ranging between 47,52% and 83,40% (relative to the total number of commits)**. Developers are changing files written in diverse languages at the same time, which indicates a strong logical coupling between these multi-language files. Multi-language commits involve more than 50% of inter-LD in 90% of the systems (except the case of VLC where the %inter-LD is 48,27%). The values range between 48,27% and 83,34%.

The results from Table 7.4 show that **the number of indirect inter-language dependencies is higher (average of 2.7 times) than the number of direct inter-language dependencies in 90% of the cases (nine systems), while it is nearly equal for**

the case of Libgdx. The high number of indirect inter-language dependencies can be precarious for system maintenance activities ; since these dependencies are hidden from static code analysis tools, any change to them can negatively impact the system.

During the IILD identification (*i.e.,* generation of sets of files from co-changes and removal of the common sets with DILD), **we found that all of the DILD (*i.e.,* the sets of files Java and C/C++) were included *i.e.,* a part of in the multi-language co-changes** (SMLDA $\subset HMLDA$).

Evaluation of the accuracy of H-MLDA and S-MLDA results : We discuss the accuracy of the results of H-MLDA and S-MLDA by evaluating the precision and the recall.

We manually evaluate the precision by randomly selecting a sample of data for each approach (using the sampling methodology in [103]). To select the sample, we set a confidence level of 95% and an error margin of 5%. Our final samples contained 379 DILD for S-MLDA and 382 IILD for H-MLDA.

1. Case of S-MLDA : We manually checked the source code of the direct inter-language dependencies sets for the existence of one or more of the following JNI elements as they identify the existence of JNI source code [27] : JNI header (*i.e.,* #include `<jni.h>`) ; JNI pointer (*i.e.,* JNIEnv) ; JNI keyword (*i.e.,* native(; and JNI functions (*i.e.,* FindClass(), GetMethodID(), etc.).
2. Case of H-MLDA : We manually reviewed the source code to verify that no JNI dependencies were present in the sample of 382 IILD. Then, we validated the file dependencies based on one of the following elements :
   — Similarity of the files names. We checked if the IILD files in these sets could have a same or similar names *e.g.,* a sub-string of a file name A included in file B.
   — The intent of the source code files. We reviewed the source code and the comments inside for each set *i.e.,* Java and C(++), to find if there is a behavior between the files that could explain the indirect dependency.
   — Existence of external information sources. We searched in the bug reports and developers discussions if the files indirectly dependent were involved in the same issue or were reported as related.

For the recall, we considered all the sets (DILD and IILD) presented in Table 7.4.

1. Case of S-MLDA : We implemented a script to count occurrences of the JNI header presented in all the source code and compared it with the number of JNI headers found in C(++) files involved in the IILD sets.
2. Case of H-MLDA : We implemented a script to identify all the inter-language depen-

dencies sets that have similar names and that are not JNI. From the sets found, we excluded the sets successfully detected by H-MLDA. The remaining ones are the inter-language dependencies not detected by H-MLDA which are considered in calculating the recall.

The final results show a precision of 100% and a recall of 78% for S-MLDA, and a precision of 68% and a recall of 87% for H-MLDA.

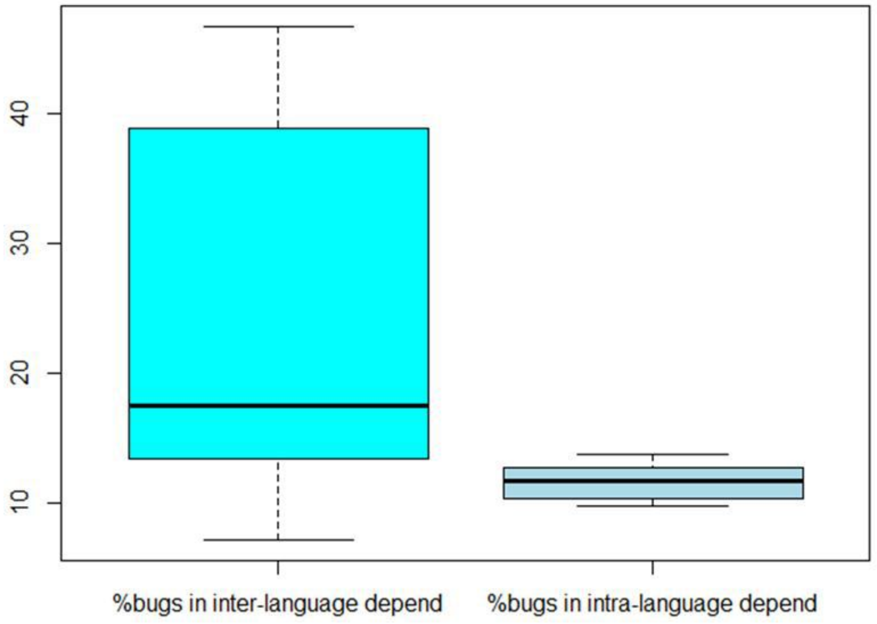


Figure 7.8 Percentage of Buggy dependencies.

RQ7.2. Are inter-language dependencies more risky for multi-language software system in terms of quality?

Approach : Considering the result of RQ7.1 (*i.e.,* S-MLDA is a subset of H-MLDA), in the following, we studied the interaction between the (intra)inter-language dependencies and the quality issues.

We show in Table 7.5 the percentage (out of %inter-LD and %intra-LD) of the buggy inter-/intra-language dependencies. **Results : The percentage of bug-introducing co-changes was as high as 46,66% in inter-language dependencies and 13,7% in the case of intra-language dependencies**. We report that when the number of inter-language dependencies increased, the number of bug-introducing commits increased with a significant correlation of 0,918 and vice-versa. Conversely, we observe that bug-introducing co-changes

Table 7.4 Number of (In)Direct Inter-language Dependencies.

	Systems	#DILD (S-MLDA)	#IILD (H-MLDA)
1	Conscrypt	1341	2827
2	RethinkDB	4321	8299
3	JatoVM	1128	3154
4	Libgdx	6232	5803
5	Lwjgl	733	3149
6	Openj9	4438	7364
7	React-native	2116	6416
8	Realm	2266	6177
9	Seven-Zip	174	513
10	VLC	3172	9004

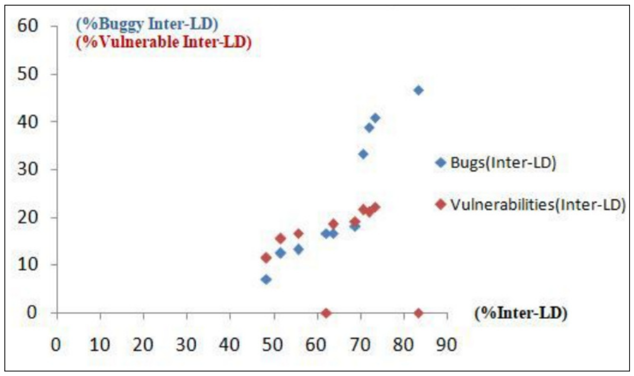
Table 7.5 Percentage of Bugs and vulnerabilities within co-changes.

Systems	%Buggy Inter-LD	%Buggy Intra-LD	%Vulnerable Inter-LD	%Vulnerable Intra-LD
Conscrypt	33,33	12,03	21,66	5,02
RethinkDB	40,90	10,72	22,18	3,76
JatoVM	46,66	11,42	0	9,44
Libgdx	18,18	10,33	19,27	8,33
Lwjgl	12,5	13,7	15,5	4,31
Openj9	38,88	12,66	21,11	7,83
React-native	13,33	9,77	16,66	3,52
Realm	16,66	11,82	0	0
Seven-Zip	16,66	9,78	18,57	11,27
VLC	7,14	12,87	11,45	0

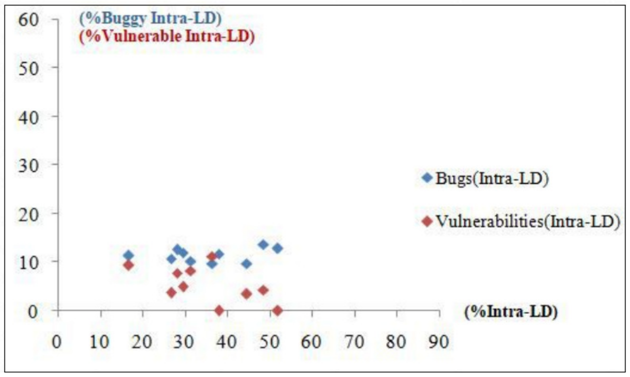
are constant for intra-language dependencies, with values range between 9,77 and 13,70. Hence, there is no significant correlation between bugs and intra-language dependencies.

The box-plot in Figure 7.8 shows the difference between the median of each set *i.e.,* bugs in inter-LD and bugs in intra-LD. Moreover, the scatter-plot in Figure 7.9 allows a better analysis to answer the research question with the following : **The more the X axis in Figure 7.9a increases for inter-language dependencies, the higher the risk of bugs (blue color) being introduced, while this risk remains constant for intra-language dependencies (Figure 7.9b).**

We used the Mann-Whitney U test [104] with a 95% confidence level (*i.e.,* $\alpha = 0.05$) to determine if there is a significant difference between inter-LD and intra-LD in terms of bugs. The test shows a significant difference **(p-value = 0.017)** between the percentage of bugs



(a) Inter-language dependencies



(b) Intra-language dependencies

Figure 7.9 Percentage of Quality and security issues detected in intra- and inter-language dependencies.

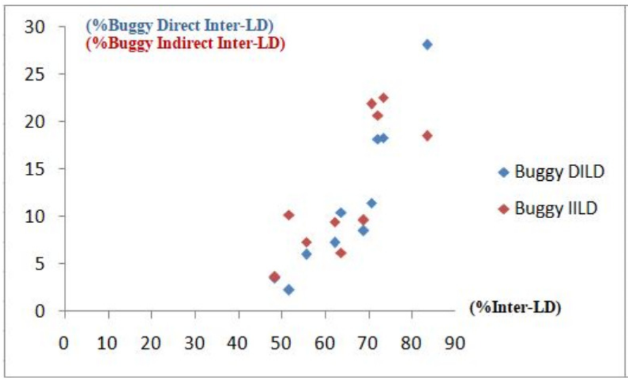
presented in the inter-LD and the percentage of bugs presented in the intra-LD.

The results in Figure 7.9b show no correlation between bugs and intra-LD. Thus, we focus our next analysis on buggy (in)direct inter-LD. We illustrate the obtained results in a scatter-plot presented in Figure 7.10a. We can observe that **the more the X axis increases for inter-LD, the higher the risk of quality bugs introduced in both indirect inter-LD(IILD) and direct inter-LD(DILD)**.

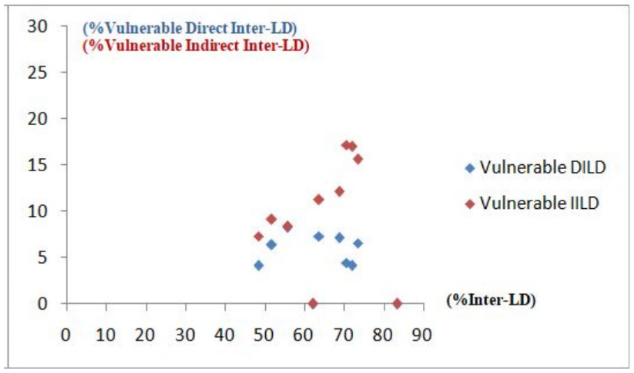
RQ7.3. Are inter-language dependencies more risky for multi-language software system in terms of security?

Approach : Security software vulnerabilities are weaknesses in software systems that can be exploited by a threat actor, such as an attacker, to perform unauthorized actions within a computer system. Software vulnerabilities can be defined as incorrect internal states detected in the software source code that could allow an attacker to compromise its integrity, availability, or confidentiality. Most software security vulnerabilities fall into one of a small set of categories :

— Memory faults : such as buffer overflows and other memory corruptions which impact the security of a software system.
— Null-pointer exceptions : which can threat the system confidentiality when it is used to reveal debugging information.
— Initialization checkers : which can threat the system integrity when the software components are used without being properly initialized.
— Race conditions : related to weak time checking between software tasks and can allow



(a) Buggy DILD and IILD



(b) Vulnerable DILD and IILD

Figure 7.10 Percentage of Quality and security issues detected in (in)direct inter-language dependencies.

an attacker to obtain unauthorized privileges.

— Access control problems : related to weak specifications of privileges defining the access or the modification of software files .

We show in Table 7.5 the percentage (out of inter-LD and intra-LD) of the dependencies involving vulnerabilities.

Results : We observe that 80% of the studied systems revealed security issues introduced within inter- and intra-language dependencies.

The percentage of vulnerabilities in inter-language dependencies can reach up to nearly 22,18%, and 11,27% in intra-language dependencies. We present the findings in a scatter-plot, Figure 7.9, for a better visual analysis. We observe that **the more we have inter-language dependencies, the higher the risk of vulnerabilities being introduced**. Without considering JatoVM and Realm (where vulnerabilities could not be found), a correlation of 0,961 was found between inter-language dependencies and security vulnerabilities.

The box-plot presented in Figure 7.11 and the p-value of the Mann-Whitney U test **(p-value = 0.007)** shows a significant difference between the percentage of vulnerabilities in inter-LD and the percentage of vulnerabilities in intra-LD. Regarding the difference between the risks of vulnerabilities in direct inter-LD and indirect inter-LD, we report from Figure 7.10b that **the more we have inter-LD the higher is the risk of vulnerabilities introduced in indirect inter-LD comparing with direct intra-LD where it remains nearly the same.**

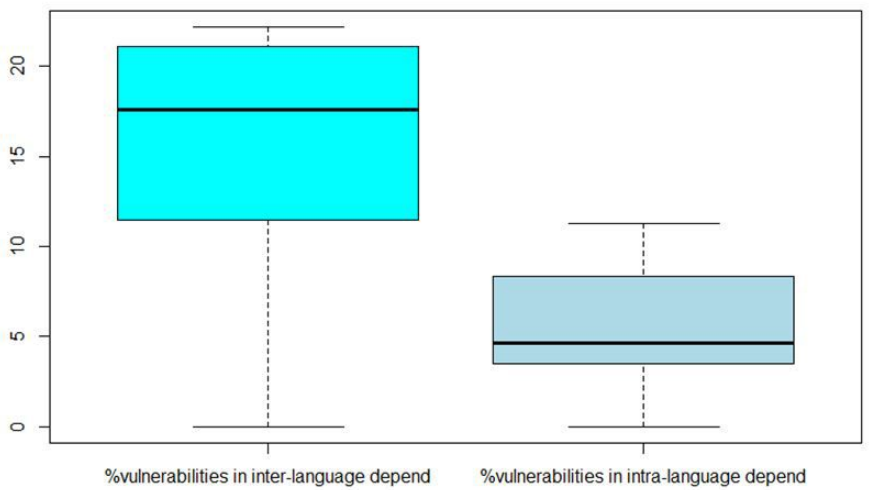


Figure 7.11 Percentage of Vulnerable dependencies.

7.4 Discussion

we discusses now the empirical study performed on ten open source Java Native Interface to analyze the inter-language dependencies.

RQ7.2. Are inter-language dependencies more risky for multi-language software system in terms of quality? The percentage of quality issues in inter-language dependencies (46,66%) is higher nearly to three times compared with quality issues in co-changes involving intra-language files (13,7%). Several previous works suggested that combining programming languages presents always a challenging activity as it increases the complexity of the software and leads to hard maintenance [9]. Thus, analyzing the impact of a change through multi-language files is important to avoid software issues. In many cases, indirect dependencies could be risky to the quality of the system as these kinds of dependencies are hard to identify via static analysis. Our results show that the risk of introducing bugs are 1.5 times higher in indirect inter-LD (not detectable by static analysis) than direct ones.

The following are two examples of bugs introduced within inter-language dependencies.

— The first example was extracted from *Conscrypt*. It presents a co-change that involved four files written in Java and C++. It was responsible for the introduction of a bug because of a miss of a change in the native function NativeCrypto.EVP_DigestVerifyFinal() implemented in org_conscrypt_NativeCrypto.cpp. The author of the change modified the signature of the native Java method EVP_DigestVerifyFin and missed the corresponding modification in the CPP file that results from the inter-language dependency. We explain it by the fact that the author of the change was not able to identify the JNI dependencies and to track the change propagation *i.e.,* a miss of the dependency analysis.

— The second example concerns a co-change extracted from *Realm.* It involved inter-language dependencies with a total of six Java files and three CPP files. Our analysis shows that this change introduced a bug when the new return type of the Java native method in Realm.java did not match with the old return type of the corresponding implementation of CPP file. These kinds of changes are subject to bugs especially when several changes are involved (*i.e.,* on nine files that were indirectly dependent). JNI practices [1] should be well mastered by the developers as they present another way to protect the source code from quality decrease.

RQ7.3. Are inter-language dependencies more risky for multi-language software system in terms of security?

From the results, we observe that vulnerabilities in inter-language dependencies (22,18%) are twice as common as in intra-language dependencies (11,27%). This leads to the conclusion that inter-language dependencies are more risky than intra-language dependencies and developers should consider the challenge provided with multi-language systems.

We observe from Table 7.5 that no vulnerabilities were found in inter-language dependencies of Realm and JatoVM. Realm is written mostly in Java (82.3%) where C++ presents 8%. Java does memory management automatically, the compiler catches more compile-time errors, and it does not allocate direct pointers to memory. We observed that the selected software written mostly in Java is less vulnerable than C or C++ to memory security vulnerabilities. Indeed, a similar observation was reported previously by other researchers [105]. The results of Lwjgl are similar to Realm. Lwjgl is mostly written in Java (85,1%) but the difference is that the second language used is C and not C++ (Object-oriented programming (OOP)). We are exploring in an ongoing work if the fact of having dependencies among files written in Java and procedural programming language instead of an OOP language may increase the existence of vulnerabilities in multi-language systems.

JatoVM is the implementation of the Java virtual machine. Vulnerabilities have not been found within inter-language dependencies, however, they were detected within intra-language dependencies. JatoVM is written mostly in the C language (73%). C is a low-level programming language that provides access to low-level IT infrastructure. We noticed that previous researchers reported that manipulating C language is critical in the software security context [106]. Conduction further empirical studies can better explain the fact that we are founding several security vulnerabilities propagated between C files. In future work, we will study the reasons behind not finding vulnerabilities within the inter-language dependencies to investigate whether if this low vulnerabilities is related to the architecture of the system *i.e.,* how it is designed or if it is related to the domain, as it is a java virtual machine.

Figure 7.12 shows the distribution of the percentage of the vulnerabilities for each category *i.e.,* Memory faults, Null-pointer exceptions, Initialization checkers, Access control problems, Race conditions. We can observe from the bar-plot that **the most vulnerabilities in inter-LD are the Memory faults and the Access control (presented in 80% of the systems) followed by Race conditions (presented in 40% of the systems) while the rest are shown in less than 30% of the systems**. However, for intra-LD, Memory faults and Initialization checkers are presented in 50% of the systems while Null-pointer exceptions were found in 40% of the systems and the rest are under of 30%. Abidi *et al.* through their study [84], support this finding as Memory faults *e.g.,* buffer_overflow are the most known vulnerability subject of security issues in multi-language systems. The pool of practitioners who participated in that survey explained this fact by claiming that these programming languages do not provide security protection against overwriting data in memory and do not automatically check that data written to an array is within the boundaries of that array. Moreover, Tan *et al.* [107] discussed the importance of caring about violating access control rules in JNI systems as native methods can access and modify any memory location in the heap.

7.5 Threats to validity

Threats to internal validity We relied on the literature to extract the JNI rules and to apply the co-changes method. We evaluate the accuracy of the approaches used through the study where we found precision (recall) values of 100% (68%) for S-MLDA and 68% (87%) for H-MLDA.

Threats to construct validity : The process followed in collecting the data may introduce some inaccuracies. The use of PADL meta-model, co-changes method, SZZ algorithm, and vulnerability classification may present a threat to construct the study and it may exist other means. However, we mentioned that many previous works relied on these means and validated them. For this reason, the precision and the recall of these means is a concern that we agree to accept.

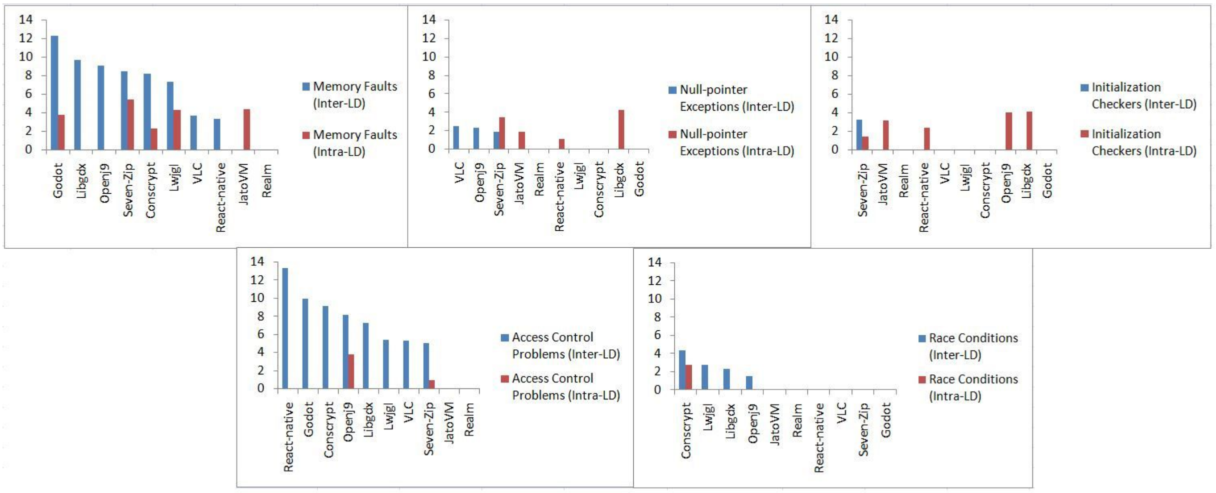


Figure 7.12 Distribution of the five security vulnerabilities categories.

Threats to external validity Our results may not be representative of all multi-language software systems, since we only studied the case of JNI on ten open-source systems. We accept this threat as software system' characteristics could vary depending on different criteria. However, to mitigate this threat, we varied the selected systems based on the software status, size, and languages used.

7.6 Chapter Summary

Building on the survey's findings from the previous chapter, this chapter focused on dependency analysis, the most challenging analysis for change impact analysis within multi-language systems. Thus, we analyzed inter-language dependencies *i.e.,* dependencies between components written in different programming languages and their relation with the software quality and security. We empirically applied two approaches based on historical dependency analysis (H-MLDA) and static dependency analysis (S-MLDA) on ten open-source multi-language systems (i) to identify the inter-language dependencies and to (ii) study their impact on the quality and the security of multi-language systems.

We evaluated the accuracy of the results and we found precision (recall) values of 100% (68%) for S-MLDA, and 68% (87%) for H-MLDA. We find that IILD are 2.7 times more common than DILD. The more inter-LD, the higher the risk of bugs and vulnerabilities, while this risk remains constant for intra-LD. The number of bugs introduced in inter-LD is three times higher than in intra-LD while the number of security vulnerabilities introduced in inter-LD are two times higher than in intra-LD. We recommend to multi-language developers to first use specific approaches (*i.e.,* S-MLDA) during their multi-language changes to identify the inter-LD, especially the (hidden) indirect ones. Secondly, they are recommended to take care of the quality aspect while changing a multi-language software system as the impact is higher than in the mono-language software systems *i.e.,* specific approaches are needed to help spot quality issues during the change tasks.

CHAPTER 8 WHAT IS THE IMPACT OF MULTI-LANGUAGE ADOPTION IN MACHINE LEARNING FRAMEWORKS VS. TRADITIONAL SYSTEMS ? (Sub-hypothesis 3)

8.1 Chapter Overview

In the previous chapters, we studied the prevalence of multi-language development within traditional systems *i.e.,* non-Ml systems, its associated challenges with respect to change impact analysis, and its impact on the quality and security of systems. In this chapter, we extend these prior analyses to the domain of machine learning applications.

Given the complex nature of machine learning (ML) frameworks, the artificial intelligence (AI) community has been taking advantage of multiple languages in a single machine learning framework. In particular, while Python has evolved into the most commonly used language for developing machine learning frameworks due to its large range of powerful features [65], there are some inconveniences of using Python alone *i.e.,* it lacks computational performance needed for high-frequency real-time predictions [66], it takes significant CPU time for interpretation, etc. Hence, the Python C extension is often used as a solution to interface with highly performant C code for frequently executed low-level algorithms, as required, for example, by the gaming industry [67], multi-agents [68], and so on.

Despite the benefits of using multi-language development in developing machine learning frameworks, there are also several challenges associated with it as we already showed in previous chapters on the challenges of multi-language development in traditional systems. The major issue concerning this paradigm is that multi-language programs do not necessary obey the semantics of the combined languages [68] and it is the developer's responsibility to deal with the different programming calling conventions to avoid introduction of diverse issues that can harm the software. Such concerns are not necessarily new, since multi-language development has been used for a long time for developing more traditional systems as discussed in our systematic literature review (Chapter 4) and by several works in the literature [10, 78, 85].

However, in the case of ML frameworks, the traditional issues of multi-language development are further corroborated by the inherent complexity of ML frameworks. For example, they implement highly specialized mathematical operations that are challenging to test and debug, and even require interdisciplinary collaboration between scientists and developers [108, 109]. Hence, how multi-language development is adopted in machine learning frameworks ? Are these frameworks following the multi-language trend ? Does the practice of multi-language development increase the difficulty of dealing with machine learning frameworks ?

To the best of our knowledge, in this chapter we present the first study that investigates the prevalence and the impact of multi-language development on the development of machine learning frameworks in terms of their ability to solicit high-quality open source contributions. More specifically, we study the impact of multi-language development in terms of the volume, acceptance rate, review process duration, and bug-proneness of pull requests (PR). Thus, we empirically analyze the ten largest open source multi-language machine learning frameworks (Cat-I) and the ten largest open source traditional systems (Cat-II). In addition, we considered a set of seven mono-language open source machine learning frameworks (Cat-III) that served as a control group for the comparison between Cat-I and Cat-II. We address the following research questions :

— **RQ8.1.** What is the prevalence of multi-language development in machine learning frameworks?
— **RQ8.2.** What is the impact of multi-language development on pull request acceptance in machine learning frameworks?
— **RQ8.3.** What is the impact of multi-language development on the time taken to accept pull requests in machine learning frameworks?
— **RQ8.4.** Are multi-language pull requests more bug-prone than mono-language pull requests in machine learning frameworks?

8.2 Methodology

This section discusses our methodology to empirically analyze the impact of multi-language development on open-source machine learning frameworks. An overview of our methodology is presented in Figure 8.1.

8.2.1 Project selection and cloning

In this empirical study, we analyzed a total of 27 open source projects hosted on GitHub. Our selected projects include the ten largest multi-language machine learning frameworks and seven mono-language machine learning frameworks identified by Braiek *et al.* 's study [65], as well as the ten largest multi-language traditional systems from Grichi *et al.* 's study [1]. The seven mono-language machine learning frameworks serve to control for bias and any confounding factors in our comparison of multi-language machine learning and multi-language traditional systems.

We clone each project from GitHub and extract the following information : total number

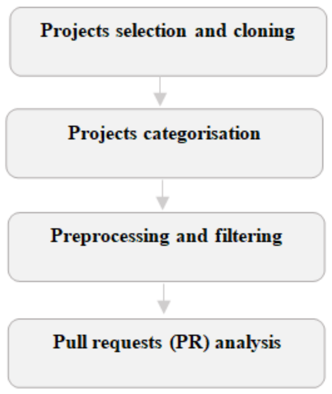

Projects selection and cloning
Projects categorisation
Preprocessing and filtering
Pull requests (PR) analysis

Figure 8.1 Overview of the methodology.

of lines of code, all pull requests (PR), all commits, and the percentage of programming languages used.

8.2.2 Project categorisation

For clarity, the selected projects are grouped into 3 categories : Cat-I constitutes the ten largest multi-language open source **machine learning** frameworks, Cat-II constitutes the ten largest multi-language **traditional systems**, and Cat-III constitutes the seven **mono-language** open source machine learning frameworks. Table 8.1 gives an overview of the three categories.

8.2.3 Preprocessing and filtering

Accepted and Rejected Pull requests — We categorize pull requests as either accepted or rejected based on the pull request status (*i.e.,* Merged, Closed). Pull requests with both closed and merged status are classified as accepted pull requests. We identify a rejected pull request as being closed but not merged. We do not consider open pull requests in this study as they are still under review. For the rest of the chapter, we call accepted pull requests “accept-PR" and rejected pull requests “reject-PR".

Multi- and Mono-language Pull requests — We identify the set of changed files for each commit linked to a pull request, as well as the programming language(s) used in each file. A pull request that has at least one multi-language commit is considered as a multi-language pull request. Conversely, a pull request with no multi-language commit is considered a mono-language pull request. A commit is tagged as multi-language if it has files written in

Table 8.1 Selected case study projects, grouped by category and, per category, ordered from largest to smallest in terms of total number of lines code.

	Project	#Code lines	#Commits	#Pull Requests
Cat-I	Spacy	6.02M	10,382	1057
	Tensorflow	2.49M	61,240	12,393
	Pytorch	817K	19,559	16,999
	Incubator-mxnet	414K	9869	7,965
	CNTK	327K	16,108	547
	Paddle	290K	24,724	10,858
	Caffe2	275K	3680	1260
	Theano	155K	28,081	4094
	Scikit-learn	153K	24,299	7971
	Caffe	76,3K	4154	2204
Cat-II	NativeScript	1.93M	16,150	2435
	Openj9	1.47M	8239	4519
	Godot	1.15M	19,898	12,057
	Libgdx	830K	13,580	2779
	RethinkDB	486K	33,402	363
	Mapbox GL	399K	14,976	7707
	React-native	395K	18,038	8623
	Play !framework	394K	14,059	1943
	RocksDB	346K	8341	4012
	VLC	196.1K	11,866	1884
Cat-III	Nltk	228K	13,884	1128
	Keras	50.8K	5342	3918
	Neon	49.4K	1118	88
	Torch7	29K	1337	510
	Pattern	23.6K	1433	118
	Tflearn	10.4K	605	247
	Sonnet	7.42K	764	39

more than one programming language, while it is tagged as mono-language when it has files written in only one programming language. For example, a pull request *P1* that contains two commits *C1 (file1.java, file2.c)* and *C2 (file3.c, file4.c)* is considered as a multi-language pull request. A pull request *P2* that contains two commits *C3 (file5.java, file6.java)* and *C4 (file7.c, file8.c)* is considered as a mono-language pull request.

An alternative definition of multi-language PR would have been "any PR for which the union of changed files covers at least two programming languages". However, according to the rules for inter-language dependencies (*e.g.,* , between Java and C), the (multi-language) dependant files should change together in order to compile and run, hence our commit-level definition is more realistic. For the rest of the chapter, we call multi-language pull requests "multi-PR" and mono-language pull requests "mono-PR".

8.2.4 Pull request analysis

Pull request acceptance period — We calculate the period spent (in hours) for a pull request to be accepted or rejected based on the difference between the pull request submission date and when the pull request was closed.

Bug-inducing pull requests — We collect the log messages of all pull requests and their contained commits. We split each message into words, the search for keywords and references to bug reports. Examples of the common keywords used were : '"fix", "correct", "bug", "error", "issue", "mistake", "blunder", "incorrect", "fault", "defect", "flaw", "bugfix", "bugfix :"'. As soon as a pull request was found to refer to a bug report, it was considered to be a bug fix.

Then, once we identified the pull requests that contain a fix to a bug, we applied the SZZ algorithm [100] to determine the initial bug-introducing pull request. The SZZ algorithm uses git-blame on the revision history to identify commits that are likely to have introduced bugs to determine first what changed in the bug-fix, then to locate the origins of the deleted or modified source code change that introduced this bug [110]. Finally, all identified bug-introducing pull requests are tagged automatically and assigned to the right group (multi-PR or mono-PR). We statistically compared the bug-introducing multi-PR to the bug-introducing mono-PR.

8.2.5 Statistical tests

We use the non-parametric Mann-Whitney U statistical test [104] with a 95% confidence level (*i.e.,* $\alpha = 0.05$). We considered Bonferroni correction [111] to control the family-wise error rate when we perform more than one comparison on the same data. According to this correction, we divide the confidence level α by the number of tests. We also compute the Cliff's Delta effect size [112] if a significant difference is obtained. An effect size, r, is classified as "negligible" if r<0.2, as "medium" if 0.2<r<0.5, and as "large" if 0.5<r<0.8.

The larger the effect size the stronger the relationship between the two variables. Table 8.2 shows the obtained p-values and effect sizes of the various tests used in this study.

8.3 Results

The following section presents our results and summarizes them per research question.

RQ8.1. What is the prevalence of multi-language development in machine learning frameworks?

Approach : This research question aims to identify the presence/absence of the practice of multi-language development in ML frameworks. We investigate the different languages used and the prevalence of multi-language contributions (pull requests) in machine learning frameworks, then compare the results with those of multi-language traditional software systems.

Results : Both Cat-I and Cat-II projects have similar percentages of main programming languages involved, *i.e.,* the two sets of projects are comparable. Figure 8.2 shows that the distribution of programming languages involved in the studied Cat-I (Figure 8.2a) and Cat-II (Figure 8.2b) projects are similar. We found that regarding Cat-I, the main languages are Python and C, while Java and C/C++ are the main languages for Cat-II. Other languages, especially Objective-C and Perl are the least common for both categories.

Cat-I and Cat-II projects are also comparable in terms of the total number of PRs and the number of multi-language PRs.

Figure 8.3 presents the total number of pull requests (PR) in the studied Cat-I and Cat-II systems, respectively. The number of pull requests of Cat-I is not significantly different from Cat-II (p-value=0.6305) *i.e.,* the categories are similar to each other. As shown in Figure 8.4, we observe that Cat-I and Cat-II systems have a similar proportion of multi-language pull requests (multi-PR) : both Cat-I and Cat-II have the same median (39,08 for Cat-I and 39,09 for Cat-II), while the variance of Cat-I is larger.

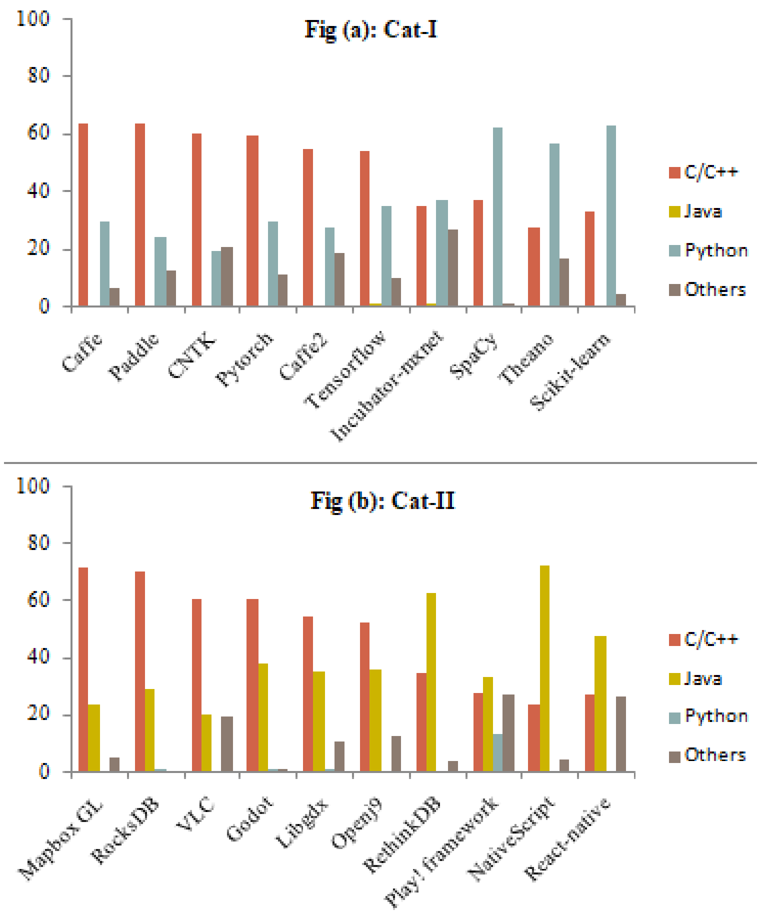


Figure 8.2 Percentage of the programming languages used.

The sets of Cat-I and Cat-II are comparable according to the usage of multi-language development (Figure 8.2), the number of pull requests (Figure 8.3), and the percentage of multi-language pull requests (Figure 8.4).

RQ8.2. What is the impact of multi-language development on pull request acceptance in machine learning frameworks?

Approach :

Existing research shows that multi-language development requires substantial additional effort from software developers [1]. Rahman and Roy [113] also report that programming languages involved in pull requests can influence the success and failure rates of the pull requests. Since pull requests represent the process through which a collaborator contributes in a software project, this research question aims to study the impact of multi-PR and mono-PR on the pull request acceptance rate of Cat-I projects. We compare our results to those of

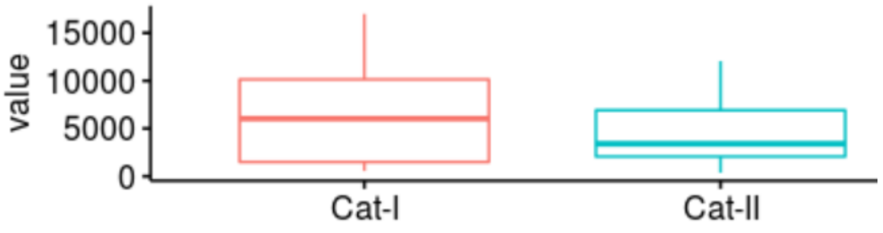


Figure 8.3 Total number of pull requests.

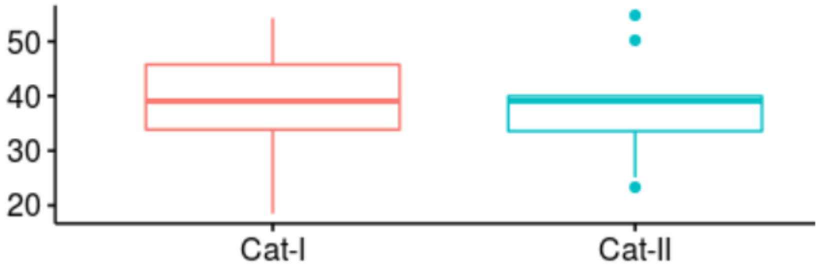


Figure 8.4 Distribution of the percentage of multi-language pull requests

Cat-II and Cat-III (mono-PR only) projects.

Results : There is no significant difference between the proportion of accepted pull requests in both Cat-I and Cat-II. Figure 8.5 shows the total percentage of all accepted pull requests (both multi-PR and mono-PR) in the two system categories. While the Figure shows that the acceptance rate in Cat-II generally exceeds the acceptance rate in Cat-I, the Mann-Whitney U test shows an insignificant difference with a p-value = 0,16. Hence, both Cat-I and Cat-II are, in general, equally likely to accept PRs.

Given this inconclusive result, we performed further analysis to compare the acceptance rate of multi-PR and mono-PR (relative to the totality of pull requests).

There is no significant difference between the acceptance rate of either multi/mono-PRs between Cat-I and Cat-II. We observe from Figure 8.6a that Cat-II systems have a generally higher multi-PR acceptance rate than Cat-I systems. Also, mono-PR in Cat-II seems to have a higher acceptance rate than mono-PR in Cat-I, as shown in Figure 8.6b. However, the Mann-Whitney U test did not show a significant difference in either the multi-PR acceptance rate (p-value=0,85) or the mono-PR acceptance rate (p-value=0,53) comparisons between Cat-I and Cat-II systems. This finding shows that ML (Cat-I) and non-ML (Cat-II)

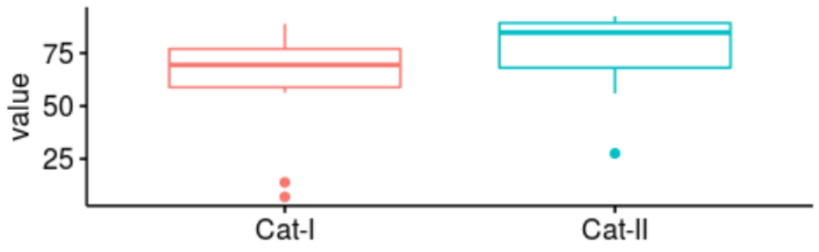


Figure 8.5 Percentage of accepted pull requests.

systems have similar acceptance rates even at the finer granularity of mono- and multi-PRs.

In Figure 8.6b, we further compare the acceptance rate of mono-PR in Cat-III with the mono-PRs in Cat-I and Cat-II. We observe that the median percentage of accepted mono-PR of Cat-III systems is lower than that of Cat-II systems, but higher than the median accepted mono-PR of Cat-I systems. The Mann-Whitney U test shows that these differences are significant, with p-values of 0,0068 (with a large effect size of 0,6) and 0,025 (with a medium effect size of 0,5), respectively.

Mono-PR have a significantly higher acceptance rate than multi-PR in Cat-I systems, while there is no such difference in Cat-II systems. Multi-PR in Cat-I and mono-PR in Cat-I show a significant difference with a p-value of 0,022 and a small effect size of 0,2. However, multi-PR and mono-PR in Cat-II did not show a significant difference (p-value = 0,35).

Multi-language pull requests are significantly harder to get accepted in ML frameworks than in traditional software systems.

RQ8.3. What is the impact of multi-language development on the time taken to accept pull requests in machine learning frameworks?

Approach :

While the previous RQ's observations in terms of PR acceptance rate are able to provide some insights, they do not tell the full story, since a multi-language PR that took a lot of time and effort to be accepted might still indicate a kind of overhead imposed by multi-language development. This motivated us to investigate the time taken for a multi- or mono-PR to be accepted or rejected. Since the time until a PR is accepted/rejected could be impacted

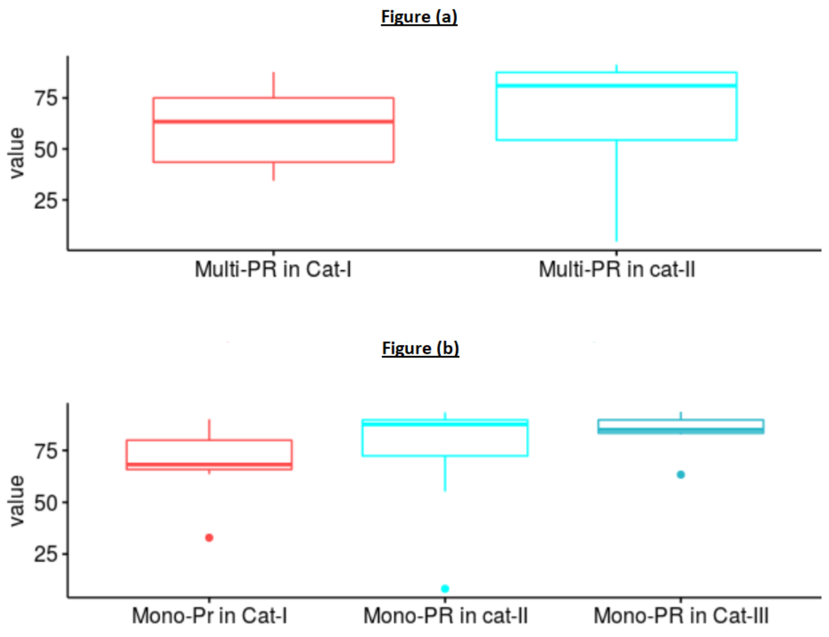


Figure 8.6 Acceptance rate in multi-/mono-language pull requests.

by several factors, we control the time required with (1|) the effort required to review each specific PR (approximated by the number of changed files in the PR), as well as (2) the size of the developer community.

Results : Pull requests take longer to be rejected than accepted in both Cat-I and Cat-II, for both mono- and multi-PR.

Figure 8.7 shows the time taken (in hours) by a developer to accept or reject a mono-PR/multi-PR in all three software system categories. From the figure, we can see that the period taken by the reviewers to reject a mono-PR (multi-PR) in Cat-I or to reject a mono-PR (multi-PR) in Cat-II is higher than the period taken to accept them. These observations are confirmed by the Mann-Whitney U tests, which yield p-values of 0,00033 (0,0068) and 0,00032 (0,00049), respectively. The effect size shows a large effect in all cases of r=0,87 (r=0,65) and r=0,87 (r=0,84), respectively.

Rejecting a multi-PR takes longer than rejecting a mono-PR in both Cat-I and Cat-II. Figure 8.7 shows that reviewers spend a significantly longer amount of time to reject a multi-PR in Cat-I than to reject a mono-PR in Cat-I , with a p-value of 0,016 and a medium effect size of 0,54. A similar observation can be made for Cat-II systems : it took longer for reviewers to reject multi-PR in Cat-II than to reject a mono-PR in Cat-II, with p-values of 0,00073 and a large effect size of 0,75.

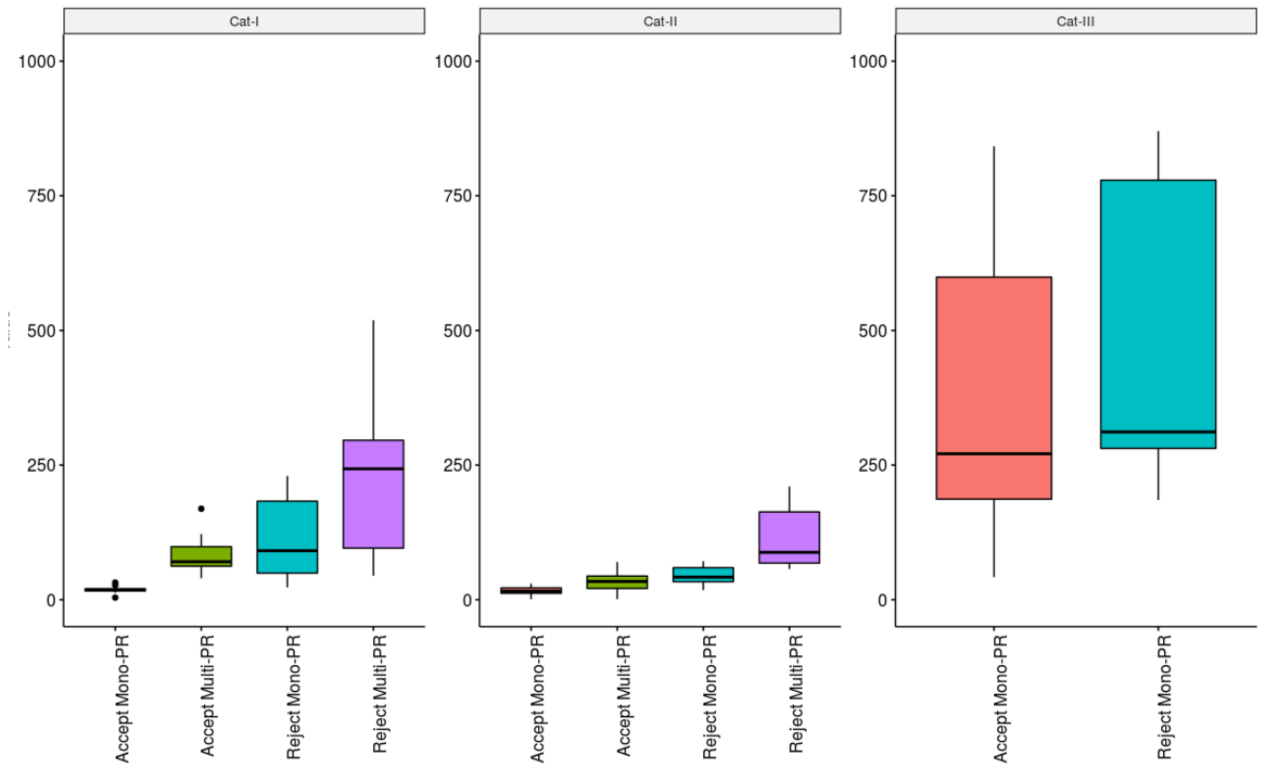


Figure 8.7 Period taken (in hours) to accept/reject a multi-/mono-PR. (It should be noted that two outliers were removed from Cat-III's Accept Mono-PR (value = 1352.5) and Reject Mono-PR (value = 1552.5) to improve the presentation of the figure.)

Only in ML frameworks, multi-PR take significantly longer to accept than mono-PR. While we obtained a p-value of 0,00018 (and a large effect size of 0,83) for the comparison of multi-PR and mono-PR acceptance times in ML frameworks (Cat-I), the p-value for the corresponding comparison in traditional software systems (Cat-II) did not show a significant difference (0,037. This finding shows that not only do multi-PRs have it harder to get accepted (as reported in RQ8.2), the PRs that are accepted generally take more time to do so as well.

We did not find significant differences between Cat-I and Cat-II in terms of the period taken to accept/reject mono-PRs. Hence, even though our comparisons within both categories showed some differences, the categories again are similar to each other, just like in RQ8.1.

ML frameworks (Cat-I) take longer to accept and reject a multi-PR than traditional systems (Cat-II). The period to accept (Reject) multi-PR in Cat-I is higher than Cat-II, with significant p-values : 0,0052 (0,023) with a large effect size of 0,62 and a medium effect size of 0,5, respectively.

The acceptance (rejection) of mono-PRs in Cat-III systems take equally long (and significantly longer) than the corresponding periods of Cat-I and Cat-II, respectively. We perform further analysis based on the mono-PR of Cat-III systems to understand whether earlier findings in this RQ apply across all ML frameworks (since Cat-III contains only mono-language pull requests). The Mann-Whitney U tests comparing the mono-PR acceptance/rejection times of Cat-III to the corresponding Cat-I and Cat-II times all are significant (see Table 8.2 for details). However, there is no significant difference between the mono-PR acceptance and rejection times within Cat-III (p-value of 0.41).

These findings suggest that the observed differences in terms of the period to accept multi-PR compared to mono-PR for Cat-I are not necessarily due to the fact that the ML domain is more complex, since Cat-III systems seem to suffer much more from longer accept/reject periods than Cat-I/II. One possible confounding factor might be that the Cat-III systems receive larger pull requests than Cat-I/II, potentially explaining their slower review process. Hence, we explore this confounding factor next.

The more file changes in a PR, the longer its acceptance period and the shorter its rejection period (in both Cat-I and Cat-II). In Figure 8.8(a), we compare the relationship between the median number of changed files in multi-PRs (discussed in Section 8.2.3) and the accepted (rejected) periods of multi-PR in both Cat-I and Cat-II systems. We can see that when the median number of multi-language files increases (X axis), the period spent (Y axis) to accept a multi-PR in Cat-I (red dots) and multi-PR in Cat-II (blue dots) increases as well. Spearman rank correlation showed a strong positive relationship with the following respective coefficient, p=0,88 and p=0,84.

However, the period spent to reject a multi-PR in Cat-I (yellow dots) and a multi-PR in Cat-II (gray dots) decreases when the number of multi-language files increases. Spearman rank correlation showed a strong relationship (p=-0,90) only for Cat-I (p=-0,57 for Cat-II). This may be because when a pull request has many files, the reviewer does not invest much time to review it and instead asks to slice the large pull request in more manageable chunks [114].

For mono-PRs, including Cat-III, there is no correlation between the size of a PR and its acceptance/rejection period. In Figure 8.8(b), we compare the relationship between mono-language files and the accepted (rejected) periods of mono-PR in all three project categories. Spearman rank correlation showed a significant correlation only between the median of mono-language changed files (X axis) and the period taken for reject mono-PR in Cat-III (p=0,95). As future work, we will conduct more in-depth investigations to understand the reason behind this finding.

Cat-III systems have significantly fewer contributors than Cat-I and Cat-II. As

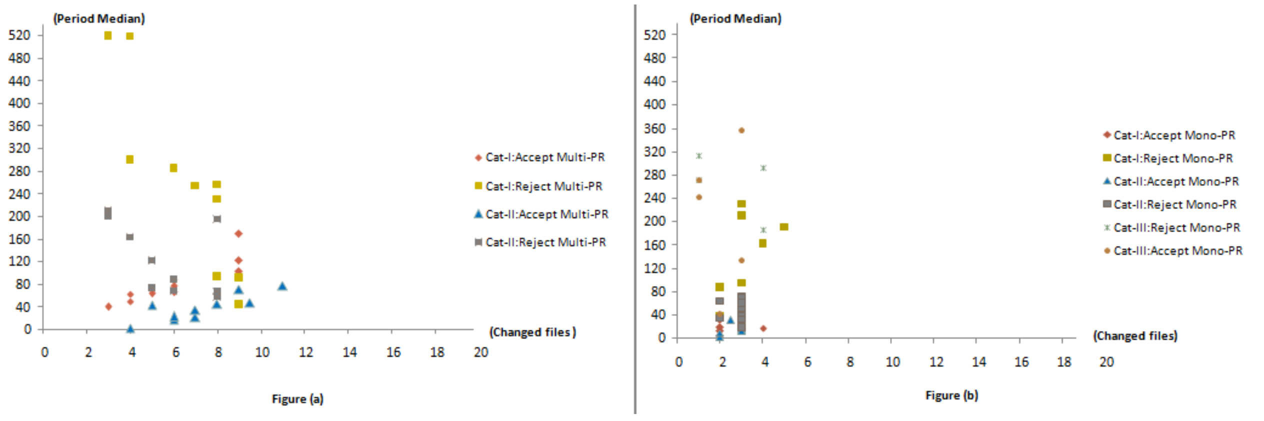


Figure 8.8 Comparing period to accept/reject a multi-/mono-PR according to the changed files.

an alternative explanation to the different PR acceptance/rejection periods between Cat-III and Cat-I/Cat-II (see Figure 8.7), we consider the size of a project's community *i.e.,* the number of contributors involved in each software system. For example, a project with 100 contributors would have a larger pool of reviewers than a project with only 50 contributors, and hence might be more effective in reviewing, regardless of multi- or mono-PRs.

Figure 8.9 presents the distribution of the number of contributors per studied system. It shows that Cat-III projects have the least number of contributors compared to contributors in Cat-I and Cat-II. Our Mann-Whitney U tests show significant differences between both Cat-III vs. Cat-I (p-value= 0.014, effect size = 0.26) and Cat-III vs. Cat-II (p-value = 0.025, effect size = 0,51) comparisons. We conclude that the differences between the ML frameworks (Cat-III and Cat-I) in terms of mono-PR periods are not specific to the complexity or other characteristics of ML code, but rather due to the size of the developer community. A low number of contributors could cause delays in the revision process because there are not enough contributors for all the pull requests and this is what may causes pull requests to remain under revision for a long time before being accepted or rejected.

In ML frameworks, multi-language PRs take longer to be accepted than mono-language PRs and ML frameworks take longer to accept/reject a multi-PR than traditional systems.

RQ8.4. Are multi-language pull requests more bug-prone than mono-language pull requests in machine learning frameworks?

Approach : Despite the diverse advantages of multi-language development, it presents some challenges to developers such as decreasing the quality and security of software systems [78].

Table 8.2 P-value of the Mann-Whitney U test.

	Compared categories	Compared analysis	P-value
Acceptance rate	Cat-I vs. Cat-I	Accept multi-PR vs. Accept mono-PR	**0,02175626**
	Cat-II vs. Cat-II	Accept multi-PR vs. Accept mono-PR	0,3526814
	Cat-I vs. Cat-II	Accept multi-PR vs. Accept multi-PR	0,8534283
	Cat-I vs. Cat-II	Accept mono-PR vs. Accept mono-PR	0,5288489
	Cat-I vs. Cat-III	Accept mono-PR vs. Accept mono-PR	**0,00678733**
	Cat-II vs. Cat-III	Accept mono-PR vs. Accept mono-PR	**0,02498972**
Acceptance period	Cat-I vs. Cat-I	Accept multi-PR vs. Reject multi-PR	**0,006841456**
	Cat-I vs. Cat-I	Accept multi-PR vs. Accept mono-PR	**0,000181651**
	Cat-I vs. Cat-I	Reject multi-PR vs. Reject mono-PR	**0,01552552**
	Cat-I vs. Cat-I	Accept mono-PR vs. Reject mono-PR	**0,000328133**
	Cat-II vs. Cat-II	Accept multi-PR vs. Reject multi-PR	**0,000487129**
	Cat-II vs. Cat-II	Accept multi-PR vs. Accept mono-PR	0,03749167
	Cat-II vs. Cat-II	Reject multi-PR vs. Reject mono-PR	**0,000725281**
	Cat-II vs. Cat-II	Accept mono-PR vs. Reject mono-PR	**0,000324753**
	Cat-I vs. Cat-II	Accept multi-PR vs. Accept multi-RP	**0,005196042**
	Cat-I vs. Cat-II	Reject multi-PR vs. Reject multi-PR	**0,02323064**
	Cat-I vs. Cat-II	Accept mono-PR vs. Accept mono-PR	0,8796494
	Cat-I vs. Cat-II	Reject mono-PR vs. Reject mono-PR	0,03546299
	Cat-II vs. Cat-III	Accept mono-PR vs. Accept mono-PR	**0,000102838**
	Cat-II vs. Cat-III	Reject mono-PR vs. Reject mono-PR	**0,000102838**
	Cat-I vs. Cat-III	Accept mono-PR vs. Accept mono-PR	**0,000754633**
	Cat-I vs. Cat-III	Reject mono-PR vs. Reject mono-PR	**0,000719868**
	Cat-III vs. Cat-III	Accept mono-PR vs. Accept mono-PR	0,4057174
Bug-inducing pull requests	Cat-I vs. Cat-I	Accept multi-PR vs. Accept mono-PR	0,4358722
	Cat-II vs. Cat-II	Accept multi-PR vs. Accept mono-PR	0,08920955
	Cat-I vs. Cat-II	Accept multi-PR vs. Accept multi-PR	0,3930481
	Cat-I vs. Cat-II	Accept mono-PR vs. Accept mono-PR	**0,02280556**
	Cat-I vs. Cat-III	Accept mono-PR vs. Accept mono-PR	0,03301111
	Cat-II vs. Cat-III	Accept mono-PR vs. Accept mono-PR	**0,000719868**

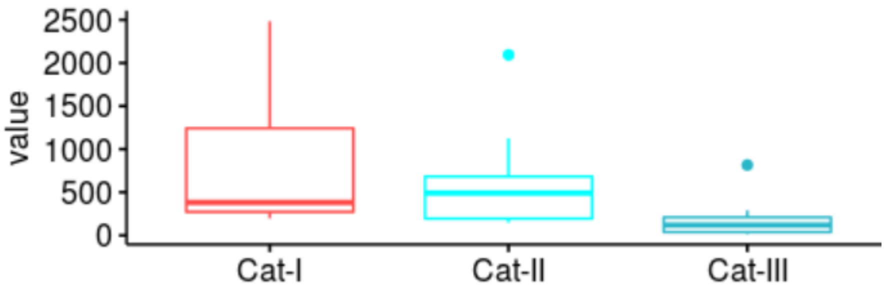


Figure 8.9 Number of contributors per software system.

In this research question, we aim to understand the correlation between multi-language development and the introduction of bugs in ML frameworks, compared to traditional systems. Our analysis focuses only on the accepted pull requests (rejected pull requests are incapable of introducing bugs since they are never merged into the code base). We compare our results across all three project categories.

Results : We only found a significant difference between the bug-proneness of mono-PRs of ML frameworks (Cat-I/III) and Cat-II. Figure 8.10 shows the percentage of bug-introducing multi-PRs and mono-PRs (relative to the total number of pull requests) in the studied software system categories. As shown in Figure 8.10a, the median percentage of the bug-introducing multi-PR (mono-PR) of Cat-I projects is generally higher than the percentage of bug-introducing multi-PR (mono-PR) of Cat-II projects. However, only a significant difference was found between mono-PR of Cat-I and mono-PR of Cat-II (p-value = 0,023) with a medium effect size of 0,48. No significant difference was found between both Cat-I's buggy multi-PR vs. mono-PR (p-value = 0,44) and Cat-II's buggy multi-PR vs. mono-PR (p-value = 0,089) comparisons. Furthermore, statistical tests showed a significant difference between the buggy mono-PRs in Cat-II and Cat-III (p-value = 0,000719868) with a large effect size of 0,82, but no significant difference was found between the buggy mono-PRs in Cat-I and Cat-III (p-value = 0,033).

Despite the longer acceptance period and lower acceptance rates of multi-PR, no difference was found between the bug-proneness of Cat-I and Cat-II multi-PR. However, mono-PR in ML frameworks seem to be more bug-prone.

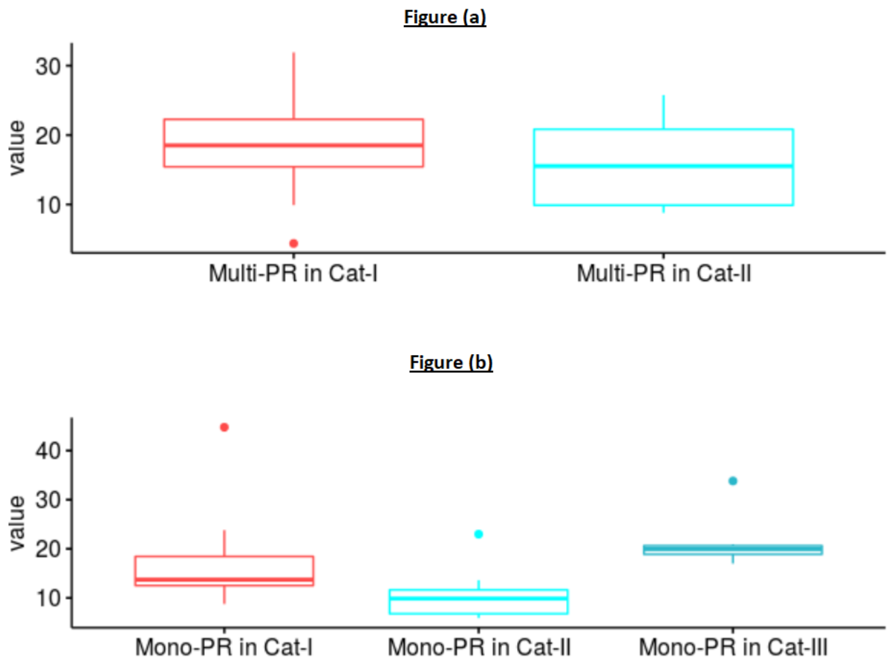


Figure 8.10 Percentage of Bug-inducing pull requests.

8.4 Discussion

We observed from the results that machine learning frameworks follow the same multi-language development trend as traditional projects. Out of the top 20 open source machine learning frameworks identified by Braiek *et al.* 's study [65], we analyzed their source code and defined the different languages used. We find that only 35% (seven frameworks) are mono-language frameworks, while 65% (13 frameworks, of which we studied the largest 10 frameworks for our study) are multi-language. In other words, machine learning developers are generally aware of the benefits of multi-language development, and are equally able to attract open-source contributions just like the traditional (*i.e.,* non-ML) open-source projects. The next RQs analyze to what extent those contributions are successful in getting accepted and of high quality (bug-free).

Multi-language development has been presented as a solution for diverse problems, but, at the same time, it represents a difficult practice that needs to be used carefully. ML is a relatively new domain and the development of ML systems requires competences of both software developers (experience in programming languages) and data scientists (experience in ML algorithms and the involved data). Pull request reviewers could be either data scientists or software developers, and either volunteers or employees, as as discussed by Braiek *et al.* [65].

Thus, the acceptance rate of a multi-PR could vary depending on the difference in expertise of the reviewers involved. In other words, the findings in this RQ highlight that the interaction between the complexity of the ML domain and of multi-language development can have an impact on the contribution review process. Future work should consider this issue in order to support ML framework developers and reviewers in dealing with multi-PRs.

The results show also a correlation between mono-PR and the introduction of bugs in ML frameworks. The lack of correlation between multi-PR and bug introduction could be explained by the fact that multi-PR in ML systems are less accepted than mono-PR as shown in (RQ8.2). Also, the longer time spent by reviewers before rejecting a multi-PR (as shown in RQ8.3) shows that reviewers pay more attention when reviewing multi-PR due to its complexity (*i.e.,* inter-language dependencies between the multi-language changed files) and the potential risks (bug introducing) it can cause. Since we analyze only the accepted PRs in this research question, we argue that a big percentage of risky multi-PR may already been cleaned in the review process.

8.5 Threats to validity

Threats to internal validity : Threats to the internal validity of our study concern the selected projects, the scripts used, and the pull requests analysis methodology. To mitigate these threats, we relied on the literature to identify projects shown to be among the largest projects in terms of lines of code and contributions. We developed diverse python scripts that we ran on GitHub API. We ensured the validity of the scripts' outcome by performing a manual validation on a sample.

Threats to external validity : Threats to external validity concern the factors that could affect the generalizability of our findings. Our findings may not be generalizable for all the existing multi-language systems (including machine learning and traditional systems) since we only studied a sample of 27 open-source projects. Software system' characteristics could vary depending on different criteria and factors. However, to mitigate this threat, we selected the largest ML frameworks [65] and Java/C systems [1] and we ensured that subjects of both categories are comparable regarding programming languages and pull requests.

Threats to conclusion validity : Threats to conclusion concern the relationship between the treatments and the findings. To mitigate this threat, we used Mann-Whitney U test, a non-parametric test, to compare the different analysis results across the three project categories. For the control of family-wise error rate, we used the Bonferroni correction to calculate an adjusted p-value whenever the same sample is tested more than once.

8.6 Chapter Summary

In this chapter, we studied the prevalence of the multi-language practice in machine learning (ML) frameworks. Since the challenges of multi-language development in traditional systems were a subject of interest of several existing research studies as we show in Chapter 4, we performed, throughout our study, a comparison of our analysis results between ML and traditional multi-language systems. Our major results show that (1) Multi-language PRs in ML frameworks have a lower acceptance rate than mono-language PRs ; (2) multi-language PRs in ML frameworks take longer to be accepted than mono-language PRs, and ML frameworks take longer to accept/reject a multi-PR than traditional systems ; and (3) mono-language PRs of ML frameworks are more bug-prone than traditional systems. Other characteristics were found to be similar between the studied ML and traditional projects. The study's findings provided a correlation between the existence of multi-language development with machine learning while multi-language development influences the software contributions (pull requests) to ML frameworks as discussed in RQ8.2 and RQ8.3.

CHAPTER 9 CONCLUSION

In this thesis, we conducted a set of studies to understand the prevalence and the process of integrating multi-language development within traditional systems. Thus, we studied the main elements that a developer needs to consider when developing multi-language systems *e.g.,* the languages to integrate, the techniques that allow the communication between the languages, the best practices, and the dependencies that ensure a safe maintenance between the different components. We also investigate the impact of multi-language development on software quality and security, and the risk that it could involve.

In recent years, multi-language development has been adopted massively in the domain of AI-based software systems. Thus, we also analyzed the prevalence of multi-language development in machine learning frameworks (ML) where we studied the impact of multi-language adoption in these frameworks and compared them with traditional systems (*i.e.,* non-ML systems). Our research hypothesis stated :

Thesis Hypothesis :
We hypothesize that (1) multi-language development is common, spanning many combinations of languages through specific techniques, and, over time, has led to a catalog of best practices ; (2) the integration of multiple languages with different development rules (semantic and lexical) makes maintenance activities, such as change impact analysis, more complex and bug/vulnerability-prone ; (3) the impact of multi-language development on software contributions and bug introduction is larger in machine learning frameworks than in traditional systems.

We will go more in detail on each sub-hypothesis, however, in general, based on our results, we were able to validate our research hypothesis.

In particular, we found that multi-language development is everywhere nowadays and is incarnated even in modern software systems that are using machine learning frameworks. We found also that multi-language developers face an important number of challenges when using multi-language development, in particular maintenance challenges that lead to complex dependencies between the integrated languages with different semantic and lexical rules. Our findings show that multi-language development exposes software systems to higher risks of being buggy, especially because of the inter-language dependencies *i.e.,* dependencies between the different languages used in the system. Furthermore, we found that multi-language development impacts open-source software contributions, in the case of ML frameworks, in terms of the acceptance rate of the pull requests, in the sense that multi-language pull requests are likely to introduce more bugs to the software than mono-language pull requests.

9.1 Sub-hypothesis One (Chapters 4 and 5)

Sub-hypothesis : Multi-language development is common, spanning many combinations of languages through specific techniques, and, over time, has led to a catalog of best practices.

We conducted a systematic literature review (SLR) on multi-language development usage from Engineering Village, starting with 3964 research papers and ending up with 138 papers after exclusions and inclusions. 138 papers published during 10 years seems a limited number compared to the prevalence and importance of multi-language development nowadays.

The findings showed that the number of published papers involving multi-language development is increasing over time with a large increase between 2017 and 2019. We found that 66,66% of the papers belong to the "Software analysis" category and 22,46% of the papers claimed that change impact analysis is the major concern of developers as reported by researchers. Furthermore, we identified 60 combinations of languages discussed in the literature where most papers focused on the analysis of the combination of Java and C/C++ (37.68%), through the Java Native Interface, following by Python/C(++) (9,42%), through Python-C-extensions.

We then conducted a qualitative study on the Java Native Interface, as it was found, through the systematic literature review, to be the most used technique for integrating languages. Thus, we semi-automatically studied the source code *i.e.,* the JNI part of 100 software systems using Java Native Interface collected from GitHub. The goal of this second study is to identify development practices that the JNI users follow. We considered a practice to be any methodology of the implementation of JNI functions, a recurrent or a common JNI piece of code, a recurrent security check, or a common use of specific attributes identified in ten or more of the studied JNI systems. To classify a practice as a good one, it should verify the following requirements : it respects the JNI specifications, the programming language rules, and the design patterns. We mined for practices in JDK v9, considered as our oracle, then checked their occurrences in the other JNI software systems. We believe that JDK could be an appropriate repository of good practices as it is a large code base that includes many usages of JNI and written by experts.

The qualitative study identified 11 JNI practices that seem to be followed the most by the developers of the studied software systems. They are mainly related to library loading, exceptions management, return types management, local and global references management, String uses, and the JNI different implementation ways.

Based on these findings, we validated our first sub-hypothesis by showing through our systematic literature review that multi-language development is common and prevalent nowadays. We also validated that diverse techniques are available to support developers in integrating languages together with different programming rules. Through the qualitative study, we validated that there are good practices of multi-language development (a case study of Java Native Interface) used sometimes by developers and ignored by others.

9.2 Sub-hypothesis Two (Chapters 6 and 7)

Sub-hypothesis : The integration of multiple languages with different development rules (semantic and lexical) makes maintenance activities, such as change impact analysis, more complex and bug/vulnerability-prone.

While it is highly recommended by the multi-language development practitioners to adopt solid practices for impact analysis of changes within multi-language systems, we found in the previous study that it still presents a challenge and a topic of debate in the literature. Change impact analysis in multi-language systems needs to account for interactions between components written in different programming languages. A lack of understanding of the impact of a change in a multi-language system can lead to a higher negative impact on system performance than in a mono-language system.

Hence, we technically surveyed 69 experienced developers and interviewed six of them for the validation of the survey's answers, to understand their current practices and needs regarding their daily process when maintaining a multi-language system. Our survey contained 30 questions grouped into four parts. We identified, firstly, the different challenges and issues of change impact analysis in multi-language systems ; secondly, the steps followed by the developers when changing such systems ; thirdly, we investigated the existing change impact analysis means used by developers in the industry ; and last, the requirements for a change impact analysis tool dedicated for multi-language systems.

We found that, in general, developers are performing change-impact analysis in multi-language systems implicitly *i.e.,* they are aware about the use of the right testing methods adequate for multi-language systems but without knowing that they are doing a part of the change impact process. They opt solely for testing methods and manual checks to analyze the impact of their changes in multi-language systems as they are worried that their companies do not provide enough specific tools to guide them during their changes within multi-language systems. Thus, the need to have a change-impact analysis tool that supports multi-language systems and analyse the dependencies is still very high. Furthermore, we found, from the developer's experiences, that multi-language systems are more susceptible to be negatively impacted by vulnerabilities through changes *i.e.,* code injections, memory dump, or incorrectly released memory, etc.

To answer developer's needs and minimize the negative impact of the maintenance process *i.e.,* software changes, within multi-language systems, we introduced two approaches for multi-language dependency analysis : S-MLDA (Static Multi-language Dependency Analyzer) and H-MLDA (Historical Multi-language Dependency Analyzer), which we apply on ten open-source multi-language systems to empirically analyze the prevalence of the dependencies across languages *i.e.,* inter-language dependencies and their impact on software quality and security. These two approaches could be combined to ensure higher precision and recall. We evaluated the accuracy of the approaches and we found precision (recall) values of 100% (68%) for S-MLDA, and 68% (87%) for H-MLDA.

Regarding the empirical study, we selected 10 open-source Java Native Interface systems to analyse as Java Native Interface was shown in our first study *i.e.,* the systematic literature review, as the most used technique for combining languages. We found that the more inter-language dependencies *i.e.,* dependencies between components written in different languages, in a system, the higher the risk of bugs and vulnerabilities being introduced, while this risk remains constant for intra-language dependencies *i.e.,* dependencies between components written in the same language. Furthermore, we found that the percentage of bugs within inter-language dependencies is three times higher than the percentage of bugs identified in intra-language dependencies while the percentage of vulnerabilities within inter-language dependencies is twice the percentage of vulnerabilities introduced in intra-language dependencies.

Our findings confirmed the second sub-hypothesis : our survey shows that the maintenance phase is the most challenging phase during the life cycle of a multi-language system (see figure 6.13). We validated that change impact analysis becomes more complex due to a complex multi-language dependency analysis between the components written in different languages. Furthermore, from the survey we confirmed that the mishandling of change impact analysis in multi-language development can lead to serious problems such as a decrease in quality, introduce of security vulnerabilities, increase in project costs, the introduction of incompatibilities, as discussed in Section 6.4.5. The empirical study on the multi-language dependencies validated also the findings of the survey and the sub-hypothesis by confirming that multi-language dependencies are more bug-prone than mono-language dependencies and risks to face the system to several vulnerabilities.

9.3 Sub-hypothesis Three (Chapter 8)

Sub-hypothesis : The impact of multi-language development on software contributions and bug introduction is larger in machine learning frameworks than in traditional systems.

The role of machine learning frameworks in software applications has exploded in recent years. Similar to non-machine learning frameworks, those frameworks need to evolve to incorporate new features. The use of multiple programming languages in their code base enables scientists to write optimized low-level code while developers can integrate the latter into a robust framework using a higher-level programming language. While, in the previous studies we showed that multi-language code bases have an impact on the development process in the traditional multi-language systems, here we empirically analyzed the prevalence of multi-language development in the ten largest open-source multi-language machine learning frameworks and we compared their findings with those of ten large open-source multi-language traditional systems in terms of software contributions *i.e.,* the volume of pull requests, their acceptance rate, review process duration, and bug-proneness.

We found that multi-language development is indeed used in machine learning frameworks. We showed that multi-language pull requests are significantly harder to get accepted in multi-language machine learning frameworks than in multi-language traditional systems. Also, we showed that multi-language pull requests in multi-language machine learning take longer to be accepted than mono-language pull requests while machine learning frameworks take longer to accept/reject a multi-language pull requests than traditional systems. Furthermore, in both machine learning and traditional systems, multi-language pull requests are less likely to be accepted than mono-language pull requests ; it also takes longer for both multi- and mono-language pull requests to be rejected than accepted. Finally, we find that pull requests in machine learning frameworks are more bug-prone than those of traditional systems.

The above findings validate our third sub-hypothesis. We validated that software contributions (multi-language pull requests) are significantly harder to get accepted in ML frameworks than in traditional systems. Multi-language development in ML frameworks also slows sown the decision to accept or reject a multi-language software contribution, which makes the multi-language adoption in ML frameworks hard and challenging.

9.4 Limitations

This thesis provides results regarding the impact of multi-language development in software systems. However, during this dissertation we studied only two types of multi-language systems : traditional systems where we focused on Java native interface systems and machine learning frameworks where we focused on the Python-C-extension.

Other well-known types of multi-language systems are distributed J2EE systems. Such systems rely on the J2EE platform and containers that offer infrastructure and architectural services to ensure correct functionality. They present similar challenges as Java native interface systems, *i.e.,* hidden dependencies, higher risk of bug introduction, difficult static code analysis, etc [8] as they combine several languages : (1) Java, for both server and client code (embedded in HTML, JSP, or JSF tags), (2) JavaScript, on the client side, embedded in HTML, JSP, or JSF tags, (3) various property files (key/value pairs), and (4) various XML configuration files (web.xml, ra.xml, ejbjar.xml) [63]. J2EE systems heavily rely on dynamic tracing as they rely on dynamic binding-techniques, using a combination of : (1) reflection or introspection, (2) data-driven control, where data-values control calls between clients and servers, in a message-driven style, (3) run-time input where control-data values control calls between clients and servers, and (4) runtime code-generation where executable code is generated at load-time or runtime from configuration files [63]. That combination of techniques requires more advanced analysis, both static and dynamic.

One could also argue that any Java program in fact is multi-language, since each Java program requires functionality of the underlying Java VM to be executed at run-time, where this VM typically is implemented in C or C++. While it is true that the VM code is executed at run-time, it is not a part of the code base of the Java project (which is why we do not consider this to be an instance of multi-language development). The difference with our multi-language development definition is that the interaction of a Java system with the VM is performed through the stable byte-code specification that is implemented and maintained by the Java language implementers, *i.e.,* not application developers. On the other hand, the code that uses foreign function interfaces like JNI and Python-C-extension has to be implemented by regular application developers, which puts more maintenance burden on them. This is why our definition of multi-language development explicitly focuses on multi-language code that is part of a project's code base.

9.5 Future work

Our research contributions open a wide range of opportunities for future work. This section discusses some of them.

First, we plan to develop a tool in order to cover the requirements asked by the surveyed developers in Chapter 6, such as : proposing adequate testing methods, proposing potential changes automatically, etc. It could also be interesting to integrate the use of machine learning in such tool as it could help to learn from bugs and the changes already made, thus, proposing new changes safer to the software.

Most of our case studies through this dissertation focus on the combination of Java and C/C++, through the Java Native Interface. This choice was based on the findings of the systematic literature review presented in Chapter 4. However, today we observe a large volume of modern software systems being developed *e.g.,* machine learning frameworks, where the combination of python and C/C++, through the Python C extension, takes the lead. It is known that Python is the most commonly used programming language in machine learning [65], thus, we believe that replicating our studies presented in Chapters 4 and 5 on these frameworks can reveal interesting development practices and particular challenges as each combination of languages follows its own developing rules.

Thus, we plan to replicate the state of the practice study on the top 10 largest open-source machine learning frameworks used in the study presented in Chapter 8 to identify a new set of practices and to provide a guideline to the machine learning users *i.e.,* developers and data-scientists to improve the software quality. On the other hand, the S-MLDA and H-MLDA approaches presented in Chapter 7 could be extended to systems written in Python and C by implementing the programming rules of Python-C-extension technique for S-MLDA and considering co-changes involving Python and C files for H-MLDA.

Furthermore, it could be interesting to evaluate the best practices for multi-language development by performing a user study where two groups of developers are asked to perform two different coding tasks on machine learning frameworks. The user study will consist of a comprehension task on a version of a system following practices and a version without, to check accuracy, completion time, etc. Thus, we will record coding attributes to measure the relevance of such identified practices *i.e.,* we will measure coding time, the percentage of errors, access to developers' blogs to look for information, code smells *e.g.,* if the lack of use of such practices involves blob class, etc. Based on that, we will measure the accuracy of our guideline. A prototype recommending developers the relevant best practices for their multi-language development could be developed, and perhaps released as an open-source IDE plugin. Such a plugin will recommend refactoring opportunities to developers based on the identified JNI best practices in Chapter 4.

Through the survey presented in Chapter 6, we found that developers are mainly using manual validation and testing for their multi-language dependency analysis. As S-MLDA and H-MLDA are two approaches that identify these dependencies, it will be interesting to evaluate them through a user study in order to confirm their efficiency and how they could perform better than manual analysis.

In order to better cater for machine learning frameworks users, we will investigate the existing bugs and problems in machine learning frameworks developed with multiple programming languages identified by researchers in the literature and by developers in developing blogs *i.e.,* Stack Overflow. We want to understand the kind of bugs encountered by developers when implementing a multi-language machine learning framework. We plan to propose recommendations and suggestions on how they should manage this kind of development.

In addition, we plan to propose debugging processes specific for multi-language development. Despite the advances of automated testing techniques for complex multi-language systems, locating the causes of multi-language bugs is difficult and consumes significant human effort [115]. Debugging these systems is very important ; establishing such debugging processes will help users to easily identify the location of a suspected multi-language bug and suggest an automatic program repair and program fix that makes the test case succeed.

Related Publications

The following is a list of our publications related to this dissertation.

1. **Manel Grichi**, Mouna Abidi, Yann-Gaël Guéhéneuc, Foutse Khomh
   STATE OF PRACTICES OF JAVA NATIVE INTERFACE, Published in the 29th Annual International Conference on Computer Science and Software Engineering (CASCON19).
   This paper is presented in Chapter 5.
2. **Manel Grichi**, Mouna Abidi, Fehmi Jaafar
   CHANGE IMPACT ANALYSIS IN MULTI-LANGUAGE SYSTEMS : AN INDUSTRIAL PERSPECTIVE, Submitted to the Software : Practice and Experience Journal (Special issue of JSPE).
   This paper is presented in Chapter 6.
3. **Manel Grichi**, Mouna Abidi, Fehmi Jaafar, Ellis E.Eghan, Bram Adams
   ON THE IMPACT OF INTER-LANGUAGE DEPENDENCIES IN MULTI-LANGUAGE SYSTEMS : EMPIRICAL CASE STUDY ON JAVA NATIVE INTERFACE APPLICATION(JNI), Accepted in the 20th IEEE International Conference on Software quality, Reliability, and security (QRS20), and one out of six accepted papers included in the IEEE Transactions on Reliability.
   This paper is presented in Chapter 7.
4. **Manel Grichi**, Ellis E.Eghan, Bram Adams
   ON THE IMPACT OF MULTI-LANGUAGE DEVELOPMENT IN MACHINE LEARNING FRAMEWORKS, Accepted in the 36th International Conference on Software Maintenance and Evolution (ICSME20).
   This paper is presented in Chapter 8.

The following publications are not directly related to the material in this dissertation, but were produced in parallel to the research contained for this dissertation.

1. Mouna Abidi, **Manel Grichi**, Foutse Khomh
   BEHIND THE SCENES : DEVELOPERS' PERCEPTION OF MULTI-LANGUAGE PRACTICES, Published in the 29th Annual International Conference on Computer Science and Software Engineering (CASCON19). Best Student Paper Award.
2. Mouna Abidi, **Manel Grichi**, Foutse Khomh, Yann-Gaël Guéhéneuc

CODE SMELLS FOR MULTI-LANGUAGE SYSTEMS, Published in the 24th European Conference on Pattern Languages of Programs (EuroPLoP19)

3. Ellis E.Eghan, **Manel Grichi**, William Glazer Cavanagh, Zhen Ming Jack Jiang, Bram Adams
   MACHINE LEARNING MODELS, FROM RESEARCH PAPERs TO APPLICATIONs, Submitted to the ACM Transactions on Software Engineering and Methodology Journal (TOSEM).
4. **Manel Grichi**, Fehmi Jaafar, Jean Decian, Yasir Malik, Caesar Jude Clemente
   DEMYSTIFYING THE SECURITY BUGS IN SOFTWARE SYSTEMS : AN EMPIRICAL STUDY, Submitted to the IEEE Software journal.

REFERENCES

[1] M. Grichi, M. Abidi, Y.-G. Guéhéneuc, and F. Khomh, "State of practices of java native interface," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '19. IBM Corp., 2019.

[2] *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586.* Berlin, Heidelberg : Springer-Verlag, 2014.

[3] F. Boughanmi, "Change impact analysis of multi-language and heterogeneously-licensed software," *publications.polymtl.ca*, 2010.

[4] B. D. Burow, "Mixed language programming," in *Computing in High Energy Physics' 95 : CHEP'95.* World Scientific, 1996, pp. 610–614.

[5] "Application of mixed language programming," *Computer Physics Communications*, vol. 61, no. 1, pp. 150 – 162, 1990.

[6] P. Mayer, M. Kirsch, and M. Anh, "On multi-language software development, cross-language links and accompanying tools : a survey of professional software developers," *Journal of Software Engineering Research and Development*, vol. 5, pp. 1 :1–1 :33, 2017.

[7] P. Mayer, M. Kirsch, and M. A. Le, "On multi-language software development, cross-language links and accompanying tools : a survey of professional software developers," *Journal of Software Engineering Research and Development*, vol. 5, no. 1, p. 1, 2017.

[8] A. Shatnawi, H. Mili, M. Abdellatif, Y.-G. Guéhéneuc, N. Moha, G. Hecht, G. E. Boussaidi, and J. Privat, "Static code analysis of multilanguage software systems," *arXiv preprint arXiv :1906.00815*, 2019.

[9] F. Boughanmi, "Multi-language and heterogeneously-licensed software analysis," in *17th Working Conference on Reverse Engineering*, 2010.

[10] M. Abidi, M. Grichi, and F. Khomh, "Behind the scenes : Developers' perception of multi-language practices," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '19. USA : IBM Corp., 2019, p. 72–81.

[11] K. G. Cheetancheri and H. H. Cheng, "Mixed language programming in c/c++ and java for applications in mechatronic systems," in *2006 2nd IEEE/ASME International Conference on Mechatronics and Embedded Systems and Applications.* IEEE, 2006, pp. 1–6.

[12] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, “A large scale study of programming languages and code quality in github,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA : Association for Computing Machinery, 2014, p. 155–165.

[13] P. Mayer and A. Bauer, “An empirical analysis of the utilization of multiple programming languages in open source projects,” in *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, 2015, pp. 1–10.

[14] S. Hassaine, F. Boughanmi, Y. G. Guéhéneuc, S. Hamel, and G. Antoniol, “A seismology-inspired approach to study change propagation,” in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, Sept 2011, pp. 53–62.

[15] B. Lee, M. Hirzel, R. Grimm, and K. S. McKinley, “Debugging mixed-environment programs with blink,” *Softw. Pract. Exper.*, vol. 45, no. 9, p. 1277–1306, Sep. 2015.

[16] D. Zhang and J. J. Tsai, “Machine learning and software engineering,” *Software Quality Journal*, vol. 11, no. 2, pp. 87–119, 2003.

[17] B. Kitchenham and S. Charters, “Guidelines for performing systematic literature reviews in software engineering,” 2007.

[18] R. Bluff, “9 rosalind bluff grounded theory : the methodology,” in *9 ROSALIND BLUFF Grounded theory : the methodology*, 2005.

[19] Z. Mushtaq and G. Rasool, “Multilingual source code analysis : State of the art and challenges,” in *2015 International Conference on Open Source Systems Technologies (ICOSST)*, Dec 2015, pp. 170–175.

[20] D. Binkley, “Source code analysis : A road map,” in *Future of Software Engineering, 2007. FOSE ’07*, May 2007, pp. 104–119.

[21] E. Flores, A. Barrón-Cedeño, P. Rosso, and L. Moreno, “Towards the detection of cross-language source code reuse,” in *Proceedings of the 16th International Conference on Natural Language Processing and Information Systems*, ser. NLDB’11. Springer-Verlag, 2011, pp. 250–253.

[22] M. Harman, “Why source code analysis and manipulation will always be important,” in *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, Sept 2010, pp. 7–19.

[23] H. M. Kienle, J. Kraft, and H. A. Müller, “Software reverse engineering in the domain of complex embedded systems,” in *Reverse Engineering-Recent Advances and Applications.* InTech, 2012.

[24] N. Synytskyy, J. R. Cordy, and T. R. Dean, “Robust multilingual parsing using island grammars,” in *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON ’03. IBM Press, 2003, pp. 266–278.

[25] M. Furr and J. S. Foster, “Checking type safety of foreign function calls,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 2005.

[26] “Java 8 jni specification - design overview,” http://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/design.html, 2017, (Accessed on 07/10/2017).

[27] S. Liang, *Java Native Interface : Programmer’s Guide and Reference.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[28] S. Li and G. Tan, “Finding reference-counting errors in python/c programs with affine analysis,” in *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586.* New York, NY, USA : Springer-Verlag New York, Inc., 2014, pp. 80–104.

[29] B. Kullbach, A. Winter, P. Dahm, and J. Ebert, “Program comprehension in multi-language systems,” in *Proceedings Fifth Working Conference on Reverse Engineering (Cat. No.98TB100261)*, Oct 1998, pp. 135–143.

[30] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillere, “Popularity, interoperability, and impact of programming languages in 100,000 open source projects,” in *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual.* IEEE, 2013, pp. 303–312.

[31] D. L. Moise and K. Wong, “Extracting and representing cross-language dependencies in diverse software systems,” in *12th Working Conference on Reverse Engineering (WCRE’05)*, Nov 2005, pp. 10 pp.–.

[32] L. Gong *et al.*, “Java security architecture (jdk 1.2),” *Draft Document, revision 0.8, Sun Microsystems, March*, 1998.

[33] G. Tan and J. Croft, “An empirical security study of the native code in the jdk,” in *Proceedings of the 17th Conference on Security Symposium*, ser. SS’08. Berkeley, CA, USA : USENIX Association, 2008, pp. 365–377.

[34] G. Kondoh and T. Onodera, “Finding bugs in java native interface programs,” in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA ’08. New York, NY, USA : ACM, 2008, pp. 109–118.

[35] S. Li and G. Tan, “Finding bugs in exceptional situations of jni programs,” in *Proceedings of the 16th ACM Conference on Computer and Communications Security.* ACM, 2009, pp. 442–452.

[36] S. Jiang, C. Mcmillan, and R. Santelices, "Do programmers do change impact analysis in debugging ?" *Empirical Sof. Eng.*, 2017.

[37] H. Abdeen, K. Bali, H. Sahraoui, and B. Dufour, "Learning dependency-based change impact predictors using independent change histories," *Information and Software Technology*, vol. 67, pp. 220 – 235, 2015. [Online]. Available : http://www.sciencedirect.com/science/article/pii/S0950584915001305

[38] P. K. Linos, "Polycare : a tool for re-engineering multi-language program integrations," in *Proceedings of First IEEE International Conference on Engineering of Complex Computer Systems. ICECCS'95*, 1995, pp. 338–341.

[39] J. W. Wilkerson, "A software change impact analysis taxonomy," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Sept 2012, pp. 625–628.

[40] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti : A tool for change impact analysis of java programs," in *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '04. New York, NY, USA : ACM, 2004.

[41] A. Srivastava, "Unreachable procedures in object-oriented programming," *ACM Lett. Program. Lang. Syst.*, pp. 355–364, Dec. 1992.

[42] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proceedings of the 9th European Conference on Object-Oriented Programming*, ser. ECOOP '95. London, UK, UK : Springer-Verlag, 1995, pp. 77–101.

[43] D. F. Becon, *Fast and effective optimization of statically typed object-oriented languages.* University of California, Berkeley, 1997.

[44] O. Shivers, "Control-flow analysis of higher-order languages," Ph.D. dissertation, 1991.

[45] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Software Testing, Verification and Reliability*, vol. 23, no. 8, pp. 613–646, 2013.

[46] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich, "Jripples : a tool for program comprehension during incremental change," in *13th International Workshop on Program Comprehension (IWPC'05)*, May 2005, pp. 149–152.

[47] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proceedings of the 26th International Conference on Software Engineering.* IEEE Computer Society, 2004.

[48] S. Gwizdala, Y. Jiang, and V. Rajlich, "Jtracker - a tool for change propagation in java," in *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, ser. CSMR '03. Washington, DC, USA : IEEE Computer Society, 2003, pp. 223–.

[49] S. Zhang, Z. Gu, Y. Lin, and J. Zhao, "Celadon : A change impact analysis tool for aspect-oriented programs," in *Companion of the 30th International Conference on Software Engineering*, ser. ICSE Companion '08. New York, NY, USA : ACM, 2008, pp. 913–914.

[50] O. C. Chesley, X. Ren, and B. G. Ryder, "Crisp : a debugging tool for java programs," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, Sept 2005, pp. 401–410.

[51] S. Zhang, Z. Gu, Y. Lin, and J. Zhao, "Autoflow : An automatic debugging tool for aspectj software," in *2008 IEEE International Conference on Software Maintenance*, Sept 2008, pp. 470–471.

[52] S. Hassaine, F. Boughanmi, Y. Gueheneuc, S. Hamel, and G. Antoniol, "Change impact analysis : An earthquake metaphor," in *2011 IEEE 19th International Conference on Program Comprehension*, 2011, pp. 209–210.

[53] M. C. O. Maia, R. A. Bittencourt, J. C. A. d. Figueiredo, and D. D. S. Guerrero, "The hybrid technique for object-oriented software change impact analysis," in *2010 14th European Conference on Software Maintenance and Reengineering*, March 2010, pp. 252–255.

[54] M. Bano, S. Imtiaz, N. Ikram, M. Niazi, and M. Usman, "Causes of requirement change - a systematic literature review," in *16th International Conference on Evaluation Assessment in Software Engineering*, 2012.

[55] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Transactions on Software Engineering*, Sep. 2004.

[56] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, and G. Antoniol, "Detecting asynchrony and dephase change patterns by mining software repositories," *Software : Evolution and Process*, 2014.

[57] F. Angerer, "Variability-aware change impact analysis of multi-language product lines," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA : ACM, 2014, pp. 903–906.

[58] L. Deruelle, M. Bouneffa, N. Melab, and H. Basson, “A change propagation model and platform for multi-database applications,” in *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, 2001, pp. 42–51.

[59] H. V. Nguyen, C. Kästner, and T. N. Nguyen, “Cross-language program slicing for dynamic web applications,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* New York, NY, USA : ACM, 2015, pp. 369–380.

[60] B. Cossette and R. J. Walker, “Polylingual dependency analysis using island grammars : A cost versus accuracy evaluation,” in *2007 IEEE International Conference on Software Maintenance*, Oct 2007, pp. 214–223.

[61] L. Moonen, “Generating robust parsers using island grammars,” in *Proceedings Eighth Working Conference on Reverse Engineering*, 2001, pp. 13–22.

[62] A. Shatnawi, H. Mili, G. E. Boussaidi, A. Boubaker, Y.-G. Guéhéneuc, N. Moha, J. Privat, and M. Abdellatif, “Analyzing program dependencies in java ee applications,” in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR ’17. IEEE Press, 2017, p. 64–74. [Online]. Available : https://doi.org/10.1109/MSR.2017.6

[63] G. Hecht, H. Mili, G. El-Boussaidi, A. Boubaker, M. Abdellatif, Y. Guéhéneuc, A. Shatnawi, J. Privat, and N. Moha, “Codifying hidden dependencies in legacy J2EE applications,” in *25th Asia-Pacific Software Engineering Conference, APSEC 2018, Nara, Japan, December 4-7, 2018.* IEEE, 2018, pp. 305–314.

[64] M. Sayagh and B. Adams, “Multi-layer software configuration : Empirical study on wordpress,” in *15th International Working Conference on Source Code Analysis and Manipulation*, 2015.

[65] H. Ben Braiek, F. Khomh, and B. Adams, “The open-closed principle of modern machine learning frameworks,” in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, May 2018, pp. 353–363.

[66] G. Varisteas, T. Avanesov, and R. State, “Distributed c++-python embedding for fast predictions and fast prototyping,” in *Proceedings of the Second Workshop on Distributed Infrastructures for Deep Learning*, 2018, pp. 9–14.

[67] A. M. Phelps and D. M. Parks, “Fun and games : Multi-language development,” *Queue*, vol. 1, no. 10, pp. 46–56, 2004.

[68] S. Buro and I. Mastroeni, “On the multi-language construction,” in *European Symposium on Programming.* Springer, 2019, pp. 293–321.

[69] A. Poggi and G. Adorni, “A multi language environment to develop multi agent applications,” in *International Workshop on Agent Theories, Architectures, and Languages.* Springer, 1996, pp. 325–339.

[70] S. Tasharrofi and E. Ternovska, “A semantic account for modularity in multi-language modelling of search problems,” in *International Symposium on Frontiers of Combining Systems.* Springer, 2011, pp. 259–274.

[71] F. Khomh, B. Adams, J. Cheng, M. Fokaefs, and G. Antoniol, “Software engineering for machine-learning applications : The road ahead,” *IEEE Software*, vol. 35, no. 5, pp. 81–84, 2018.

[72] A. B. Dhasade, A. S. M. Venigalla, and S. Chimalakonda, “Towards prioritizing github issues,” in *Proceedings of the 13th Innovations in Software Engineering Conference on Formerly known as India Software Engineering Conference*, 2020, pp. 1–5.

[73] E. Van Der Veen, G. Gousios, and A. Zaidman, “Automatically prioritizing pull requests,” in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories.* IEEE, 2015, pp. 357–361.

[74] G. Zhao, D. A. da Costa, and Y. Zou, “Improving the pull requests review process using learning-to-rank algorithms,” *Empirical Software Engineering*, vol. 24, no. 4, pp. 2140–2170, 2019.

[75] F. Boughanmi, “Multi-language and heterogeneously-licensed software analysis,” in *2010 17th Working Conference on Reverse Engineering*, Oct 2010, pp. 293–296.

[76] T. Arbuckle, “Measuring multi-language software evolution : A case study,” in *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, ser. IWPSE-EVOL ’11. New York, NY, USA : ACM, 2011, pp. 91–95.

[77] D. Zhang and J. J. P. Tsai, *Machine Learning Applications In Software Engineering (Series on Software Engineering and Knowledge Engineering).* USA : World Scientific Publishing Co., Inc., 2005.

[78] M. Abidi, M. Grichi, F. Khomh, and Y. Guéhéneuc, “Code smells for multi-language systems,” in *Proceedings of the 24th European Conference on Pattern Languages of Programs, EuroPLoP 2019, Irsee, Germany, July 3-7, 2019*, T. B. Sousa, Ed. ACM, 2019, pp. 12 :1–12 :13.

[79] F. Tomassetti and M. Torchiano, “An empirical assessment of polyglot-ism in github,” in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE ’14, 2014.

[80] E. Syriani, L. Luhunu, and H. Sahraoui, “Systematic mapping study of template-based code generation,” *Computer Languages, Systems Structures*, vol. 52, pp. 43 – 62, 2018. [Online]. Available : http://www.sciencedirect.com/science/article/pii/S1477842417301239

[81] M. Staples and M. Niazi, “Experiences using systematic review guidelines,” *J. Syst. Softw.*, vol. 80, no. 9, pp. 1425–1437, Sep. 2007.

[82] Z. Sharafi, Z. Soh, and Y.-G. Guéhéneuc, “A systematic literature review on the usage of eye-tracking in software engineering,” *Inf. Softw. Technol.*, pp. 79–107, Nov. 2015.

[83] S. Liang, “Java native interface : Programmer’s guide and specification,” 1999.

[84] M. Abidi, M. Grichi, and F. Khomh, “Behind the scenes : Developers’ perception of multi-language practices,” ser. CASCON ’19, 2019, p. 72–81.

[85] M. Abidi, F. Khomh, and Y. Guéhéneuc, “Anti-patterns for multi-language systems.” ACM, 2019, pp. 42 :1–42 :14.

[86] R. S. Arnold, *Software Change Impact Analysis.* Los Alamitos, CA, USA : IEEE Computer Society Press, 1996.

[87] S. S. Yau, R. A. Nicholl, J.-P. Tsai, and S.-S. Liu, “An integrated life-cycle model for software maintenance,” *Mathematical and Computer Modelling*, vol. 12, no. 9, p. 1177, 1989.

[88] B. Dit, M. Wagner, S. Wen, W. Wang, M. Linares-Vásquez, D. Poshyvanyk, and H. Kagdi, “Impactminer : A tool for change impact analysis,” in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014, New York, NY, USA, 2014, pp. 540–543.

[89] R.-H. Pfeiffer and A. Wasowski, “Texmo : A multi-language development environment,” in *Proceedings of the 8th European Conference on Modelling Foundations and Applications*, ser. ECMFA’12. Berlin, Heidelberg : Springer-Verlag, 2012, pp. 178–193.

[90] A. Fink, *The survey handbook.* Sage, 2003, vol. 1.

[91] H. Sharifipour, M. Shakeri, and H. Haghighi, “Structural test data generation using a memetic ant colony optimization based on evolution strategies,” *Swarm and Evolutionary Computation*, vol. 40, pp. 76 – 91, 2018.

[92] A. Thakur and G. Sharma, “Neural network based test case prioritization in software engineering,” in *International Conference on Advanced Informatics for Computing Research*, vol. 956, Shimla, India, 2019, pp. 334 – 345.

[93] M. Gligoric, S. Negara, O. Legunsen, and D. Marinov, “An empirical evaluation and comparison of manual and automated test selection,” in *Proceedings of the 29th*

ACM/IEEE international conference on Automated software engineering, Vasteras, Sweden, 2014, pp. 361 – 371.

[94] M. M. Lehman and L. A. Belady, *Program evolution : processes of software change.* Academic Press Professional, Inc., 1985.

[95] S. Hassaine, F. Boughanmi, Y.-G. Guéhéneuc, S. Hamel, and G. Antoniol, "A seismology-inspired approach to study change propagation," in *27th IEEE International Conference on Software Maintenance*, 2011.

[96] Y. Guéhéneuc and G. Antoniol, "Demima : A multilayered approach for design pattern identification," *IEEE Transactions on Software Engineering*, 2008.

[97] S. Mcintosh, B. Adams, M. Nagappan, and A. E. Hassan, "Mining co-change information to understand when build changes are necessary," in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sep. 2014, pp. 241–250.

[98] N. Ali, F. Jaafar, and A. E. Hassan, "Leveraging historical co-change information for requirements traceability," in *20th Working Conference on Reverse Engineering (WCRE)*, 2013.

[99] A. E. Hassan and R. C. Holt, "Predicting change propagation in software systems," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, 2004, pp. 284–293.

[100] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes ?" *SIGSOFT Softw. Eng. Notes*, 2005.

[101] F. Jaafar, Y. Gueheneuc, S. Hamel, and G. Antoniol, "An exploratory study of macro co-changes," in *2011 18th Working Conference on Reverse Engineering*, Oct 2011, pp. 325–334.

[102] D. Baca, K. Petersen, B. Carlsson, and L. Lundberg, "Static code analysis to detect software security vulnerabilities," in *Conference on Availability, Reliability and Security*, 2009.

[103] R. L. Scheaffer, W. Mendenhall III, R. L. Ott, and K. G. Gerow, *Elementary survey sampling.* Cengage Learning, 2011.

[104] M. Hollander, D. A. Wolfe, and E. Chicken, *Nonparametric statistical methods.* John Wiley & Sons, 2013, vol. 751.

[105] G. Tan, S. Chakradhar, R. Srivaths, and R. D. Wang, "Safe Java Native Interface," in *In Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering*, 2006, pp. 97–106.

[106] G. Tan and J. Croft, “An empirical security study of the native code in the jdk,” in *Proceedings of the 17th Conference on Security Symposium.* USA : USENIX Association, 2008.

[107] G. Tan, A. W. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. Wang, “Safe java native interface,” in *Proceedings of IEEE International Symposium on Secure Software Engineering*, vol. 97, 2006, p. 106.

[108] K. Patel, J. Fogarty, J. A. Landay, and B. L. Harrison, “Examining difficulties software developers encounter in the adoption of statistical machine learning.” in *AAAI*, 2008, pp. 1563–1566.

[109] M. Alshangiti, H. Sapkota, P. K. Murukannaiah, X. Liu, and Q. Yu, “Why is developing machine learning applications challenging ? a study on stack overflow posts,” in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).* IEEE, 2019, pp. 1–11.

[110] S. Kim, E. J. Whitehead,, and Y. Zhang, “Classifying software changes : Clean or buggy ?” *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[111] *Analysis of Clinical Trials Using Sas® : A Practical Guide*, 1st ed. SAS Publishing, 2005.

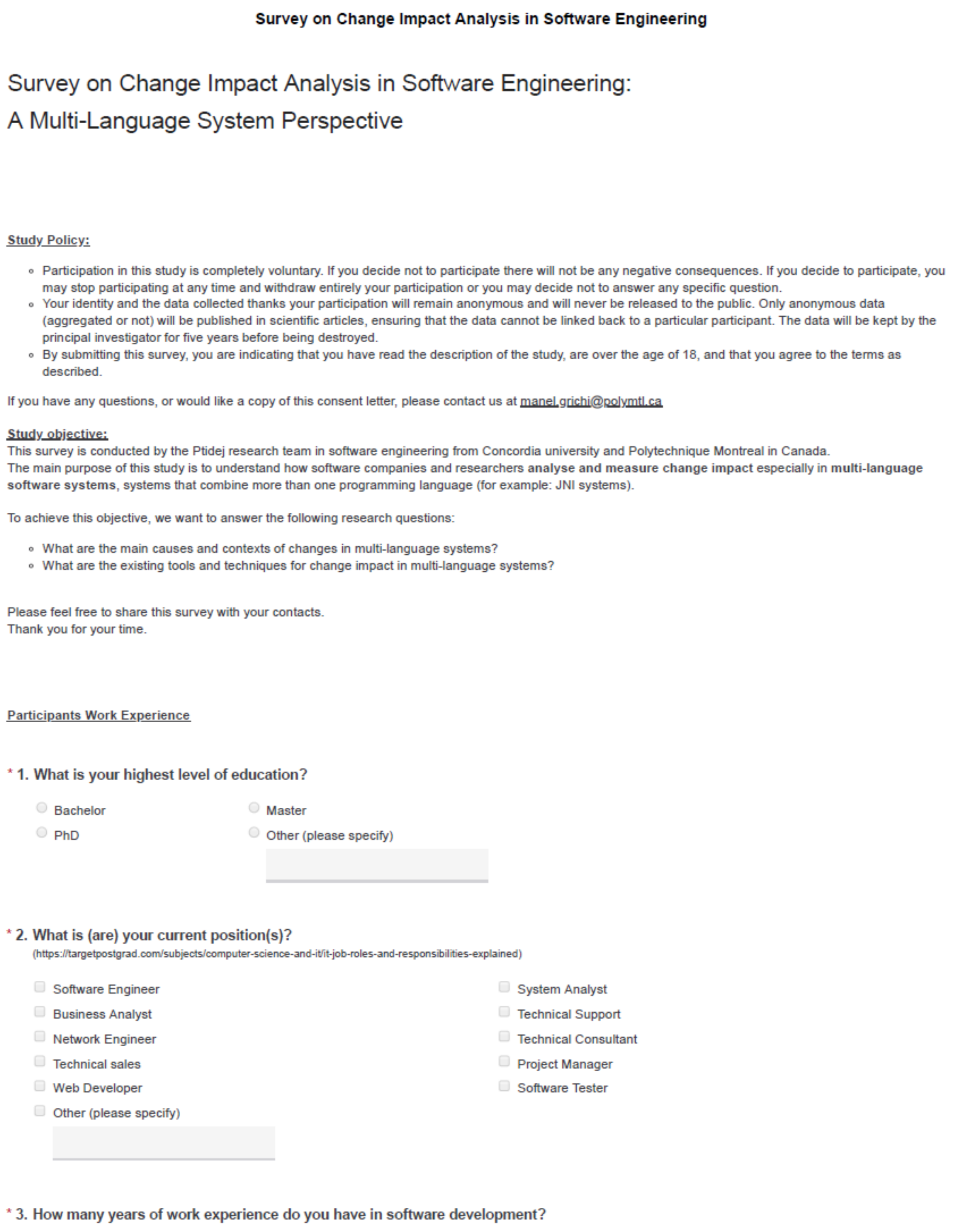
[112] N. Cliff, “Dominance statistics : Ordinal analyses to answer ordinal questions.” 1993.

[113] M. M. Rahman and C. K. Roy, “An insight into the pull requests of github,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA : Association for Computing Machinery, 2014, p. 364–367.

[114] Y. Jiang, B. Adams, and D. M. German, “Will my patch make it ? and how fast ? – case study on the linux kernel,” in *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories (MSR)*, San Francisco, CA, US, May 2013, pp. 101–110.

[115] S. Hong, T. Kwak, B. Lee, Y. Jeon, B. Ko, Y. Kim, and M. Kim, “Museum : Debugging real-world multilingual programs using mutation analysis,” *Information and Software Technology*, vol. 82, pp. 80 – 95, 2017.

ANNEXE A SURVEY QUESTIONNAIRE



Survey on Change Impact Analysis in Software Engineering

Survey on Change Impact Analysis in Software Engineering: A Multi-Language System Perspective

Study Policy:

- Participation in this study is completely voluntary. If you decide not to participate there will not be any negative consequences. If you decide to participate, you may stop participating at any time and withdraw entirely your participation or you may decide not to answer any specific question.
- Your identity and the data collected thanks your participation will remain anonymous and will never be released to the public. Only anonymous data (aggregated or not) will be published in scientific articles, ensuring that the data cannot be linked back to a particular participant. The data will be kept by the principal investigator for five years before being destroyed.
- By submitting this survey, you are indicating that you have read the description of the study, are over the age of 18, and that you agree to the terms as described.

If you have any questions, or would like a copy of this consent letter, please contact us at manel.grichi@polymtl.ca

Study objective:
This survey is conducted by the Ptidej research team in software engineering from Concordia university and Polytechnique Montreal in Canada.
The main purpose of this study is to understand how software companies and researchers **analyse and measure change impact** especially in **multi-language software systems**, systems that combine more than one programming language (for example: JNI systems).

To achieve this objective, we want to answer the following research questions:

- What are the main causes and contexts of changes in multi-language systems?
- What are the existing tools and techniques for change impact in multi-language systems?

Please feel free to share this survey with your contacts.
Thank you for your time.

Participants Work Experience

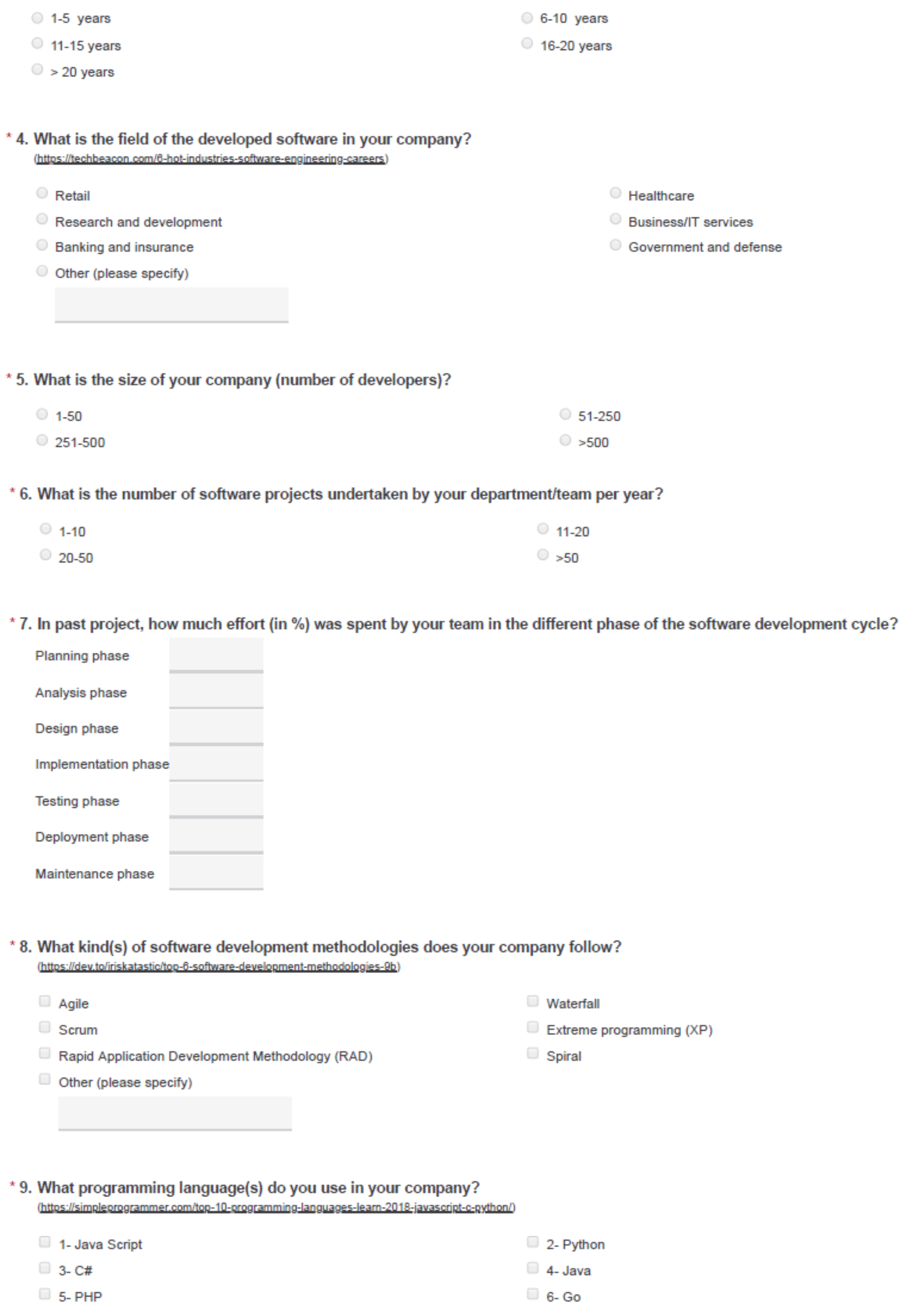
*** 1. What is your highest level of education?**

○ Bachelor	○ Master
○ PhD	○ Other (please specify)

*** 2. What is (are) your current position(s)?**
(https://targetpostgrad.com/subjects/computer-science-and-it/it-job-roles-and-responsibilities-explained)

☐ Software Engineer	☐ System Analyst
☐ Business Analyst	☐ Technical Support
☐ Network Engineer	☐ Technical Consultant
☐ Technical sales	☐ Project Manager
☐ Web Developer	☐ Software Tester
☐ Other (please specify)	

*** 3. How many years of work experience do you have in software development?**



○ 1-5 years	○ 6-10 years
○ 11-15 years	○ 16-20 years
○ > 20 years	

* **4. What is the field of the developed software in your company?**
(https://techbeacon.com/6-hot-industries-software-engineering-careers)

○ Retail	○ Healthcare
○ Research and development	○ Business/IT services
○ Banking and insurance	○ Government and defense
○ Other (please specify)	

* **5. What is the size of your company (number of developers)?**

○ 1-50	○ 51-250
○ 251-500	○ >500

* **6. What is the number of software projects undertaken by your department/team per year?**

○ 1-10	○ 11-20
○ 20-50	○ >50

* **7. In past project, how much effort (in %) was spent by your team in the different phase of the software development cycle?**

Planning phase	
Analysis phase	
Design phase	
Implementation phase	
Testing phase	
Deployment phase	
Maintenance phase	

* **8. What kind(s) of software development methodologies does your company follow?**
(https://dev.to/iriskatastic/top-6-software-development-methodologies-9b)

☐ Agile	☐ Waterfall
☐ Scrum	☐ Extreme programming (XP)
☐ Rapid Application Development Methodology (RAD)	☐ Spiral
☐ Other (please specify)	

* **9. What programming language(s) do you use in your company?**
(https://simpleprogrammer.com/top-10-programming-languages-learn-2018-javascript-c-python/)

☐ 1- Java Script	☐ 2- Python
☐ 3- C#	☐ 4- Java
☐ 5- PHP	☐ 6- Go

7- Swift	8- Rust
9- Kotlin	10- C/C++
Other (please specify)	

Multi-language Systems and Change Impact Analysis

* **10Have you used multi-language programming in your past projects?**

Yes	No

* **11What is the number of projects your company worked per year that involved developing multi-language software systems?**

between 0 and 25	between 26 and 50
between 51 and 75	between 76 and 100
>100	

* **12Have you heard about change impact analysis (CIA) in software engineering?**

Before answering, please take a look on the following simple example to have an idea about CIA:

Let's consider a multi-language software system that is built with two languages, such as Java and C: A change to one part of the system can have an impact on the other part of the system. So, if one applies a change to the Java code of the program for example, what would be the impact on the C code of the program? How be analysed? What change impact analysis tools or techniques will be used? etc.

Yes	No

Multi-language Systems and Change Impact Analysis

* **13What are recurrent situations that you faced and that demanded changes to be applied in multi-language systems?**

(M. Bano, S. Imtiaz, N. Ikram, M. Niazi and M. Usman, "Causes of requirement change - A systematic literature review"

16th International Conference on Evaluation & Assessment in Software Engineering (EASE 2012), Ciudad Real, 2012, pp. 22-31.)

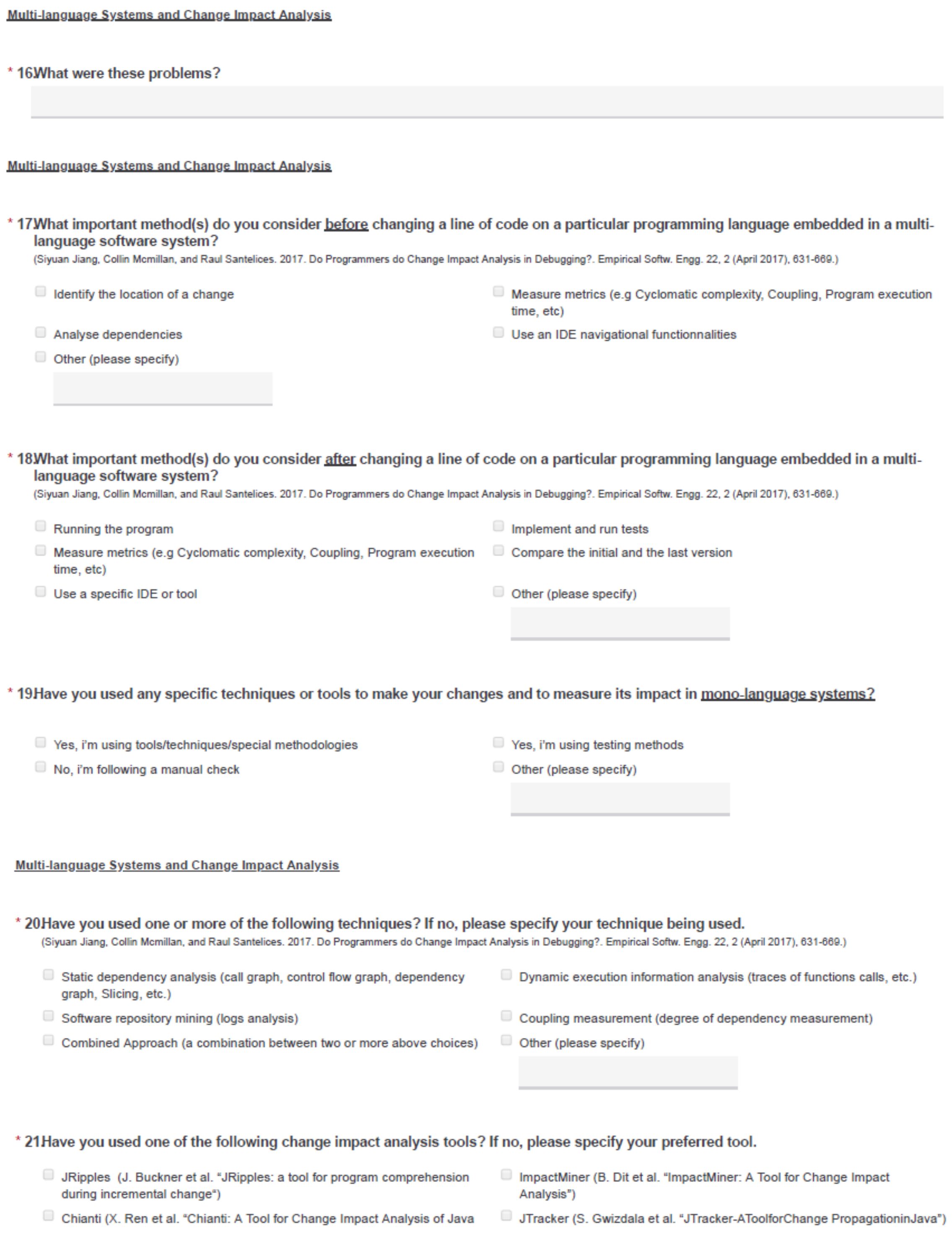
Requirements change	Miss of some features
Misunderstanding of requirements	Errors/bugs fixing
New legislation or rules	Other (please specify)

* **14How do you evaluate the challenges of change impact analysis for the following four phases?**

	1 Low challenge	2 Medium challenge	3 High challenge
Design Phase	○	○	○
Development and Implementation Phase	○	○	○
Testing and deployment Phase	○	○	○
Maintenance Phase	○	○	○

* **15Have you faced, in your department/team, problems caused by a lack of change impact analysis in multi-language systems?**

Yes	No



Multi-language Systems and Change Impact Analysis

* 16What were these problems?

Multi-language Systems and Change Impact Analysis

* 17What important method(s) do you consider before changing a line of code on a particular programming language embedded in a multi-language software system?
(Siyuan Jiang, Collin Mcmillan, and Raul Santelices. 2017. Do Programmers do Change Impact Analysis in Debugging?. Empirical Softw. Engg. 22, 2 (April 2017), 631-669.)

☐ Identify the location of a change	☐ Measure metrics (e.g Cyclomatic complexity, Coupling, Program execution time, etc)
☐ Analyse dependencies	☐ Use an IDE navigational functionnalities
☐ Other (please specify)	

* 18What important method(s) do you consider after changing a line of code on a particular programming language embedded in a multi-language software system?
(Siyuan Jiang, Collin Mcmillan, and Raul Santelices. 2017. Do Programmers do Change Impact Analysis in Debugging?. Empirical Softw. Engg. 22, 2 (April 2017), 631-669.)

☐ Running the program	☐ Implement and run tests
☐ Measure metrics (e.g Cyclomatic complexity, Coupling, Program execution time, etc)	☐ Compare the initial and the last version
☐ Use a specific IDE or tool	☐ Other (please specify)

* 19Have you used any specific techniques or tools to make your changes and to measure its impact in mono-language systems?

☐ Yes, i'm using tools/techniques/special methodologies	☐ Yes, i'm using testing methods
☐ No, i'm following a manual check	☐ Other (please specify)

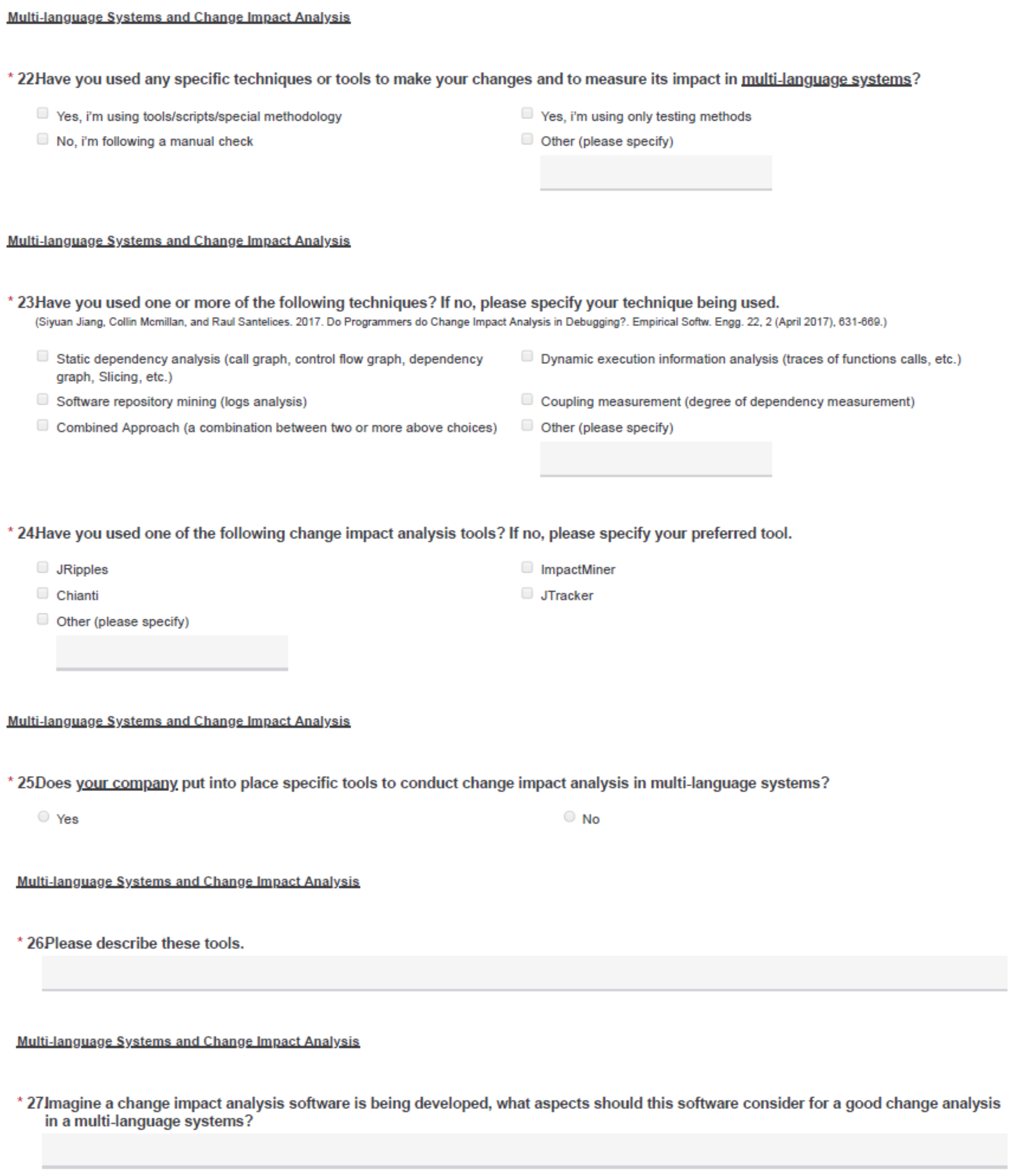
Multi-language Systems and Change Impact Analysis

* 20Have you used one or more of the following techniques? If no, please specify your technique being used.
(Siyuan Jiang, Collin Mcmillan, and Raul Santelices. 2017. Do Programmers do Change Impact Analysis in Debugging?. Empirical Softw. Engg. 22, 2 (April 2017), 631-669.)

☐ Static dependency analysis (call graph, control flow graph, dependency graph, Slicing, etc.)	☐ Dynamic execution information analysis (traces of functions calls, etc.)
☐ Software repository mining (logs analysis)	☐ Coupling measurement (degree of dependency measurement)
☐ Combined Approach (a combination between two or more above choices)	☐ Other (please specify)

* 21Have you used one of the following change impact analysis tools? If no, please specify your preferred tool.

☐ JRipples (J. Buckner et al. "JRipples: a tool for program comprehension during incremental change")	☐ ImpactMiner (B. Dit et al. "ImpactMiner: A Tool for Change Impact Analysis")
☐ Chianti (X. Ren et al. "Chianti: A Tool for Change Impact Analysis of Java	☐ JTracker (S. Gwizdala et al. "JTracker-AToolforChange PropagationinJava")



Multi-language Systems and Change Impact Analysis

* 22Have you used any specific techniques or tools to make your changes and to measure its impact in multi-language systems?

☐ Yes, i'm using tools/scripts/special methodology
☐ Yes, i'm using only testing methods
☐ No, i'm following a manual check
☐ Other (please specify)

Multi-language Systems and Change Impact Analysis

* 23Have you used one or more of the following techniques? If no, please specify your technique being used.
(Siyuan Jiang, Collin Mcmillan, and Raul Santelices. 2017. Do Programmers do Change Impact Analysis in Debugging?. Empirical Softw. Engg. 22, 2 (April 2017), 631-669.)

☐ Static dependency analysis (call graph, control flow graph, dependency graph, Slicing, etc.)
☐ Dynamic execution information analysis (traces of functions calls, etc.)
☐ Software repository mining (logs analysis)
☐ Coupling measurement (degree of dependency measurement)
☐ Combined Approach (a combination between two or more above choices)
☐ Other (please specify)

* 24Have you used one of the following change impact analysis tools? If no, please specify your preferred tool.

☐ JRipples
☐ ImpactMiner
☐ Chianti
☐ JTracker
☐ Other (please specify)

Multi-language Systems and Change Impact Analysis

* 25Does your company put into place specific tools to conduct change impact analysis in multi-language systems?

○ Yes
○ No

Multi-language Systems and Change Impact Analysis

* 26Please describe these tools.

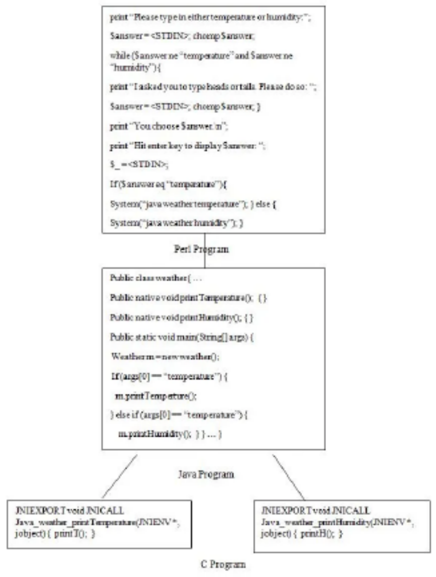
Multi-language Systems and Change Impact Analysis

* 27Imagine a change impact analysis software is being developed, what aspects should this software consider for a good change analysis in a multi-language systems?

Quick experience

Scenario: Illustration of a multi-language program (Perl, Java, and C)

"This is an example of a multi-language program (See diagram below) which is written in Perl, Java and C programming languages. This program displays temperature or humidity rate and this depends on whether the user chooses 'temperature' or 'humidity' respectively. The Main program requires the user to type 'humidity' or 'temperature'. Then a Java program (Weather.java) is invoked via system call. This program uses methods printTemperature() and printHumidity() implemented in C to display temperature and humidity rate and have respective signatures:
JNIEXPORT void JNICALL Java_weather_printTemperature (JNIEnv *, jobject)
and
JNIEXPORT void JNICALL Java_weather_printHumidity(JNIEnv *, jobject)



```
print "Please type in either temperature or humidity:";
$answer = <STDIN>; chomp $answer;
while ($answer ne "temperature" and $answer ne
"humidity"){
print "I asked you to type heads or tails. Please do so: ";
$answer = <STDIN>; chomp $answer; }
print "You choose $answer.\n";
print "Hit enter key to display $answer: ";
$_ = <STDIN>;
If($answer eq "temperature"){
System("java weather temperature"); } else {
System("java weather humidity"); }
```

Perl Program

```
Public class weather{ ...
Public native void printTemperature();  { }
Public native void printHumidity(); { }
Public static void main(String[] args) {
 Weather m = new weather();
 If(args[0] == "temperature") {
  m.printTemperture();
} else if (args[0] == "temperature") {
   m.printHumidity(); } } ... }
```

Java Program

```
JNIEXPORT void JNICALL
Java_weather_printTemperature(JNIENV *,
jobject) { printT(); }
```

```
JNIEXPORT void JNICALL
Java_weather_printHumidity(JNIENV *,
jobject) { printH(); }
```

C Program

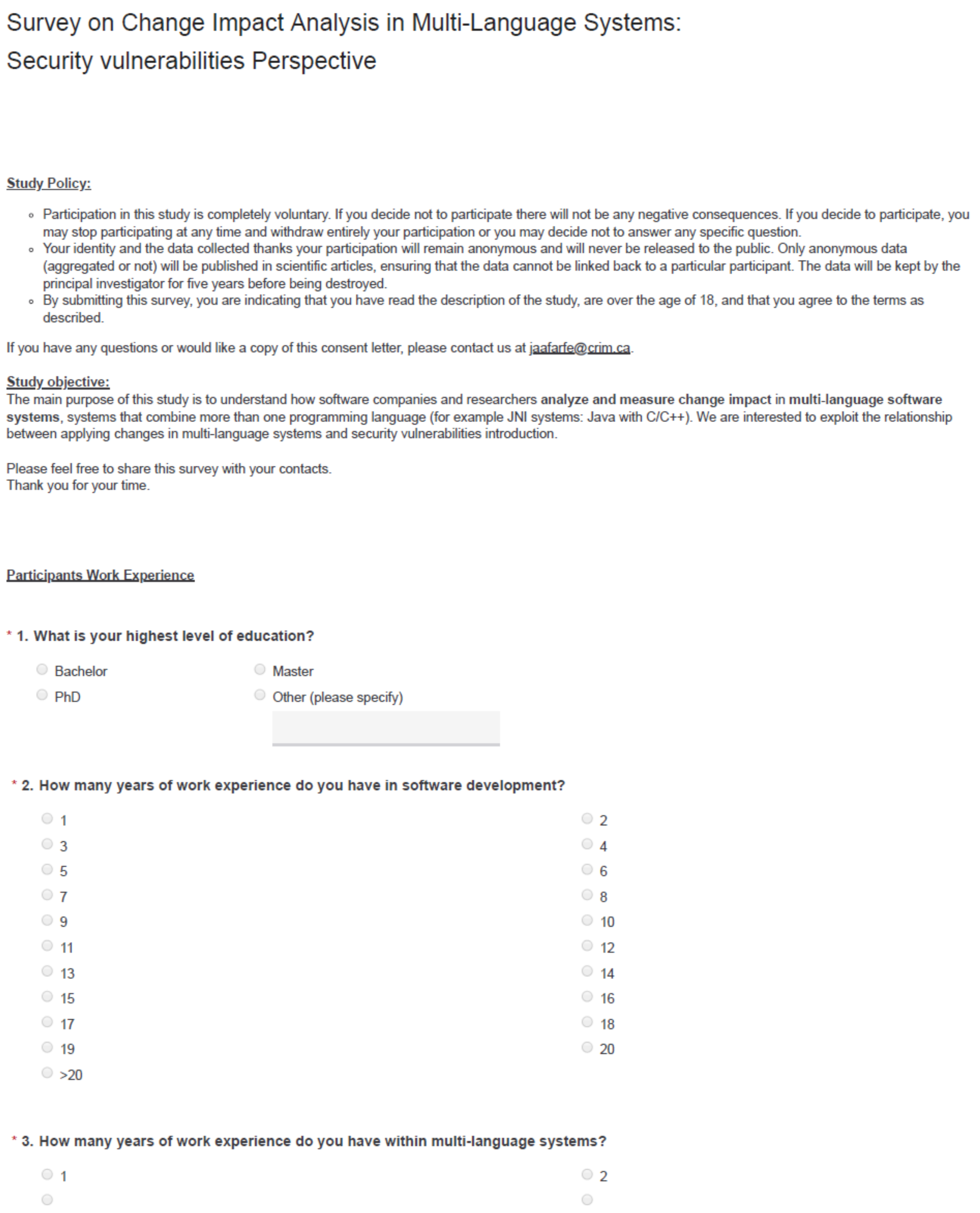
For this question, we assume that we want to add a new parameter integer, to identify week days, to the native method **Java_weather_printTemperature(JNIEnv *, jobject)** so that the native method becomes **Java_weather_printTemperature(JNIEnv *, jobject, jint).**

*** 28.What steps will you follow when applying this change?**

*** 29.What are the different tests that you would implement and execute in this case?**

*** 30.How will you measure and analyse the impact of this change (for example: methodology, techniques, or tools that will be used)?**

31. Could we contact you for a short interview by phone, Skype, or e-mail? We guarantee your anonymity and that of your answers. If yes, please add your e-mail adress or Skype pseudo to contact you.



Survey on Change Impact Analysis in Multi-Language Systems: Security vulnerabilities Perspective

Study Policy:

- Participation in this study is completely voluntary. If you decide not to participate there will not be any negative consequences. If you decide to participate, you may stop participating at any time and withdraw entirely your participation or you may decide not to answer any specific question.
- Your identity and the data collected thanks your participation will remain anonymous and will never be released to the public. Only anonymous data (aggregated or not) will be published in scientific articles, ensuring that the data cannot be linked back to a particular participant. The data will be kept by the principal investigator for five years before being destroyed.
- By submitting this survey, you are indicating that you have read the description of the study, are over the age of 18, and that you agree to the terms as described.

If you have any questions or would like a copy of this consent letter, please contact us at jaafarfe@crim.ca.

Study objective:
The main purpose of this study is to understand how software companies and researchers **analyze and measure change impact** in **multi-language software systems**, systems that combine more than one programming language (for example JNI systems: Java with C/C++). We are interested to exploit the relationship between applying changes in multi-language systems and security vulnerabilities introduction.

Please feel free to share this survey with your contacts.
Thank you for your time.

Participants Work Experience

*** 1. What is your highest level of education?**

○ Bachelor	○ Master
○ PhD	○ Other (please specify)

*** 2. How many years of work experience do you have in software development?**

○ 1	○ 2
○ 3	○ 4
○ 5	○ 6
○ 7	○ 8
○ 9	○ 10
○ 11	○ 12
○ 13	○ 14
○ 15	○ 16
○ 17	○ 18
○ 19	○ 20
○ >20	

*** 3. How many years of work experience do you have within multi-language systems?**

○ 1	○ 2
○	○

3	4
○ 5	○ 6
○ 7	○ 8
○ 9	○ 10
○ 11	○ 12
○ 13	○ 14
○ 15	○ 16
○ 17	○ 18
○ 19	○ 20
○ >20	

*** 4. What is the field of the developed software in your company?**
(https://techbeacon.com/6-hot-industries-software-engineering-careers)

○ Retail	○ Healthcare
○ Research and development	○ Business/IT services
○ Banking and insurance	○ Government and defense
○ Other (please specify)	

*** 5. What programming language(s) do you use in your company?**
(https://simpleprogrammer.com/top-10-programming-languages-learn-2018-javascript-c-python/)

☐ 1- Java Script	☐ 2- Python
☐ 3- C#	☐ 4- Java
☐ 5- PHP	☐ 6- Go
☐ 7- Swift	☐ 8- Rust
☐ 9- Kotlin	☐ 10- C/C++
☐ Other (please specify)	

<u>Multi-language Systems and Change Impact Analysis</u>

*** 6. Have you used the multi-language programming in one past project? (Multi-language systems are written in more than one programming languages, as examples, JNI systems that combine Java with C/C++, or a system that combine Javascript and Phython)**

☐ Yes ☐ No

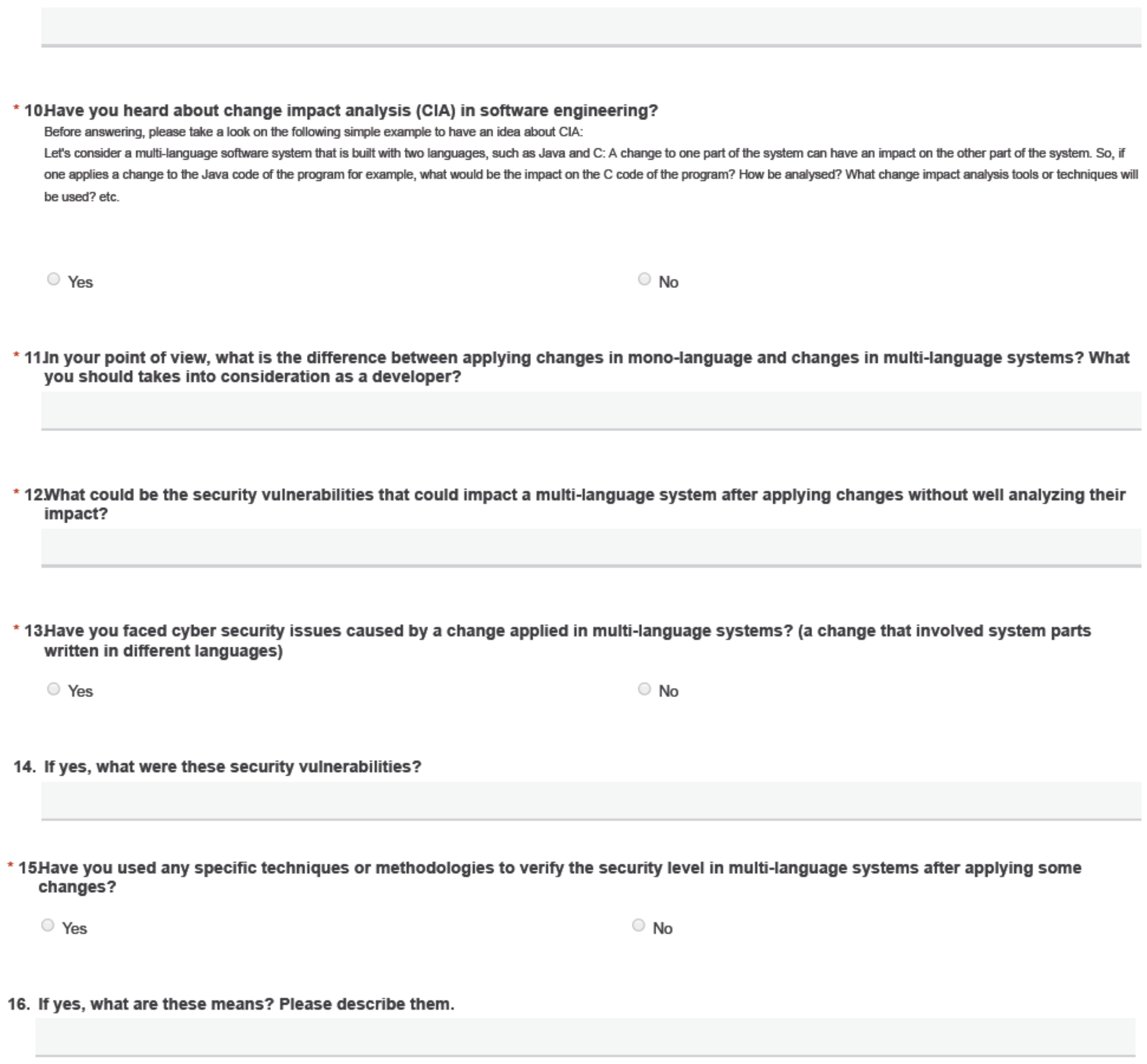
*** 7. Have you maintained before a multi-language system?**

☐ Yes ☐ No

*** 8. Do you think that the multi-language systems are more or less vulnerable to cyber security issues comparing with mono-language systems? Please explain why.**

○ Yes ○ No

*** 9. Please explain your choice.**



* 10.Have you heard about change impact analysis (CIA) in software engineering?

Before answering, please take a look on the following simple example to have an idea about CIA:

Let's consider a multi-language software system that is built with two languages, such as Java and C: A change to one part of the system can have an impact on the other part of the system. So, if one applies a change to the Java code of the program for example, what would be the impact on the C code of the program? How be analysed? What change impact analysis tools or techniques will be used? etc.

○ Yes ○ No

* 11.In your point of view, what is the difference between applying changes in mono-language and changes in multi-language systems? What you should takes into consideration as a developer?

* 12.What could be the security vulnerabilities that could impact a multi-language system after applying changes without well analyzing their impact?

* 13.Have you faced cyber security issues caused by a change applied in multi-language systems? (a change that involved system parts written in different languages)

○ Yes ○ No

14. If yes, what were these security vulnerabilities?

* 15.Have you used any specific techniques or methodologies to verify the security level in multi-language systems after applying some changes?

○ Yes ○ No

16. If yes, what are these means? Please describe them.