The Journal of Systems & Software 206 (2023) 111825

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Pragmatic evidence of cross-language link detection: A systematic literature review☆

Saira Latif [a], Zaigham Mushtaq [b], Ghulam Rasool [c], Furqan Rustam [d], Naila Aslam [e], Imran Ashraf [f,*]

[a] University of Central Punjab, Lahore 54000, Pakistan
[b] The Islamia University, Bahawalpur 63102, Pakistan
[c] COMSATS University Islamabad, Lahore Campus, 54000, Pakistan
[d] School of Computer Science, University College Dublin, Belfield Dublin 4, Ireland
[e] School of Electronics and Information Engineering, Hebei University of Technology, Tianjin 300401, China
[f] Information and Communication Engineering, Yeungnam University, Gyeongsan 38541, Republic of Korea

ARTICLE INFO

Article history:
Received 28 April 2022
Received in revised form 8 August 2023
Accepted 20 August 2023
Available online 23 August 2023

Keywords:
Software development
Reverse engineering
Multilingual source code
Cross-language link detection
Source code analysis
Cross-language dependencies
Multilingual software applications
Graph databases
Machine learning in software engineering
Software maintenance
Systematic literature review

ABSTRACT

There is a rising trend for heterogeneous software applications involving multilingual source code. The key focus of reverse engineers is to unravel the cross-language links (XLLs) and their dependencies. This study aims to perform a systematic literature review (SLR) to compile different approaches, tools, techniques, and shortcomings of such techniques and understand the XLLs and their dependencies while performing reverse engineering on state-of-the-art software applications. This SLR selects 76 primary studies and uses them to create a 'go-to' literature database, where professionals from software engineering could find all the content pertinent to the analysis and XLL detection for major multilingual applications like Java enterprise applications, Android applications, etc. It has been observed that traditional source code analysis mechanisms to reverse engineer contemporary software applications face scores of problems and limitations that need to be addressed. To assist the community in the above-mentioned goal, a general schema with definitions of XLLs and associated concepts is furnished. This study provides an SLR on XLLs, comprehensive taxonomy called cross-language analysis, which incorporates all the methods for XLL detection in multilingual source code. By pursuing future directions suggested in the end, researchers and practitioners can advance the field of multilingual applications; such as Enterprise resource planning (ERP) solutions, and cross-language software corpora, leading to improved software development practices and better understanding of language interactions in multilingual environments. The research data provided in the survey presents a comprehensive analysis of the complexities involved in working with diverse programming languages and frameworks, offering valuable insights for language technology researchers, software developers, academics, and decision-makers. This integration will enable them to identify and manage dependencies across diverse languages, leading to more efficient and reliable multilingual software systems.

1. Introduction

Software development paradigms (either mobile applications (apps) or large-scale enterprise apps, etc.) have shifted from single source code applications to multilingual applications. There is heterogeneity across these applications that contains various cross-language (XL) artifacts that are interdependent (Adams and Kear, 2003). Problems arise when XL artifacts communicate with each other and create dependencies that manifest in various forms like direct, indirect, or framework-based. A large number of possible combinations of languages exacerbates the problem to a greater degree (Polychniatis et al., 2013). Multilingual applications like Spring (Varanasi and Belida, 2015), YouTube (Youtube, 2021), Eclipse (Eclipse Foundation, 2021), etc. contain standard programming languages like C, C++, and Java, user interface (UI) description languages (hypertext markup language (HTML), extensible markup language (XML)) and database languages (structure query language (SQL)) are also used, in addition to the domain-specific languages (Mayer, 2017). Hence, the term 'multilingual' is often called 'multi-paradigm"' which includes many different language artifacts in a program (Strein et al., 2006; Lozano et al., 2016). Another reason for multilingual applications

☆ Editor: Lingxiao Jiang.
* Corresponding author.
E-mail address: imranashraf@ynu.ac.kr (I. Ashraf).

https://doi.org/10.1016/j.jss.2023.111825
0164-1212/

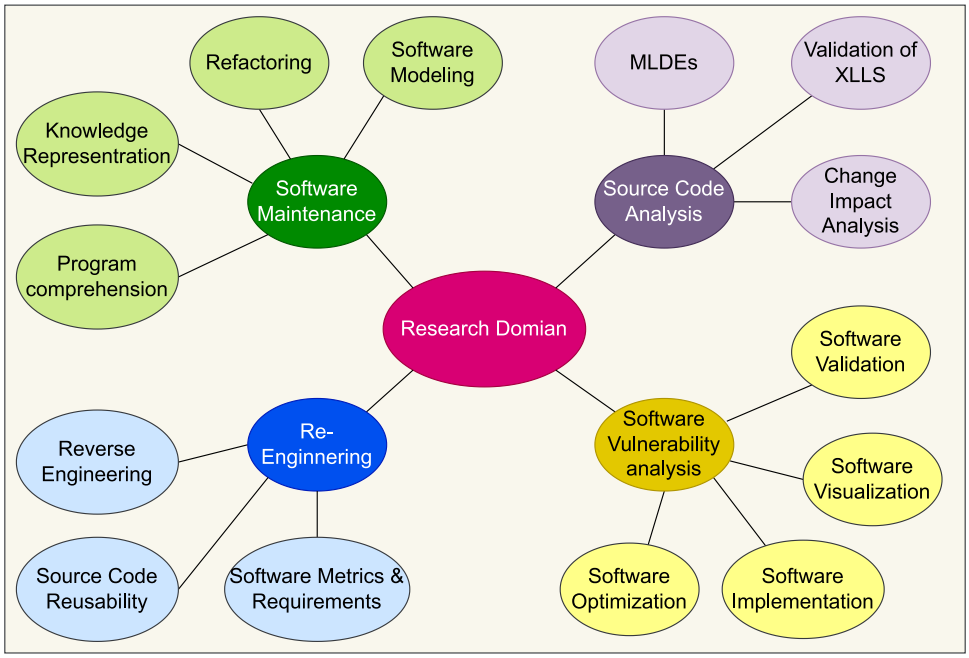


Fig. 1. Diversification and significance of cross-language domains.

(MLAs) is the integration of legacy systems, as heritage source code contains a lot of information. For all these reasons, diversity in programming languages is inevitable (Linos et al., 2007; Oliva et al., 2011).

Cross-language links (XLLs) refer to the underlying dependencies, which are accessed using shared names across different programming language borders. For example, one such instance is the use of 'textfield' in HTML and Java languages. These go unchecked by tools written for the language (Strein et al., 2006; Mushtaq et al., 2017b; Mushtaq and Rasool, 2015). In a multilingual environment (i.e. combination of libraries, frameworks, and runtimes), there are several ways to express XLL in a code. Thus, developers must remain in full control of XLLs (Mayer, 2017). Consequently, identification and management of links in large MLAs are cumbersome owing to the complexity and heterogeneity of their nature (Mayer and Bauer, 2015). Fig. 1 shows the diversification and significance of XL Domains.

This study aims at presenting a systematic literature review to cover XL dependencies in multilingual software applications (MLSAs). Keeping in view the large number of articles published during the last decade, it is pertinent to collect, critically analyze, classify, and synopsize state-of-the-art research. To the authors' best knowledge, no systematic review of tools and techniques for detecting XLLs in MLSAs is present in the existing literature. The major contributions of this paper are summarized as follows

- A comprehensive systematic literature review (SLR) on XL dependencies is presented, following a devised compact research methodology to collect, analyze, and include or exclude criteria.
- A general schema for cross-language analysis is presented where different types of analytical approaches are investigated like static analysis, dynamic analysis, and hybrid semantic analysis thereby revealing the pros and cons of each type of analysis,
- A taxonomy for cross-language analysis is designed to cover different components like a parser, internal representation, analysis of representation, etc.
- Research gaps and future research directions are outlined for the research and development community by formulating five relevant research questions and discussing the findings in comprehensive detail.

The remaining sections of the research paper are organized as follows. Section 2 presents the multilingual source code analysis and cross-language linking mechanism. Section 3 discusses the research methodology adopted to conduct SLR where research questions are defined, the domain for the literature review is selected, important sources of information for data gathering are defined, and search criteria, search string, and the information extraction procedure is defined as well. The assessment and discussions of devised research questions are presented in Section 4. Discussions on findings are given in Section 5 where the future directions are also presented from the perspective of the research gaps found in existing studies. In the end, the conclusion and future work is presented in Section 8.

2. Source code analysis and manipulation

Analyzing the source code is the backbone of design recovery and structural examination of software applications. The significance of source code analysis is extremely vital for software maintenance, reusability, re-engineering, refactoring, and re-documentation (Strein et al., 2006; Moshenska, 2016). Today, software applications are modernized and updated day by day. The estimated size of source code may exceed more than a trillion lines by 2025 (Srinivas et al., 2016). As the complexity and size of software applications expand, so does the significance of source code analysis. Moreover, the paradigm of software applications has shifted from homogeneous to heterogeneous multilingual applications.

2.1. Significance of identifying cross-language links

Java Enterprise Edition is a multi-tiered platform that consists of several language components including JSPs, EJBs, Servlets, JDBC, etc. They are heterogeneous and composed of various cross-language components. Unraveling links between cross-languages is important for design recovery, which leads to the analysis of systems functioning on multilingual aspects.

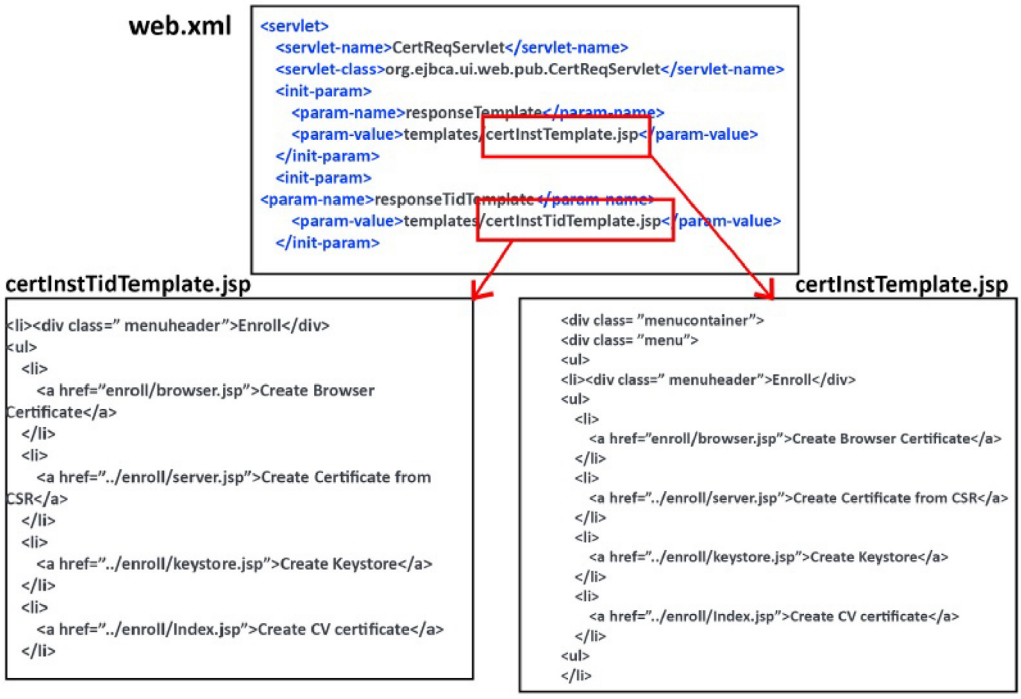


Fig. 2. Example of cross-language components (Moshenska, 2016).

To elaborate on the type of dependencies and structure of cross-language links in multilingual applications and strengthen the concept, an example of the J2EE Pattern (Moshenska, 2016), in the form of a combination of XML, HTML, and JSP files are shown in Figure 2.1. Different cross-language components are associated with each other. This pattern is implemented in the source code as XML, Java, and Property files. XML artifacts are mapped to the Java files and the components of property files are associated with the Java file. The overall functionality is executed as the combination of interdependent cross-language components.

2.2. Multilingual source code environment

Multilingual applications (MLAs) are developed using different languages in the source code (Strein et al., 2006). To elaborate on the type of dependencies and structure of cross-language links in multilingual applications, an example of the J2EE Pattern (Jawawi et al., 2007) is shown in Fig. 2. It uses a combination of XML, HTML, and JSP files.

Analyzing multilingual applications is challenging because these applications contain cross-language artifacts, which are interconnected (Marburger and Westfechtel, 2010; Bravenboer and Visser, 2004; Moshenska, 2016; Srinivas et al., 2016; Jawawi et al., 2007; Canfora and Di Penta, 2007; Chikofsky and Cross, 1990). Therefore, to fully comprehend these applications, it is necessary to identify, extract, and resolve cross-language artifacts and their dependencies. The term XLL (short for cross-language link) refers to these underlying dependencies. These dependencies are essential for working MLAs: Since XLL is beyond the scope of a single language, these are accessed by using shared names across language borders. Consequently, they tend to go unchecked by the tools written for the language (Strein et al., 2006; Mushtaq et al., 2017b; Marburger and Westfechtel, 2010).

Handling relations in cross-language artifacts is not supported during the development process (Marburger and Westfechtel, 2010). Thus, developers must remain in full control of XLLs because of massive frameworks, runtimes, and libraries that underline several ways to indicate a cross-language link (XLL) in code (Varanasi and Belida, 2015). Moreover, there is no comprehensive solution available to analyze multilingual applications (Bravenboer and Visser, 2004; Moshenska, 2016; Srinivas et al., 2016; Jawawi et al., 2007; Canfora and Di Penta, 2007; Aslam and Ashraf, 2014; Rashid et al., 2013). Program comprehension and architectural extraction in large and complex applications are quite difficult. Due to the existence of cross-language artifacts and massive source codes of multilingual applications, the process of analyzing and design recovery is challenging for the research community

Java Enterprise Edition is a multi-tiered platform that consists of several components including JSPs, EJBs, Servlets, JDBC, etc. (Jawawi et al., 2007). These components are heterogeneous and composed of various cross-language components (Lozano et al., 2016). This pattern is implemented in the source code as XML, Java, and Property files. XML artifacts are mapped to the Java files and the components of property files are associated with the Java file. The overall functionality is executed as the combination of interdependent cross-language components. Unraveling links between cross-languages is important for design recovery which leads to the analysis of systems functioning on multilingual aspects (Jawawi et al., 2007).

3. Research methodology

The principal aim of this systematic review is to study, compare, analyze, evaluate, summarize, and classify the finest available procedures, techniques, models, and tools for analyzing XLLs in multilingual software systems. The process of SLR helps to designate the search plan like making search strategies to extract relevant literature. It includes research questions, scope inclusion, and exclusion criteria, sources of information, and literature assessment criteria. The process of this systematic literature review follows the steps shown in Fig. 3.

3.1. Systematic literature review

The SLR is a disciplined procedure of presenting the findings, extracted from studies related to a particular topic. It is associated

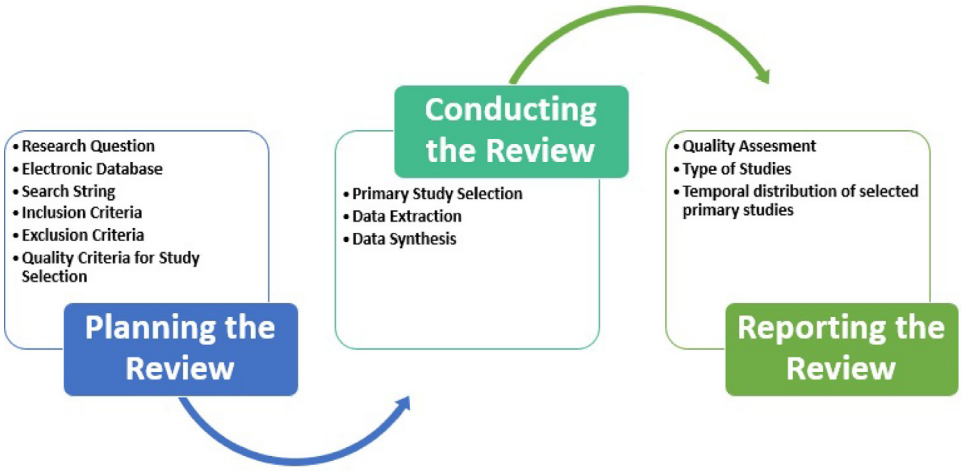


Fig. 3. Phases followed in the systematic literature review.

with the research questions in an iterative manner. Further, SLR fuses the research work in a methodical and scientific manner (Mushtaq et al., 2017a; Khan et al., 2017). The purpose of conducting SLR is to provide an unbiased appraisal of the research subject by adopting a dependable, accurate, and auditable approach. The approach of SLR is acquired to ascertain issues, pertaining to the detection of XLLs in multilingual applications. The adopted systematic search procedure and the methodology used to select primary studies are provided by Kitchenham et al. (1999), Kitchenham (2004) and Kitchenham et al. (2002). The population consists of publications from selected sources. Strategies are applied to analyze the analysis of multilingual applications. SLR comprises three phases which are

1. Review planning,
2. Review conducting, and
3. Review reporting

All three phases are discussed in detail in the following sections. The phases of SLR, along with their sub-strata, are illustrated in Fig. 3.

3.2. Planning review

Planning the review phase consists of six steps. All the steps are discussed in detail here.

3.2.1. Research questions

The elementary goal of this SLR is to abridge the current state-of-the-art research in multilingual source code analysis (MLSCA) and to categorize procedures used by existing literature. MLSCA techniques are evaluated empirically to identify potential areas of future research. For this purpose, the following research questions (RQs) are formulated.

RQ1: What are the potential research domains for MLSCA?

RQ2: What are the prevalent research techniques reported in the literature for cross-language link detection?

RQ3: What are the limitations of prevalent research techniques and tools used for cross-language link detection?

RQ4: How to develop a conceptual schema and taxonomy that covers all essential components of cross-language analysis (XLA)?

RQ 5. What are the challenges faced by the research community in cross-language link detection?

RQ 6. What are the future contributions to the recognition of cross-language links?

3.2.2. Electronic databases

A total of 5 suitable data repositories are used based on the existing research knowledge and recommendations given in Afzal et al. (2009). Electronic databases which are selected in the research study are as follows

- IEEE Xplore
- Science Direct
- Semantic Scholar
- Springer
- Google Scholar

Since the review topic consists of Cross-language link detection in multilingual software applications, the focus is placed on the main research databases that specialize in the computer and software engineering domain. The three largest and most commonly used databases in the software engineering field are IEEE Xplore, Google Scholar, and Semantic Scholar. After going through these research venues, we extended the search venues to Springer and Science Direct in order to get more reliable studies. Since research results are indexed and cited in these databases, we have opted to search in these databases by using well-formulated search strings. Online Wiley is also searched to get primary research studies, but three papers are found after applying our search string. Therefore, we removed it from the searched sources.

3.2.3. Search string

The search string used to extract the selected studies is as follows
("Multi" *OR* "Cross" *OR* "Heterogeneous" *OR* "Hybrid") *AND* ("Language Detection" *OR* "Lingual Detection" *OR* "Link Detection" *OR* "Dependency Detection") *AND* ("Source Code" *OR* "Source Code Analysis" *OR* "Software Program" *OR* "Software Application" *OR* "Software Evolution" *OR* "Source Code Parsing" *OR* "Source Code Refactoring").

3.2.4. Inclusion criteria

The following criteria for inclusion are adopted for the current study

- The period for selection ranges between January 2005 and December 2020.
- The research is published in the English language.
- Only full-length papers are considered.
- The selection process strictly follows the search string.
- The assessment mechanism is under the specified criteria.

Table 1
Quality criteria questions for selecting articles.

S. No.	Quality criteria question	Score
QA 1	Does the study discuss any XL dependencies in MLSC?	"Yes = 1, Partial = 0.5, No = 0"
QA 2	Does the study report any key research challenges encountered in the detection of XLLs and their dependencies in MLSC?	"Yes = 1, Partial = 0.5, No = 0"
QA 3	Does the study use any tool or technique for detecting and visualizing XL dependencies in MLSC?	"Yes = 1, Partial = 0.5, No = 0"
QA 4	Does the study reveal any limitations of prevalent research techniques and tools used for detecting XL dependencies in MLSC?	"Yes = 1, Partial = 0.5, No = 0"

3.2.5. Exclusion criteria

Studies were excluded based on the below-defined criteria

- 'Slides', 'tutorial', 'editorials', 'posters', and other non-peer reviews were discarded as they were not published in any digital library.
- Likewise, blogs are excluded as their authenticity is doubtful.
- Publications other than English are excluded.
- Studies that do not mention the XLLs-related dependencies in MLAs are also excluded.

3.2.6. Quality criteria for study selection

Quality criteria of the selected studies are defined to evaluate the quality of the chosen studies. Those studies that properly answer the quality criteria questions are scored 1 point and 'yes'. Similarly, the studies that contain partial knowledge regarding quality criteria questions scored 0.5 points. However, those studies that do not answer the quality assessment questions are scored 0. The quality criteria questions are provided in Table 1.

3.3. Conducting review

Conducting the review is the next step of SLR where existing literature is examined in light of the steps defined in the 'planning the review' phase. This process helps to extract the pertinent research works from the existing literature. Conducting the review has three steps which are discussed ahead.

3.3.1. Primary study selection

Several articles are shortlisted herein. The tollgate approach proposed in Afzal et al. (2009) has been adopted to improve the primary study selection process. It comprises five levels where each level has a criterion, as shown in Table 2. All selected studies are discarded at the level at which they fail to fulfill the criterion.

1st Level: Searching the pertinent studies on the basis of the search query and criteria for inclusion.

2nd Level: Including and excluding the studies according to abstract and title.

3rd Level: Including and excluding the studies in accordance with the introduction and conclusion.

4th Level: Including and excluding the studies on the basis of the full text.

5th Level: Including the final primary studies based on quality assessment criteria.

Table 2
Selected papers from electronic datasets.

Database	Level 1	Level 2	Level 3	Level 4	Level 5
Google Scholar	1110	123	52	23	19
IEEE Xplore	307	97	49	29	29
Semantic Scholar	103	29	19	15	14
Science Direct	373	36	9	5	3
Springer	133	31	22	13	11
Total	**2025**	**312**	**151**	**86**	**76**

3.3.2. Data extraction

Data and findings collected from the primary studies are recorded in sheets. This sheet provides the details in which data is derived from the primary studies. It also provides a clear account and data relationship to the research queries. The following data is procured: 'title' and 'author of the paper', 'publication year', 'type of source' (conference/journal), 'geographical distribution', 'tools and technique', 'database', 'languages', and 'research type'.

3.3.3. Data synthesis

A comprehensive list of multilingual cross-language dependencies along with its tools and techniques is presented by synthesizing the data from 76 shortlisted primary research studies. The selected studies contain the contribution of different parts of the study like methods, findings, supporting material, etc. which creates a complication for the peer review. Similarly, the quality, level of detail, and location of the provided information are different. This condition has a high probability of missing key information, therefore, it is performed carefully with the help of peer reviewers.

3.4. Reporting review

This phase is carried out after the completion of conducting the review and comprises six steps, explained ahead.

3.4.1. Quality assessment

The shortlisted primary studies are subjected to quality criteria at the 4th level. Research articles that do not satisfy the quality assessment criteria (with scores less than 2) are excluded. Studies with quality greater than or equal to 50% are selected. Out of 89 total primary studies, 13 studies are discarded due to low quality while 76 studies satisfying the quality assessment criteria are selected. Table A.3 in Appendix presents the quality appraisal of the shortlisted primary research studies.

3.4.2. Type of studies

The selected 76 primary studies are categorized into six types, as shown in Fig. 5. It shows the distribution of types of studies with respect to framework, tools, techniques, taxonomy, etc.

3.4.3. Temporal classification of primary studies

Fig. 6 illustrates the distribution of shortlisted research papers in chronological order of publication. All the primary studies are published between 2005 and 2019. It is observed that 10 studies are published between 2005 and 2009 while 37 studies are published between 2010 and 2014. In addition, 29 studies are published from 2015 to 2019. Conclusively, 49% of studies are published from 2010 to 2014 whereas 12 studies are published in 2013.

3.4.4. Source type

The source type is based on a total of 76 primary studies that discuss multilingual source code analysis in terms of XLL detection. The selected studies are subdivided into four categories including 55 conference proceedings, 17 journal papers, 2 articles, and 2 workshops. Fig. 7 shows the source type of multiple research publications.

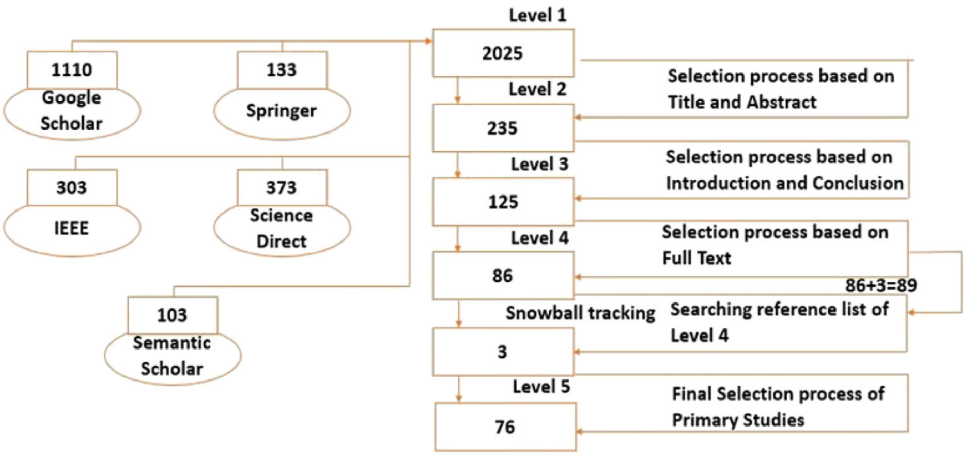


Fig. 4. Tollgate approach adopted for primary studies selection.

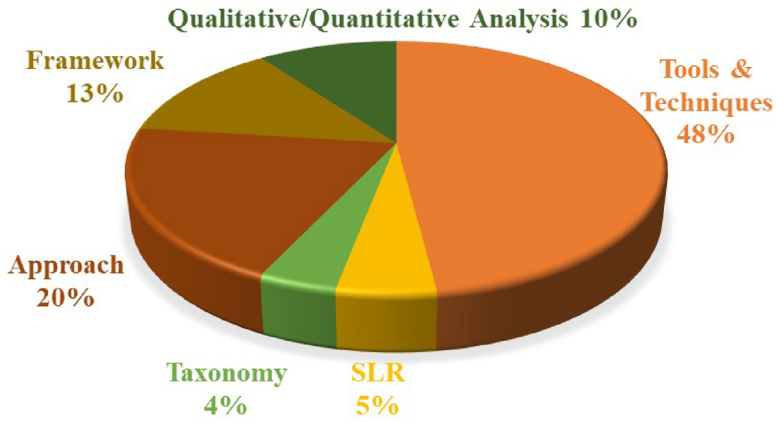


Fig. 5. Types of selected primary studies.

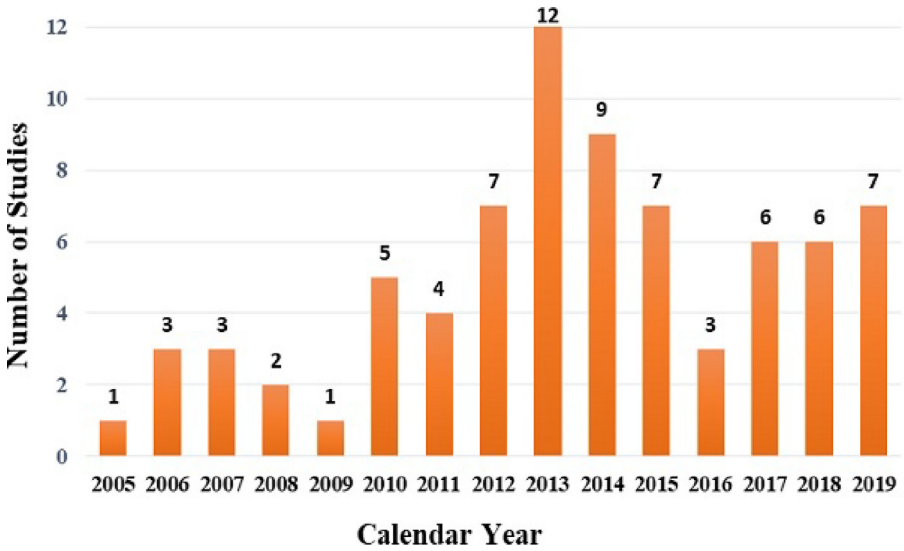


Fig. 6. Temporal classification of the selected primary studies.

3.4.5. Geographical distribution of primary studies

Geographical distribution is presented to help readers understand the trend of the topic under investigation. The review is not restricted by the location of the research. Fig. 8 depicts the distribution of locations where the studies are conducted. The 76 primary studies are published in 22 different regions which include Argentina, Japan, Canada, Serbia, Denmark, Netherlands, Austria, Pakistan, India, Germany, the US, Iran, France, Switzerland, Sweden, China, Hungry, Romania, Brazil, Italy, and Norway. It has been observed from the literature that a maximum number of studies regarding cross-language link detection are published in Germany (12 primary studies) and the US (contributing 10

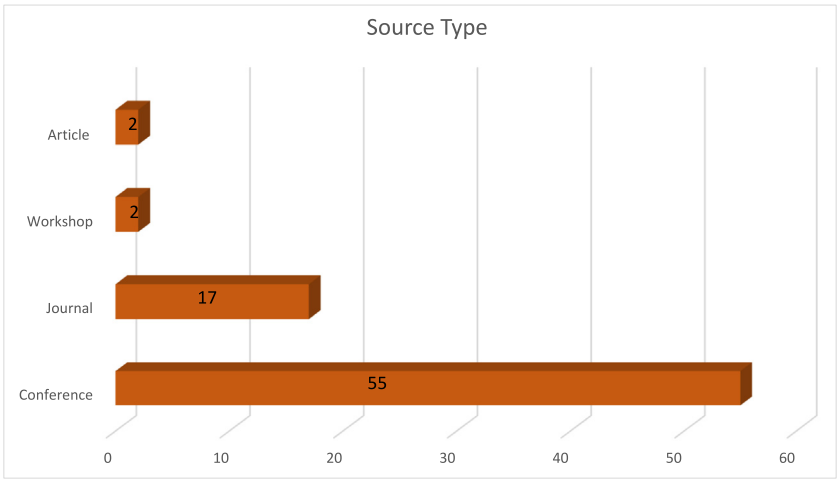


Fig. 7. Source type of primary studies.

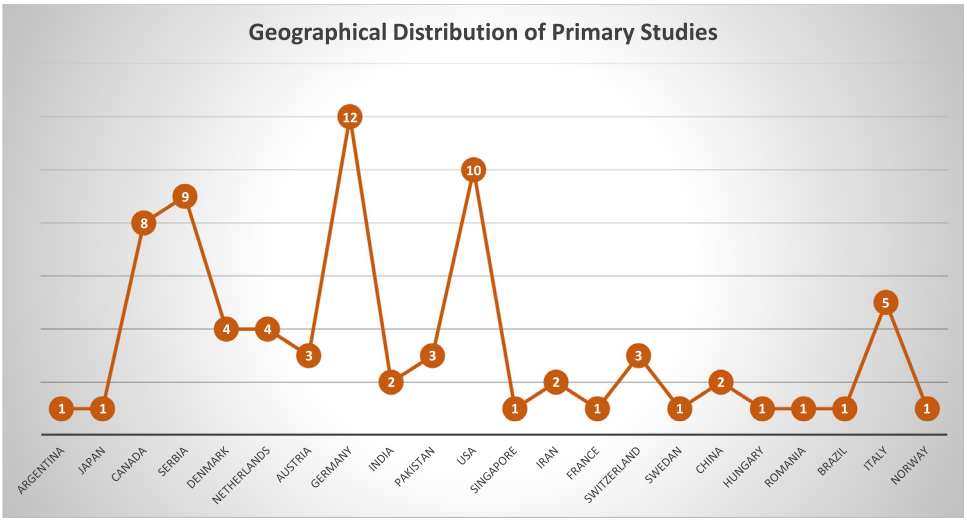


Fig. 8. Geographical distribution of primary studies.

primary studies). This distribution shows the growing trend of cross-language link detection in multilingual source code analysis in these two countries as compared to the others.

4. Assessment and discussion of research questions

Advanced software applications employ different programming languages which makes software testing challenging. It is impossible to overlook the impact of analyzing complex source codes and detecting the XLLs in modern applications (Polychniatis et al., 2013). Its usefulness is proven and demand is growing with the increasing complexity and magnitude of the source codes (Adams and Kear, 2003; Binkley, 2007; Cerulo, 2006). Analysis of shortlisted 76 primary studies is performed in this section in light of the research queries. Post-analysis and facts from miscellaneous research domains of MLSCA are stated followed by a detailed discussion of each research question.

4.1. RQ 1: What are the potential research domains for multilingual source code analysis?

Fig. 9 shows major research domains that have been extracted from the selected research papers. Domains are categorized based on three analysis mechanisms including static, dynamic, and hybrid semantic analysis. The most cited and discussed research domain is software maintenance and evolution which is discussed in 33 research articles.

4.1.1. Software implementation

Multilanguage software development (MLSD) is extremely common in open-source applications. Whenever different languages are used to develop a project, they usually encode different features of a framework. Such features do not work in isolation, instead, they are linked through a medium. Most often, names (common identifiers) are employed to create connections between discrete artifacts of code. Consequently, the same is used between the languages involved (Pfeiffer and Wasowski, 2011). Using such XLLs can act as an explicit interface specification. However, such links are mostly found between domain-specific languages (DSLs) and general purpose languages (GPLs) and more implicitly are scattered throughout the code under the requirements of a certain framework/library (Mayer et al., 2017).

4.1.2. Software maintenance and evolution

One principal challenge in software maintenance of homogeneous systems is to predict the 'change propagation'. But, accessing chronicles and code for analysis is impractical most of the time, especially for heterogeneous sources. Advanced web applications (WAs) comprise heterogeneous components. Low

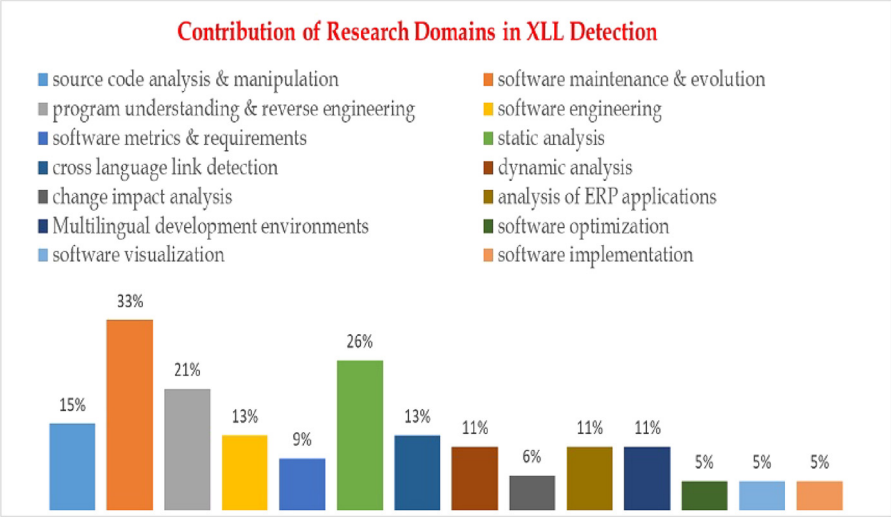


Fig. 9. Research domains.

initial cost and tight deadlines in software development have made reusability and maintenance extremely difficult (Nguyen et al., 2012). With the increase in the application scale, the maintenance cost also increased which reduces the quality and life of the software. Therefore, efficient tools are needed to understand MLAs globally (Shatnawi et al., 2018a).

4.1.3. Software refactoring

Language semantics during software development and refactoring cannot be ignored (Strein et al., 2006). The mere presence of XLLs makes a developer concerned about the changes. In our view, research on language linking is more important than mere refactoring. Before discussing refactoring, linking errors and warnings must be considered. Refactoring may even be refused if artifacts related to XLLs could not be efficiently resolved (Rakić et al., 2013). As a result, automated multilingual refactoring (MLR) does not remain a good option because of the presence of several types of artifacts and modified semantics (Mayer and Schroeder, 2014). The tools for modern-day refactoring are language dependent and thus oppose efficient incorporation of development environments (Mayer and Schroeder, 2013a). Advanced multilingual development environments (MLDEs) require support for cross-language refactoring (XLR) as existing IDEs provide refactoring support for solitary languages only (Van Der Storm and Vinju, 2015).

4.1.4. Software security

The static analysis focuses on strategies that parse the source or bytecode of a program. It usually traverses program trails to detect the dependencies. The strategies of static analysis have been proposed for various tasks e.g. assessment of security for Android apps, detection of app clones, automation of test case generation, or unearthing non-functional impediments about energy or performance (Li et al., 2017). With an increase in the complexity and size of web applications, the vulnerabilities across applications also increase. A fully automated source code reviewer is the need of the hour to handle security vulnerabilities because manual solutions are sluggish, costly, and insufficient (Lyons et al., 2017).

4.1.5. Software quality assurance

Quality assurance and debugging are vital requirements of software applications especially when such applications are getting large and complex (Albertsson, 2006). As a result, the cost of quality assurance has considerably increased. Existing software undergoes cross-language bug localization problems as they support homogeneous applications only. Effective tool support is also obligatory for ensuring quality in cross-language applications (Linos et al., 2007). Available techniques of bug fixing support homogeneous applications and the bug account does not work for the source code of heterogeneous applications (Xia et al., 2014).

4.1.6. Program comprehension & reverse engineering

Trends in software development demand greater efforts in comprehending legacy systems, which encourages reverse engineering and re-engineering. Deriving information on software change for multilingual contexts is vital for understanding MLAs (Jiang et al., 2013). Understanding the topology of web aggregates in web applications (WAs) is an additional hurdle. Compatibility of tools is needed to extract, analyze, and visualize big hypertext web apps (Misra et al., 2012) and to automatically reverse engineer the complex heterogeneous systems. The existing techniques to perform reverse engineering on object-oriented applications lack in reverse engineering the advanced enterprise applications (Perin, 2012).

4.1.7. Multilingual development environments

Current IDEs lack in providing compatibility for multilingual applications as they are language-specific and fail to efficiently process cross-language systems (Strein et al., 2006). It is also observed that contemporary languages largely use application programming interfaces (APIs) specific to platforms, raising interoperability reservations. They must be compatible with the portability of multiple languages across platforms (Grimmer et al., 2018). The current development landscape does not fully support relations across multi-language artifacts (Pfeiffer and Wasowski, 2012b). Moreover, they fail to perform refactoring of artifacts in different languages (Pfeiffer and Wasowski, 2015). A genuine multilingual IDE must be able to handle multi-language meta information, cross-language refactoring (XLR), and inter-language containment. Thus, a system is mandated to integrate IDE compatibility across language boundaries (Lyons et al., 2017).

4.1.8. Source code analysis and manipulation

Integrated support to combine the source code analysis and manipulation domains is insufficient. Available tools either focus

on the analysis or transformation; therefore, combinations to accomplish both tasks are needed. The more the integration of analysis with transformation, the higher the investment cost (Klint et al., 2009). Information in large enterprise applications (EAs) is scattered across different constituents and corresponding relationships. Hence, source code analysis and manipulation of large EAs have become thought-provoking for researchers. The present object-oriented source code analysis approaches are unable to support EAs (Jiang et al., 2013). For instance, the heterogeneity of JEAs is tough to analyze and authenticate characteristics and architectural constraints (Perin et al., 2010). Therefore, an appropriate framework for consistent analysis of the source code is required where the quality of the heterogeneous applications is ensured, checked, and improved (Savić et al., 2014).

4.1.9. Source code reusability

Contemporary software models focus on cross-language applications which are complex. This makes the analysis, modifications, and reusability of these applications difficult (Binkley, 2007). Weak compatibility exists for analyzing the behavior and static authentication of such applications (Yazdanshenas and Moonen, 2011). These applications necessitate reusable multilingual constituents, linked through unique categories of files. All probabilistic cases of source code need identification for reusability across multilingual applications (Kargar et al., 2020). Finding pertinent source-code snippets required for software reusability is one more hurdle (Bui et al., 2019). Existing systems warrant a repository of appropriate code samples but such repositories do not exist. Further, a mechanism for source code recommendation is needed as most libraries lack API records on reusable components (Nguyen et al., 2012).

4.1.10. Change impact analysis

Propagation of change impact analysis (CIA) across multilingual artifacts is a crucial prerequisite for multilingual applications (MLAs) analysis (Jiang et al., 2013). Forecasting change propagation in the source code is a prime difficulty in the analysis of enterprise applications (EAs). To evaluate the impact of change, retain the history of code, and avoid inaccuracies, a language-independent approach is needed (Lehnert et al., 2013). Some of the present CIA approaches are compatible with one language only and are deficient in context-awareness (Kargar et al., 2020). Heterogeneous artifacts of different languages cannot be effectively analyzed employing such techniques (Lehnert et al., 2013).

4.1.11. Cross-language link detection

The XLLs help in understanding a software application, maintaining it, handling the errors, and refactoring it. Advanced software applications like JEAs, comprise interdependent artifacts in cross-languages. Semantic links are used to refer to them but relations between these artifacts are not structured to concealed dependencies (Savić et al., 2012). Present tools to analyze large codes are restricted to languages, making it difficult to detect and refactor XLLs among artifacts of a given heterogeneity (Perin et al., 2010). Minor changes in code have the potential of impacting the overall conduct of the software system. Moreover, a stock approach is not available to fuse multilingual artifacts (Mayer and Schroeder, 2014). This dearth of XLLs across MLAs impedes the stability and productivity of the system (Mayer and Schroeder, 2013a). XLLs are considered a major issue in the creation of multilingual development environments (MLDEs) as well.

4.2. RQ 2: What are the prevalent research techniques reported in the literature for cross-language link detection?

To address the given research query, a detailed examination of 76 shortlisted primary studies is conducted. Through this analysis, the following details are derived

- Tools used,
- Nature of the application,
- Meta-model,
- Language support, and
- Supporting model to evaluate the work done in this field.

Selected primary studies rely on three forms of analysis mechanisms, i.e. static, dynamic, and hybrid-semantic analyses. It is observed that out of 76 primary studies, 45 (59%) rely on static analysis of multilingual source code, while 21 (28%) use hybrid-semantic analysis mechanisms, and just 10 (13%) focus on dynamic analysis mechanisms, as shown in Fig. 10.

4.2.1. Static analysis mechanism

Project development teams feel under pressure because product updates are expected to be completed on time. Code and implementation requirements need to be met and mistakes are not a choice. Static analysis is thus better defined by the debugging process, where the source code is tested immediately before program execution. However, static analysis lacks an effective and efficient analysis of a program. It is quite difficult to discover all dependencies in a source code. The developer's intent is not clear and in some situations, faults can produce false positives and false negatives (Adams and Kear, 2003; Caracciolo et al., 2014; Rakić and Budimac, 2011). With growing diversity and heterogeneity in modern software, static analysis is becoming challenging and problematic (Pribela, 2012; Gosain and Sharma, 2015; Pfeiffer et al., 2014). On syntactic, conceptual, technical, and semantic grounds, there are various ways to analyze and transform intricate HAs. For example, the SSQSA framework developed by Pribela (2012), Rakić and Budimac (2013) and Rakić and Budimac (2011) is a suite of software-integrated tools for the static analysis of heterogeneous systems. The architecture in Rakić and Budimac (2013) is based on the intermediate representation of source code (eCST) that helps in identifying semantic links written in different languages (Strein et al., 2006). Another tool Eclipse MoDisco (Bruneliere et al., 2010), offers model-driven reverse engineering in Java applications, with compatibility for the extraction and recovery of models in legacy applications. These models are thus used in the analysis process. RASCAL (Aarssen et al., 2019; Klint et al., 2009) is an open-source meta-programming framework and workbench language for the analysis and transformation of source code.

Hecht et al. (2018), Mili et al. (2019) and Shatnawi et al. (2019) built an integrated DeJEE tool (Hecht et al., 2018; Mili et al., 2019; Shatnawi et al., 2019) that describes system components and their associated dependencies XLLs within Java enterprise edition (JEE). Similarly, source navigator (SN) (Moise and Wong, 2005) is a source code analysis tool, designed for large-scale enterprise systems. It shows the relationship between classes or their members, as well as the method of constructing call trees (Tomassetti et al., 2013b). Software networks are directed graphs (call trees) that describe the dependencies between entities in software applications. SNEIPL extracts a collection of networks representing the internal configuration of the software system at various abstraction levels. To extend this suite, the software structure change analyzer (SSCA) was developed later by Rakić et al. (2013), Rakić and Budimac (2011), Savić et al. (2013) and Kolek et al. (2013). In Kargar et al. (2020) a genetic algorithm is presented to achieve an integrated dependency

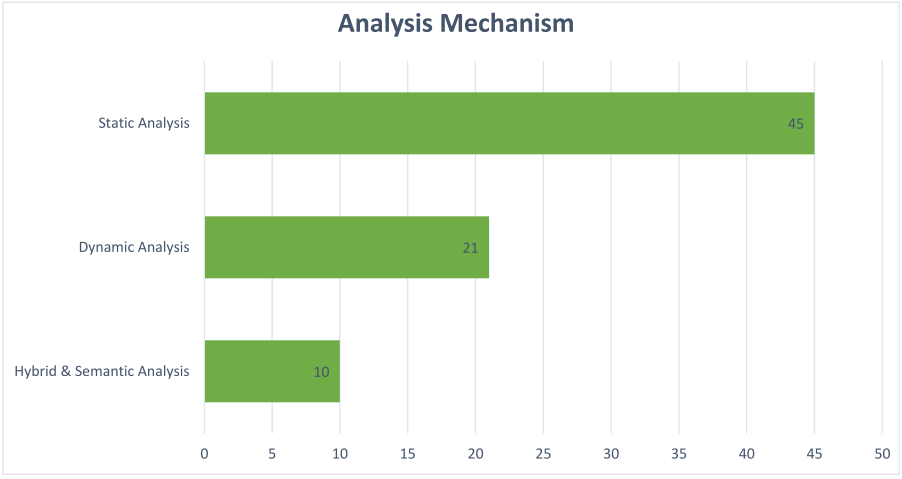


Fig. 10. Analysis mechanism used in primary studies.

graph from the source code for the modularization of multilingual applications. Automatic extraction of metrics from MLAs helps in the production of dependency graphs and software evolution (Marchetto et al., 2012). SMIILE (Savić et al., 2014; Gerlec et al., 2012; Savić et al., 2012) is a language-independent software metric tool that uses eCST as input for automation tools. A standardized solution for a multilingual development environment (MLDE) is provided by TexMo and Coral (Pfeiffer and Wasowski, 2015, 2012b). Pangea (Caracciolo et al., 2014) refers to the architecture, analysis, and extraction of intermediate results and static analysis of cross-language software corpora. Table A.4 in Appendix provides static analysis information including tool name, application type, language support, meta-model, and description.

The most commonly used tools are presented in Fig. 11 like MoDisco (Hecht et al., 2018; Mili et al., 2019), SMILE (Gerlec et al., 2012; Rakić and Budimac, 2013; Kolek et al., 2013; Pfeiffer and Wasowski, 2015), etc. In addition, the application type identified for each tool is subdivided into the following categories: J2EE application (Mushtaq et al., 2017a; Hecht et al., 2018), web applications (Tomassetti et al., 2014a), IDE (Strein et al., 2006; Mayer and Schroeder, 2013a), Android applications (Li et al., 2017), heterogeneous application (Klint et al., 2009) and prototype (Yazdanshenas and Moonen, 2011). Furthermore, the meta-model extensively used in static analysis mechanisms are procedural (Kolek et al., 2013; Hadjidj et al., 2008), object-oriented (OO) (Gerlec et al., 2012), and XML (Li et al., 2017). Such tools, which are available for static analysis, utilize a variety of supported languages including languages that are both general-purpose and domain-specific. The first types of languages include Java, C#, COBOL, C++, etc., while the latter include HTML, Groovy, XML, Rascal, Ruby, Spring, etc.

4.2.2. Dynamic analysis mechanism

Analyzing the complex nature of the software is important for software developers as it allows them to modify the scope of its applications. It includes program analysis during run time as well. Deployment of applications involves several dynamically connected libraries which makes the static analysis inaccurate (Nguyen et al., 2012). Additionally, the growing use of OO languages for applications, particularly Java, has led to the use of run-time features such as dynamic linking, threads, polymorphism, etc. Static analysis is considered unsuccessful in this sort of complex environment. Dynamic analysis is thus becoming increasingly popular for program analysis (Gosain and Sharma, 2015). Various tools for dynamic analysis are presented for the evaluation of multilingual applications (Ajax-based systems, web systems, and enterprise applications (EAs)). A language integration approach with complete platform support using a meta-programming system (MPS) and the Eclipse platform is introduced in Tomassetti et al. (2013b), as shown in Table A.5. The focus is placed on enterprise Java beans (EJB), interceptors, and dynamic analysis. Client-side code is also dynamically created from server-side code in a complex web application. However, current tools for the maintenance of code like automated renaming support work only with program entities for a single language on either the server-side or the client-side. Other tools have already been introduced like BabelRef. (Nguyen et al., 2012) that automatically describes and renames client-side system components and their references. Only the call graph of client-side code inside the server-side code can be approximated and the target of the call can vary among various server-side code executions. Given the various execution path in the code, such values cannot be resolved before the code is run, and in fact, they can be non-deterministic (Binkley, 2007). To construct these call graphs for embedded client-side code in complex WAs, variability-aware, and symbolic execution parsing techniques are used in Nguyen et al. (2014).

CHECKCAMP (Grimmer et al., 2015) is an automatic method for identifying inconsistencies in the same native framework implemented for Android and iOS platforms. This would help software developers in accurately extracting and analyzing mobile application models from various platforms. TruffleVM (Shatnawi et al., 2018b) is a multilingual runtime framework that permits the seamless composition of various language implementations. Language implementations help in converting source code into an intermediate representation (IR) that is implemented on top of a shared runtime environment. Researchers introduced the ideas based on the Truffle method and its guest language implementation like Ruby, C, and JavaScript. Call trees are built-in methods (Shatnawi et al., 2018b) which reflect the hierarchical relationships between the API methods. A clustering algorithm based on a graph is used to segment the graphs into smaller graphs that display the dependencies between the API methods. Successful compilation of programming language to Java bytecode is achieved using Lässig prototype in Pfeiffer et al. (2014). The approach specifies the standardized traceability of the languages installed on a virtual machine. Lässig is a simple, lightweight, and low-cost approach for incorporating traceability into modeling frameworks. Traceability changes are handled by MaTraCa in Lozano et al. (2016) which is an eclipse plugin that establishes horizontal links in heterogeneous applications. It maintains traceability links between components with the same

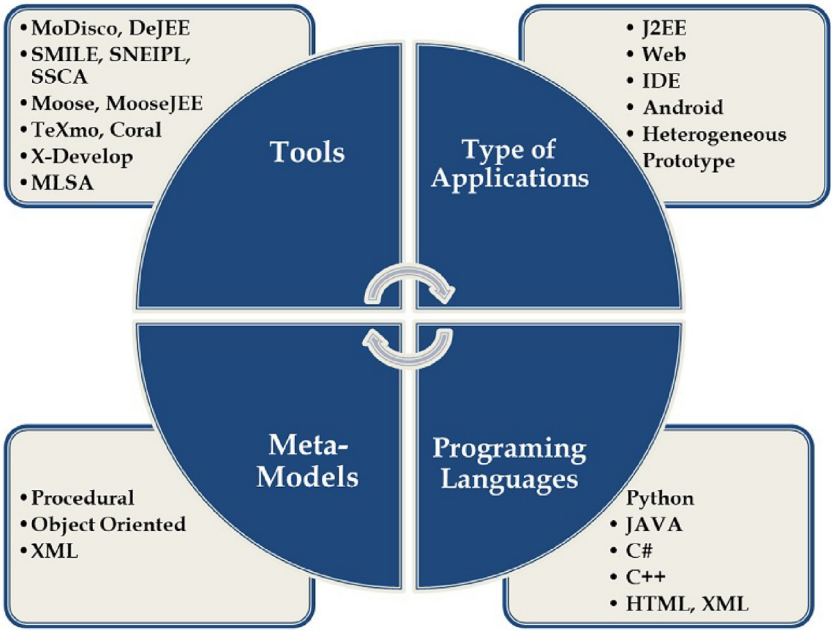


Fig. 11. Static analysis tools and types of applications.

level of abstraction, i.e. source code, configuration files, and HTML forms.

ReAjax (Marchetto et al., 2012) is a reverse engineering tool capable of creating GUI-based state models from Ajax applications. ReAjax (Marchetto et al., 2012) performs detailed analyses and utilizes execution traces to construct a finite state machine of the intended application interface. These verify that semi-automatically obtained GUI-based state models are identical to those acquired manually. Thus, they can be used for program understanding. In recent years, dynamic modeling has garnered considerable significance because of its potential to predict the run-time behavior of the program. A detailed account of the tools used for dynamic analysis is given in Table A.5. Efforts are made to highlight the dynamic analysis information provided in the form of the tool name, application type, meta-model, language support, and description model.

Tools used for dynamic analysis are presented in Fig. 12 including MPS (Tomassetti et al., 2013b), BabelReF (Nguyen et al., 2012), Var-DOM (Nguyen et al., 2014), CHECKCAMP (Joorabchi et al., 2015), TruffleVM (Grimmer et al., 2015), Lässig (Pfeiffer et al., 2014), MaTraca (Lozano et al., 2016) and ReAjax (Marchetto et al., 2012). Moreover, the identified application type for each tool is sub-divided into the following categories: web applications (Nguyen et al., 2012; Marchetto et al., 2012), IDE (Nguyen et al., 2014), Android/IOS application (Joorabchi et al., 2015), OO application (Shatnawi et al., 2018b) and virtual machine (Pfeiffer et al., 2014). Furthermore, the meta-models used in the dynamic analysis mechanism are procedural (Grimmer et al., 2015), OO (Gosain and Sharma, 2015; Joorabchi et al., 2015), and XML (Nguyen et al., 2012, 2014). The dynamic analysis mechanism tools use a variety of supportive languages that include Java, XML, C, C++, HTML, PHP, Javascript, and Ruby. The biggest challenge with multilingual systems is the model's co-evolution which includes cross-language artifacts. By modifying one model the associated models have to change accordingly. Identifying the affected artifacts in the form of trace links is quite challenging. Current programming languages do not help traceability. Automatic pursuing of entity relationships is needed to help the co-evolution of multilingual software applications (Pfeiffer et al., 2014). The use of traceability links has been ignored while the complexities of the existing implementation are handled (Lozano et al., 2016).

4.2.3. Hybrid analysis mechanism

Conventional testing approaches are not adequate for detecting concealed errors in the control system software. If overlooked, these may lead to run-time failures which are extremely costly to amend. To overcome this limitation of traditional approaches, a hybrid analysis method is used to determine possible causes of such errors at compile-time. It ensures safe execution and easy detection. To define run-time errors, the control system implementations are initially interpreted to construct a set of abstract syntax trees (ASTs) which are utilized to build a data flow graph of the program. Consequently, the number of ASTs from source code analyzers can become too large to understand or analyze within a fair time. Some analyses should be detailed to ensure accuracy, precision, and recall of results (Nair et al., 2015). The Shatnawi et al. (2017) outlined the main challenges in evaluating dependencies across core JEE technologies (Servlets, JSPs, JSFs, and EJBs) and addressed them.

The DeJEE tool identifies 70.5% more reliable dependencies and program components that are not found throughout the implementation process. Software networks (Hecht et al., 2018) are directed graphs that can be used to analyze the intricacy and development of high-magnitude software applications and measure software architecture-related metrics. Software network abstraction contributes to the first phase of reverse engineering tasks.

A hybrid algorithm is employed in Nair et al. (2015) to track the flow control graph for constraint types, properties of liveliness, and reach. Many of these methods transform control code into an intermediate version, which is input into a model checker to validate the specifications. The industrial suitability of these methods is restricted due to state explosion issues and a lack of adequate mapping between the tools. A lightweight and flexible multilingual parsing method is provided in Bogar et al. (2018) that utilizes island grammar for concurrent parsing of MLAs. This method can be applied to JSP and related dynamic web content paradigms only. The research, visualization, and creation of MLA software metrics can be made economical only by utilizing automated development methods that help in the creation of dependency graphs, software evolution analysis, and source code metrics (Terceiro et al., 2010). The Analizo Toolkit (Rakić and

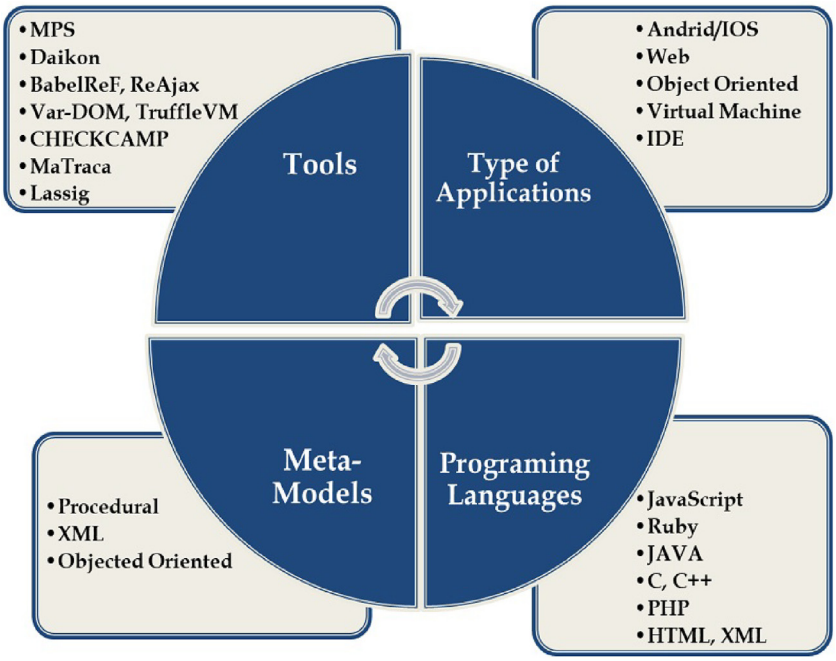


Fig. 12. Dynamic analysis tools and types of applications.

Budimac, 2011) offers tools for evaluating and visualizing MLAs in source code metrics, evolution analysis, and correlation graphs. However, being based on the Doxygen parser, it is unable to fully decode the source code. It is incompatible with web applications, so the forthcoming development of the Analizo Toolkit is a web-focused edition. The work (Tomassetti et al., 2014b) defines the CrossLanguageSpotter framework presently enabled by XML, Ruby, JavaScript, Markup, Property, and Java. The models are developed from the source code (Abstract Syntax Trees (AST)) to identify cross-language connections in polyglot frameworks. The library is designed for modular architecture and integration with existing IDEs. Clustering approaches for MLA semantic analysis use related feature categories to approximate the gap between source code components. The required techniques will not yield high-quality outcomes in the absence of sufficient input.

Table A.6 given in Appendix, displays details regarding the hybrid semantic analysis mechanism in the form of tool name, application type, meta-model, language compatibility, and description model. Several hybrid methodologies have been implemented to support static and dynamic analysis processes. Such tools perform semantic (Marinescu and Jurca, 2006; Tomassetti et al., 2013a, 2014b), static and dynamic analysis (Shatnawi et al., 2017; Savić et al., 2014; Rakić and Budimac, 2011; Savić et al., 2013; Grimmer et al., 2018), and static, semantic, and dynamic analysis (Kontogiannis et al., 2006; Terceiro et al., 2010; Schink, 2013). In addition, the most widely used application type for each tool is subdivided into the following categories, and famous of them are presented in Fig. 13: J2EE (Shatnawi et al., 2017; Kolek et al., 2013), web applications (Linos et al., 2007; Tomassetti et al., 2014b), IDE (Bogar et al., 2018; Tomassetti et al., 2013a), virtual machine (Grimmer et al., 2018) and prototype (Nair et al., 2015). In comparison, procedural (Rakić and Budimac, 2011; Jinan et al., 2017), OO (Binkley, 2007; Marinescu and Jurca, 2006; Tomassetti et al., 2014b), and XML (Mayer, 2017; Pribela, 2012) are the widely used meta-model throughout the hybrid semantic analysis mechanism. Both, hybrid and semantic research systems support a range of languages including those which are general-purpose and domain-specific. The first category includes Java, JavaScript, PHP, COBOL, C++, C, C#, etc. while the latter includes HTML, XML, SQL, JSON, UML, YAML, and Groovy.

4.3. RQ 3: What are the limitations of prevalent research techniques and tools used for cross-language link detection?

In order to answer the RQ3, a thorough review of 76 primary studies is performed and subdivided into three forms of analyses process i.e. static, dynamic, and hybrid semantic analyses. Each limitation is analyzed in the following dimensions: implementation plugin, metric extracted, case study, support visualization, and support documentation to evaluate the extent of research work in cross-language link detection for multilingual applications.

4.3.1. Limitation of static analysis

In static analysis, it has been observed that several limitations need to be addressed as shown in Fig. 14. In Mili et al. (2019), re-engineering of service-oriented JEE applications focuses on the program dependency graph. The research has its limitations as it does not provide any support for metrics calculation, graphical user interface (GUI), or documentation. In Gerlec et al. (2012), eCST dependent SMIILE tool is introduced for computing metrics (Savić et al., 2012; Rakić and Budimac, 2013; Kolek et al., 2013). This has language independency as a short-term target. However, as a long-term objective, a smart software analytics tool is needed, which can suggest the developers improve the source code. In Pfeiffer and Wasowski (2015), TexMo and Coral tools lack automated identification of cross-language interactions and require enhancement for efficient language representation. The platform has minimal flexibility in cross-language representation and the multilingual development environment (MLDE) needs expansion. The drawback with a domain-specific language RASCAL (Aarssen et al., 2019; Klint et al., 2009) is that its performance is quite limited for heterogeneous applications and normally supports OO languages. A systematic method for the study and refactoring of multilingual source code applications is provided in Strein et al. (2006). This method incorporates semantic relationships. The tool support is restricted to a subset of languages.

Cross-language relations are studied in Java and domain-specific languages (DSLs) in Mayer and Schroeder (2013b) and

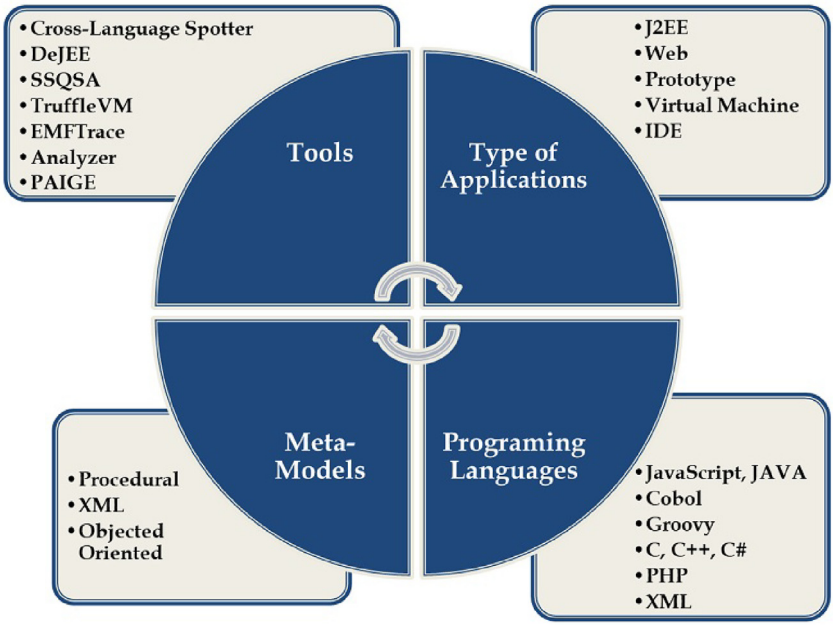


Fig. 13. Hybrid analysis tools and types of applications.

Mayer and Schroeder (2013a). A multilingual framework for the exploration, maintenance, and refactoring of XLLs in Java and DSLs is introduced in Mayer and Schroeder (2013b). It includes platform support for XLLs in DSLs and integration with legacy applications. The linking and refactoring of cross-language objects is another problem in MLAs (Mayer and Schroeder, 2014). Bilateral neural networks (Bi-NN) (Bui et al., 2019) are analyzed only in Java and C++ programming languages. The framework should be applied to cross-language paradigms, such as OO versus functional, to improve its applicability. In Strein et al. (2006), a language-independent meta-modeling method X-DEVELOP is introduced that facilitates analyzing and refactoring several languages. However, the method has a poor understanding of multilingual constituents. In addition, X-DEVELOP needs compatibility for dynamic, generalization, and low-level languages, as well as the number of supporting languages (Caracciolo et al., 2014). A simplified and extendable approach for identifying multilingual dependencies is proposed in Polychniatis et al. (2013). The current program, therefore, needs categorization according to language and scale. The system's response has to be looked at a component level while its performance requires improvement.

In Bruneliere et al. (2010), MLSA architecture is proposed for the construction of call graph dependencies to perform lightweight static analysis of large multilingual software applications. The research is limited because it focuses on the specific problem of generating a call graph and developing a detailed system diagram. Moreover, it does not provide any support for metrics calculation, GUI, or documentation. The main difficulty in multilingual source code implementations is to create a parser for every language used in the source code (Janes et al., 2013). The study suggests using AST, as an internal representation to efficiently store cross-language code. The parser is extended only to support C/C++ languages. Pangea (Caracciolo et al., 2014) is a workbench with a minimal data repository for dynamically examining multilingual software corpora and object-oriented languages. In Yazdanshenas and Moonen (2011), the knowledge discovery meta-model (KDM), a language independent of the meta-model, is built for reverse engineering software applications. The method is tested by designing two prototype tools. However, a GUI to examine information flow is not provided. The current tool only provides a few updates and is inadequate for analyzing multilingual applications on a wide scale. The study focuses on only analysis or transformation but does not support additional design recovery techniques.

Reverse engineering is an iterative process that requires continuous reporting (documentation), as new requirements are incorporated into the system (Mayer and Schroeder, 2014; Kontogiannis et al., 2006). Current approaches for cross-language link detection are compiler-dependent and only compatible with monolingual comparison (Moise and Wong, 2005). The main difficulty in evaluating MLAs is creating a different parser for each language used in a software (Terceiro et al., 2010). For instance, E-commerce systems include intertwined multiple source code syntax. An efficient multilingual parser is necessary to manage the multilingual source code simultaneously (Boughanmi, 2010). Modern web applications contain dynamic and multilingual pages jumbled together. Owing to the complex nature of HTML code and multiple language interactions in an application, static analysis of WAs is complicated (Sultana et al., 2016). Tools and techniques reported in existing literature for software development and management have many limitations. To name a few, minimal support for programming languages, inconsistent and poor metrics usage and/or measurement strategies, etc.

4.3.2. Limitation of dynamic analysis

Fig. 14 shows the limitations of prevalent research techniques in MLSAs for dynamic analysis mechanisms. The integration of multiple languages called meta-programming systems (MPS) on the Eclipse project is provided in Tomassetti et al. (2013b). This methodology requires an empirical appraisal and additional enhancements for reliability classification in cross-language links. Evaluation of several tools and techniques for the dynamic analysis of code is presented in Gosain and Sharma (2015). However, the study has restricted uses as it lacks proper visualization, and testing metrics are not provided. In Jenkins and Kirk (2007), a multilingual virtual machine called Truffle VM is presented which is an analysis mechanism that executes at the top level of the same virtual machine. The tool improves the usability, efficiency, and versatility of multilingual applications, However, this methodology has the following drawbacks

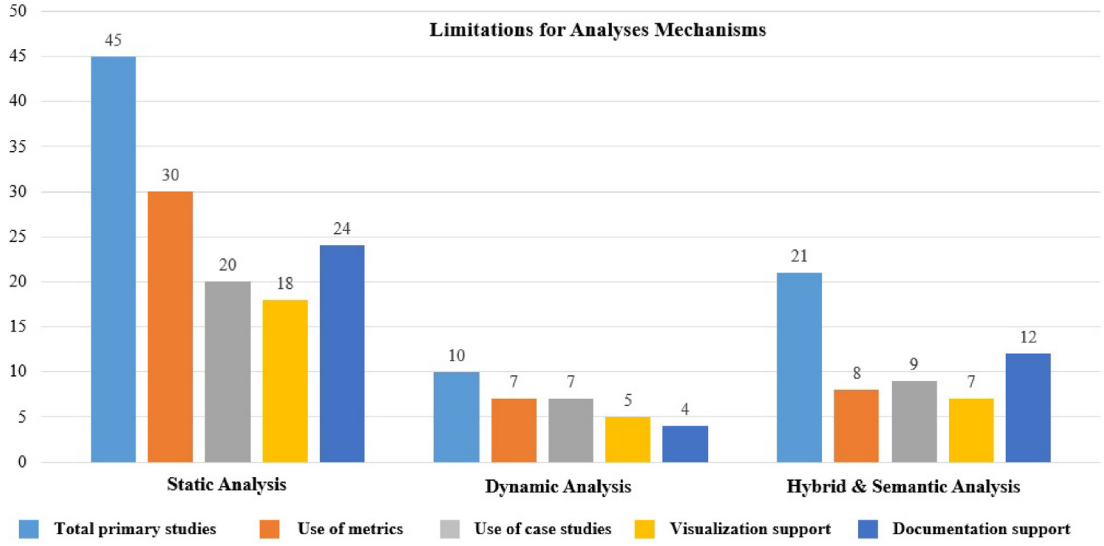


Fig. 14. Limitations for analysis mechanisms.

1. It still needs to support the visualization aspect,
2. No sufficient support documentation is offered,
3. The method is not evaluated in any case study to determine, whether it produces reliable results or not.

A popular traceability modeling tool Lässig is presented in Pfeiffer et al. (2014) that defines traceability across multilingual artifices. This method handles model-to-text transformations and tags affected artifacts through trace links. Yet this method has minimal support for heterogeneous applications. Dynamic analysis and reverse engineering are carried out using execution traces in Ajax applications. In Marchetto et al. (2012), ReAjax reverse engineering tool uses a similar methodology and is capable of generating practical state-based GUI templates. The proposed method may be used to produce test cases, so it is important to check functionality by implementing a case study. The analysis of MLAs is complicated and challenging due to their diverse and dynamic nature (Bruneliere et al., 2010).

Understanding multilingual interaction is challenging due to a wide variety of artifacts lacking integration and tool compatibility (Tomassetti et al., 2014b). Current methods can only derive and establish views for single system execution. Such methods are not capable of retrieving application-level interface actions for dynamic heterogeneous applications (Pfeiffer and Wasowski, 2012a). Dynamic web applications use mixed source code from different programming languages like PHP, HTML, CSS, and JavaScript. This makes them complicated and challenging to evaluate and handle clones (Grimmer et al., 2015). Existing cross-language systems suffer from complicated architectures, limited functionality, or bad performance.

4.3.3. Limitation of hybrid analysis

Fig. 14 provides the constraints of hybrid semantic analysis mechanisms for the prevalent research techniques and tools in MLSAs. In Shatnawi et al. (2017), researchers have improved traditional analysis strategies to obtain the internal representation of links between JEE program elements. However, the proposed approach lacks identifying run-time dependencies. In Mayer (2017), the authors presented a taxonomy of the mechanisms of encoding XLLs in popular open-source frameworks from the perspective of developers. The research is limited by the fact that it does not target all the open-source frameworks in XLL mechanisms. Moreover, the developed taxonomy has a generalizability issue. A tool for control system software is proposed which synthesizes data flow analysis and user-defined syntactic checks. In Nair et al. (2015), a tool is presented that uses a hybrid analysis framework to extend across different application domains. The study (Jinan et al., 2017) briefly assesses the research in vulnerability enumeration, detection methods, and taxonomy models using intelligent data processing and analysis. The limitation of the research is that the technique is not implemented for any case study and evaluation is inappropriate. Moreover, it lacks proper visualization and documentation of the results.

A meta-model is proposed in Marinescu and Jurca (2006) for carrying out reverse engineering on enterprise applications. For scalability validation of the proposed approach, it is necessary to avoid manual inspection and the use of smaller case studies. Dependency analysis is carried out to identify dependencies among all types of artifacts including requirements, static and dynamic UML models, test cases, and source code. A more methodical investigation of dependency types is mandated to extend the approach for fine-grained couplings with dynamic UML models (Lehnert et al., 2013). The CIA is applied to identify the impact of change on a multilingual software product line in Angerer (2014). But the technique does not provide any graphical visualization of changes made. Similarly, no metrics are extracted to measure the extent of evolution (Lehnert et al., 2013).

Evolution and maintenance of software, multi-perspective modeling, change impact analysis, reengineering, and horizontal traceability are the prominent domains in hybrid analysis. The hybrid analysis uses language features such as reflection and dynamic class loading, at the link or run time. Tools and techniques based on hybrid analysis combine results from different times, require information from edit, compile, and link, run time, and continue to include a combination of multiple views of an MLA such as structure, behavior, and run-time snapshots. Such methods that may enable reliable analysis have specific characteristics inherited from the existing joint analysis. They all are linguistically independent, scalable, and adaptable. The methodologies presented in the literature indicate that a straightforward hybrid approach increases interoperability/portability issues, and is improbable over fully dynamic detection support. Even when compared with a fully static detection, the hybrid dynamic/static analysis mechanism fail to show consistent improvement.

4.4. RQ 4: How to develop a conceptual schema and taxonomy that covers all essential components of cross-language analysis?

Extracting XLLs from multilingual source code entities is the prerequisite to performing XLA. The XLA tools usually outline the adopted methodology in this regard. Even though each methodology has its positives, the negatives and limitations may only be dealt with by other methodologies. Existing works use evaluation and comparison with other tools and techniques for source code analysis but the literature lacks a comprehensive guideline. To assist the researchers and practitioners in the above-mentioned goal, a general schema with definitions of cross-language linking and associated concepts is furnished. The contribution of this research study is to offer SLR on XLLs and a comprehensive taxonomy that incorporates all the methods for XLL detection in multilingual source code.

4.4.1. Cross-language analysis

XLA is the information extraction of all potential executions from multilingual codebases. These include dynamic web pages containing HTML and scripts, Microsoft active server pages (ASP) which include server and client-side code in one file, or Java/Cobol programs embedded with SQL statements (Bogar et al., 2018).

Preliminaries

Defining some terms here is necessary which are later used in the following sections of this work.

AST: (Abstract Syntax Trees)

It is the code representation that accurately encodes the syntax of a program (Bui et al., 2019).

CST: (Concrete Syntax Tree)

The representation illustrates how to derive a programming language construct, under the context-free grammar of the language (Savić et al., 2014).

eCST: (Enriched Concrete Syntax Tree)

It is a specific blend of classical AST and CST enriched with universal nodes. Universal nodes can be considered as uniform imaginary nodes that describe elements of language semantics, independent of language and its syntax (Rakić and Budimac, 2013).

Call Dependency Graph

It contains nodes, representing the functions of a program and, edges between nodes, only if there exists at least one function call between the corresponding functions (Yazdanshenas and Moonen, 2011).

Language-Dependent Meta-Model

A language adapter requires to be implemented manually to support each language by the meta-model (Savić et al., 2014).

Language Independent Meta-Model

It offers APIs to describe the problems and solutions of uncovering invocation relationships and discover dependencies through dependency call graph (Shatnawi et al., 2017, 2019; Perin et al., 2010).

General Schema of Cross-Language Analysis

To understand the cross-language linking mechanism, it is necessary to identify the types of artifacts available along with their relations. Artifacts, in most languages, are interconnected in a tree- or graph-like structure. Navigation of such a structure to identify the linked artifacts is important to retrieve individual instances for actual link resolution and have a link back into the source code for annotation, navigation, and refactoring. Cross-language links are generally dealt with at three levels:

- Source code.
- Meta-models that are language dependent.
- Meta-models that are language independent.

Parsing is required to use source code; for instance, abstract syntax trees (Savić et al., 2014) and concrete syntax trees (Rakić and Budimac, 2013). However, adapting this approach is difficult to support code relationships that are more complicated like method calls. Moreover, the link descriptions are required to handle all convolutions of a language. The second option is to use a greater abstract which is a language-dependent representation of artifacts. This methodology mandates one adapter per language to manage the classes and instances of artifacts. Although it hides the complex operations from link specifications in the adapter, it does require the construction of an adapter for each language. Finally, choosing a common meta-model helps in refactoring rules and writing analysis in a language-agnostic way. Nevertheless, one limitation of this approach is that it disregards the concepts of classes, tables, etc. available in each language. Further, it requires changing the common model for every new language added (Savić et al., 2014).

Real-life software systems are signified by complex XLLs. These links can be modeled in terms of a unified standard representation (Tree/Meta-model) highlighting dependencies between the software entities. Call dependency graphs, representing software systems, are essential to extract for the following reasons:

- To comprehend the intricacy of dependency structures in software systems.
- To calculate metrics pertinent to the design of the software.
- To extract software networks for software comprehension.
- To extract fact bases for reverse engineering.
- To recover system architecture from the source code.

The XLA schema consists of two parts namely the XLA front-end (also called standard representation generator) and the set of XLA back-end cross-language analysis methods. The main idea is that given a program or a code snippet, it is first parsed into AST or a unified meta-model, and then using XLA dependency relations (XLLs) are extracted to build a call dependency graph. The schema of XLA is presented in Fig. 15. Call dependency graphs can be used for maintenance and evolution purposes.

Step1: Generation of Unified Standard Representation

Standard representation generator uses parsers to produce an internal representation of multilingual source code which is inserted as input. The standard representations (AST, unified meta-model) are created by all semantics present inside the vocabulary of multilingual source code. These are created based on the resemblance between the names of identifiers present in the source code.

Step2: Employing Analysis Mechanisms

In the second step, analysis methods are employed on the unified standard representation (AST/unified Meta-model) to produce the call dependency graph which highlights the relationships among artifacts present in the source code.

Step3: Construction of Dependency Graphs

Dependency graphs denote the connection between software artifacts. They cover structural and non-structural features of the source code and are independent of the syntax of the programming language. The construction of graphs is supported by automated tools, which represent program elements and their associated dependencies within and between entities, as one dependency called graph (Nguyen et al., 2014).

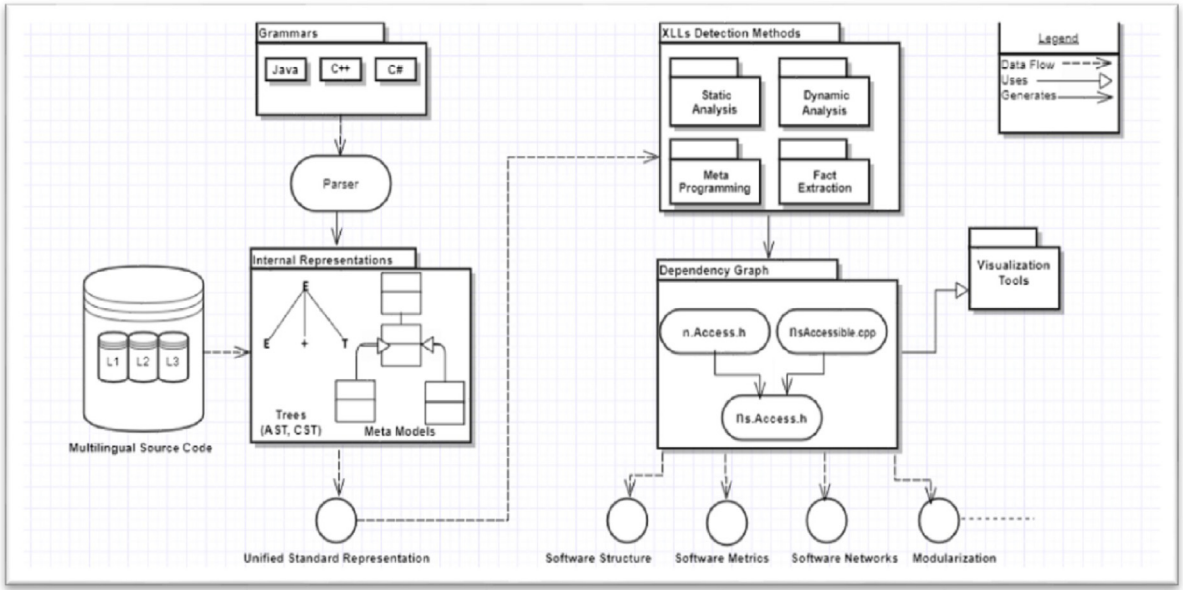


Fig. 15. General schema of cross-language analysis.

Step4: Reverse Engineering Activities

The obtained call graphs are the essential component for further reverse engineering activities. They can be stored in a repository for the extraction of dependency networks to analyze the design complexity of software systems. Or networks can be considered as fact-bases to calculate metrics associated with the design of software (Savić et al., 2013). If the software structure deviates from the original architecture, modularization techniques are applied. It partitions a software system into a variety of meaningful and understandable modules to facilitate the comprehension and development processes (Kargar et al., 2020).

4.4.2. Taxonomy

The purpose of this study is to provide a thorough literature review and the creation of a taxonomy (XLA) to help the critical sections of the cross-language analysis such as the parser, the unified representation, and the representation analysis. The taxonomy is developed to keep the schema and SLR results consistent.

Components of XLA

Three elements of the XLA are the parser and the internal representation, and their representation analysis are shown in Fig. 16.

The Parser

The first phase of analyzing the source code is parsing the source code into one or more internal representations, i.e. parse trees or abstract syntax trees. There are standardized lightweight parsers (Doxygen) (Terceiro et al., 2010) which transform the evaluated source code into a tree of entries. Such trees are created through lexical analysis and gain linguistic independence using a multilingual fuzzy parser. However, there are drawbacks to the hierarchical fuzzy parsing approach which limits the extensibility of a specific language class (e.g. Pascal-like or C-like languages). Many methods follow the Doxygen path and compose their parser e.g., Kienle and Müller (2010), PAIGE (Bogar et al., 2018), resulting in significantly worse performance than utilizing a complete compiler.

Compared to Doxygen, ANTLR (ANTRL, 2021) is a parser generator that can create lexers, parsers, tree parsers, and combined lexer parsers using LL(*). SNEIPL (Savić et al., 2014) uses a much richer intermediate representation of the source code provided by ANTLR that does not limit the extensibility to a particular language family. Bytecode parsers explore a source code file system and collect specific dependencies. Dependency Finder (Savić et al., 2014) finds bytecode Java class dependencies and reads all sorts of compressed Java data: JAR data, zip files, or object files. When developers have to process just a subset of the specific grammar, lighter-weight strategies can be implemented. For example, Island Grammars (Lyons et al., 2017) is used for evaluating all the language components together in a single parse tree. A creative approach to PAIGE (Bogar et al., 2018) is published as an Island Grammar in this context (Moonen, 2001), but the method is not bound to a specific language.

The Internal Representation

The second component of a multilingual source code analysis is internal representation (Binkley, 2007). Each source code analysis method may choose the correct internal representation (i.e. trees, graphs, meta-models) to reflect cross-language connections in a multilingual code representation. The collection of internal representation relies on the underlying technologies and the aspired methodology of program analysis. Therefore, various conceptual representations have different structures depending on the point of view of the inventors as to what is right lexically, syntax, and semantics. Consequently, the volume of XLLs retrieved by source code analyzers such as compact lightweight parsers (Doxygen) can become too large to interpret or evaluate in a fair time (Terceiro et al., 2010). On the other side, some research requires information to maintain good accuracy and recall for analysis results. Some internal representations are generated explicitly by the parser such as the control-flow graph, while others need the results of previous analysis such as dependency graphs requiring specific analysis points. There are many internal representations similar to different source code analyses. Control-flow graph (CFG), call graph, and abstract syntax tree (Kitchenham et al., 1999) are all popular examples of such representations.

Meta-modeling strategies are another way to represent the source code effectively. These techniques involve source code module parsing using the grammar and syntax of a language. Static analysis is performed using a single snapshot of the system. Moose, Eclipse MoDisco1, and GenDeMoG are representative projects that benefit from meta-modeling techniques (Polychniatis et al., 2013; Bruneliere et al., 2010). Different technologies

are developed for specific purposes. Therefore, they adopt various meta-models and include different structures depending on the opinions regarding lexical, syntax, and semantics. As servlets are built using Java programming, they adopt an OO meta-model where the system components are identified with objects, methods, and attributes, and the software dependencies are primarily focused on function invocations, attribute access, object instantiations, tribute access, and object instantiations. On the other hand, JSPs and JSF implement an XML-driven meta-model in such a way that the software components are built on XML tags that describe the attributes and dependencies. This can be applied to other technologies as well. There is also a need for a unifying model description that can describe all JEE technologies, taking into consideration the software elements and dependencies. To address the complexity of the variety of meta-models, an intermediate language-independent meta-model is chosen. Such a meta-model can include a standard framework for consistency that integrates all JEE technologies into their device elements and relationships. Examples of related existing language-independent meta-models include KDM (Polychniatis et al., 2013), FAMIX (Shatnawi et al., 2019; Gerlec et al., 2012), etc. **The Analysis of Representation**

The actual analysis is the third and final aspect of a multilingual analysis of the source code. Analyses can be categorized in the following dimensions:

- Static versus dynamic.
- Metaprogramming.
- Fact extraction.
- Token-based extraction.
- Pattern matching.
- Use of Universal Language.
- Use of Algorithm.
- Symbolic execution

Static Versus Dynamic Analysis

The static analysis does not accommodate system input; thus, the result applies to all program executions (Shatnawi et al., 2019; Nair et al., 2015). In comparison, the dynamic approach takes into consideration the feedback of the program (usually a single entry). This allows for better precision. However, the tests are still assumed to be accurate for the specific input (Nguyen et al., 2012; Pang et al., 2018).

Metaprogramming

Meta-programming consists of matching, transforming, and analyzing syntax trees. Many meta-programming systems process abstract syntax trees, but this involves extensive knowledge of the data form framework representing the abstract syntax. As a result, meta-programming is error-prone, and meta-programs are not resilient to the evolution of the structure of such ASTs. The study (Aarssen et al., 2019) developed a methodology to expand meta-programming systems based on external black box parsers with pluggable concrete syntax patterns.

Fact Extraction

Fact extraction is an automated method that analyzes the source code to classify program entities and their shared connections. This method results in an abstract representation (model) of the extracted information. The primary goal of reverse engineering is to define the components and relationships among them to establish system representation at a higher abstraction level (Shatnawi et al., 2019). One approach to identifying cross-language links in heterogeneous software systems is to determine fact bases for fact extraction.

Token-Based Approach

An easy and efficient algorithm for detecting cross-language dependencies is a token-based method to substitute the conventional, inflexible code parsing methods. An efficient algorithm focuses on matching similar tokens between two compatible source modules. Such comparison is based on statistical filtering to measure all the frequencies of distinct tokens per language inside the framework. So, a benchmark is needed to classify systems according to their size and languages. These are required to evaluate the component-level system behavior. In addition, adding additional statistical filtering can be carried out for accuracy improvement (Polychniatis et al., 2013).

Pattern Matching

Knowing XLLs and maintaining their stability throughout the life cycle of software development is a significant issue of efficiency. However, little effort has been done on understanding the characteristics of these links. Such links can also be seen as specifications for designing and developing a language linking and tooling infrastructure. Mayer and Schroeder (2013a) defined popular cross-language linking trends with DSLs in the Java frameworks domain.

Use of Intermediate Language

The modern software paradigm is shifting towards developing cross-language SW applications to meet the changing demands of the users and domains. It increases the overall complexity, making it hard to analyze, modify, and reuse. Therefore an intermediate format is required to determine the quality and complexity of these cross-language software applications. The study (Linos et al., 2007) focuses on performing a complexity analysis of multilingual applications in intermediate language format (Microsoft Intermediate Language (MSIL)), processable by any language. This method helps create language-independent syntax parsers to construct multilingual program applications.

RASCAL is a language that incorporates different aspects of analysis and transformation. It offers an easy, stable, and responsive development environment that eliminates integration overheads and typically promotes OO languages. RASCAL is a domain-specific language that conducts a source code analysis of diverse heterogeneous applications at a high degree of logical, syntactic integration, semantic, and technical grounds (Klint et al., 2009).

Symbolic Execution

VarDOM is a representation that models all possible DOMs variations in the developed client code. The authors used symbolic execution and variability-aware parsing techniques on a provided PHP program to create VarDOM. The authors then introduced HTML/CSS analysis and re-encoded the issue of reusing WALA to construct JS call graphs. An empirical assessment of real-world systems has demonstrated that the proposed methodology can achieve high precision and efficiency (Nguyen et al., 2014). BabelRef is an innovative method for the recognition and renaming of cross-language, client-side HTML/JS software entities and their connections in a PHP-based web application. The concept is to execute each PHP page symbolically and to display all possible created client pages in a single tree-based structure called D-model (Nguyen et al., 2012). Its contributions are summarized as follows

- Developed a general schema along with an MLSCA pattern from the perspective of the cross-language links.
- Performed an SLR to determine the required features and developed a detailed taxonomy named XLA.

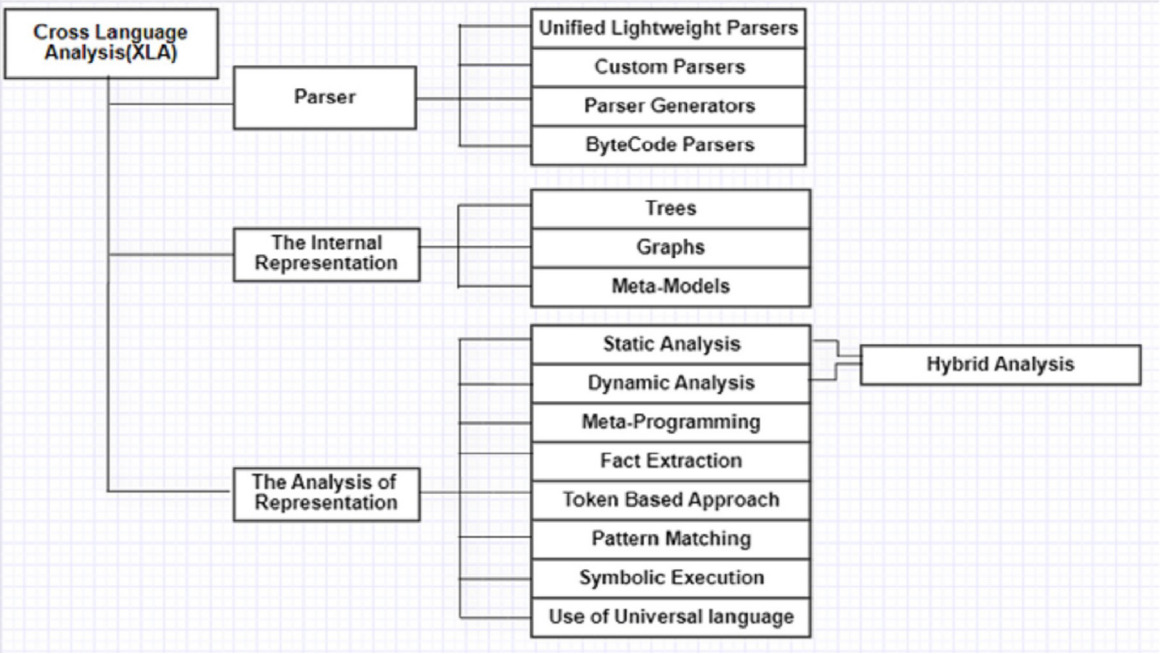


Fig. 16. Taxonomy of cross-language analysis.

4.5. RQ 5: Which challenges are faced by the research community and what are future contributions in recognition of cross-language links

Every research effort contains limitations and some guidelines for the future to go the extra mile to bring betterment. Handicaps and prospective work associated with various research approaches are discussed in this section. Multiple ongoing and future directions are presented here for the analysis of cross-language dependencies in multilingual applications (Binkley, 2007).

4.5.1. Java enterprise applications

Reverse engineering and quality assurance techniques which are currently in practice do not analyze JEAs as they focus on specific languages and components. Advanced software analysis tools and techniques lack in capturing the essential aspects of JEAs because of their heterogeneity. A little effort has been put into providing instruments that support enterprise application analysis (Shatnawi et al., 2017).

Future Directions

Improvement of generalization from JEE applications to multilingual applications and to enhance the applicability of the dependency call graph is needed. Extension of automated tools is required to include all dependencies in JEE applications. Experimentation with a bigger magnitude of case studies is also required. The ultimate objective is building a generic and extendable representation of JEAs to support effective analysis of the same (Shatnawi et al., 2019).

4.5.2. Dynamic web applications

When dealing with dynamic web applications, programmers often rely on IDEs for the development and maintenance of software. Currently, tool support to provide editor services, such as auto-completion, syntax highlighting, and 'jump to declaration' is restricted either to the server-side code or hand-written or generated client-side code (Nguyen et al., 2014).

Future Directions

Future research in dynamic web applications should focus on improving middleware support for seamless communication between client-side and server-side code. Efforts should be made to develop precise models for handling dynamic languages like JavaScript effectively. Additionally, researchers should explore ways to support low-level languages and enhance security measures to protect user data. Optimizing user experience, developing efficient debugging tools, and addressing latency caused by language interactions are essential for the advancement of dynamic web applications (Strein et al., 2006).

4.5.3. Multilingual development environments

MLDEs which are currently in use are deficient in supporting the development of multilingual systems. They fail to envisage cross-language relationships and lack static checks to determine the consistency of cross-language. In addition, they do not perform refactoring in dissimilar languages. They also demand improvement in inefficient language representation. These tools provide limited functionality to represent cross-language relationships and MLDE. They need an extension for language representation of visual languages like UML (Pfeiffer and Wasowski, 2015).

Future Directions

Multilingual development environments (MLDE) do not offer features like cross-language referencing, refactoring or debugging, etc. These applications prevail with flaws and have imprecise and erroneous support. Genuine multilingual IDE support is pivotal in improving the understanding, constructing, and maintaining high-quality software.

4.5.4. Interoperability in heterogeneous platforms

Programming languages need to support multiple platforms for interoperability but modern languages use platform-specific APIs extensively. Such APIs are based on the run time system of the software platform that causes portability problems (Yazdanshenas and Moonen, 2011).

Future Directions

Programming languages should support multiple software platforms to strengthen interoperability and increase performance. When a new platform is added, it requires interoperability with existing applications and libraries on that platform. In the future, the portability issues of software product lines need to be catered to by assimilating unique dimensions and handling dependencies on several platforms (Terceiro et al., 2010).

4.5.5. Multilingual applications

The detection of dependencies for large, complex, and diverse multilingual applications is quite challenging owing to the presence of several unique languages and their probabilistic combinations. No robust, scalable, and general approach is currently available for the detection of cross-language dependency. The available techniques target the identifications of dependencies within single source code applications (Polychniatis et al., 2013).

Future Directions

The future of multilingual applications lies in advanced dependency analysis techniques that can accurately identify cross-language links. Standardization and best practices for language interoperability will simplify the integration of different programming languages. Extending IDE capabilities and developing automated refactoring tools for cross-language dependencies will improve code maintainability. Domain-specific multilingual development environments tailored to specific industries can enhance development productivity. A benchmark is required to classify systems based on their size and language set. Besides, it is important to scrutinize the behavior of the method at the component level and to improve accuracy by applying greater statistical filters.

4.5.6. Cross-language software corpora

Software corpora assist in analyses reproducibility but static analysis of an entire application is hard and extremely challenging because of two reasons. First is the duplication of unnecessary code by multiple users and the second is the cross-language analysis, as most corpora are designed for single languages (Caracciolo et al., 2014).

Future Directions

Enhancing cross-language software corpora involves enlarging and diversifying existing datasets across various application domains and programming languages. Standardization of corpus formats and automated generation methods can ensure consistency and ease of access for researchers. Semantic analysis techniques can be explored to understand the meaning and context of language interactions. Privacy measures and anonymization techniques should be implemented to address privacy concerns related to corpora containing sensitive information (Schink et al., 2011; Shatnawi et al., 2018b).

4.5.7. Cross-language links

These hold prime significance in developing multilingual software systems. Present-day developing environments lack efficient support of links, across multilingual artifacts making it hard to establish and refactor cross-language links between artifacts associated with heterogeneity (Pfeiffer and Wasowski, 2012b).

Future Directions

It is imperative to find how cross-language links can be effectively dealt with. Also, how the frameworks for developers must be written to obtain the benefits of DSLs for analysis and refactoring. Future direction needs to encompass cross-language links between two general-purpose languages or between two domain-specific and declarative languages. Tool support is mandatory to identify cross-language links (Mayer and Schroeder, 2013a) and legacy languages require proper integration.

4.5.8. Software development and maintenance

The literature review reveals that thoroughly consistent support to develop and maintain software does not exist. Almost all tools which have been used to develop and maintain software are characterized by limitations. These limitations include but are not limited to partial program language support, and inconsistent use of metrics and testing methods. One of the important weaknesses of available tools is the deficiency in supporting the calculation of metric values autonomously on input programming language (Pribela, 2012; Rakić and Budimac, 2011).

Future Directions

Software development and maintenance demand evaluating the barrier and time to reverse engineer a product. For that, an advising system that is independent of input language is required. This system must be able to communicate with its user, not only about metric values but provide concrete advice to correct and refactor the source code as well. The integration of tools with a specific repository of software metrics is also required for future reverse engineering activities.

5. Discussions

The primary objective of this survey is to uncover state-of-the-art approaches for MLSCA. It focuses on MLSCA, its utility, and future SE and computer sciences domains. Valuable contributions extracted from the research are presented in the form of tools, languages, and application types to analyze XLLs as shown in Fig. 17. To uncover accurate pragmatic evidence, a comprehensive review methodology, focused domains, research queries, inclusion and exclusion criteria, sources of information from famous researchers and databases, and advanced search queries have been devised. A total of 2025 research papers are searched from selected research venues from January 2005 to December 2020. By employing three-staged exhaustive selection criteria, 76 publications are picked for advanced evaluation. Details of these shortlisted publications are presented as primary study type, source type, and temporal and geographical distribution in Figs. 3, 4, 5, and 6. It has been observed that contemporary software is developed by using several cross-language components.

5.1. RQ1: Research domains

To answer RQ1, the study identified research domains as a start to literature review findings. It is observed that out of 76 primary studies, 45 (59%) studies focus on static analysis of multilingual source code whereas 21 (28%) studies use a hybrid semantic analysis mechanism and only 10 (13%) studies are focused on dynamic analysis. 20 different software engineering domains have been identified from primary studies. All of these employ one of the aforementioned analysis mechanisms. The 15% of the domains include source code analysis and manipulation, 33% software maintenance and evolution, 21% program understanding and reverse engineering, 13% software re-engineering, 9% software metrics and requirements, 26% static analysis, 15% program understanding and reverse engineering, 13% XLL detection, 11% dynamic analysis%, 6% change impact analysis 6%, 11% analysis of enterprise applications and 11% multilingual development environments, 5% software optimization, 5% software visualization, and 5% software implementation 5%. The rest of the domains stands below 5%.

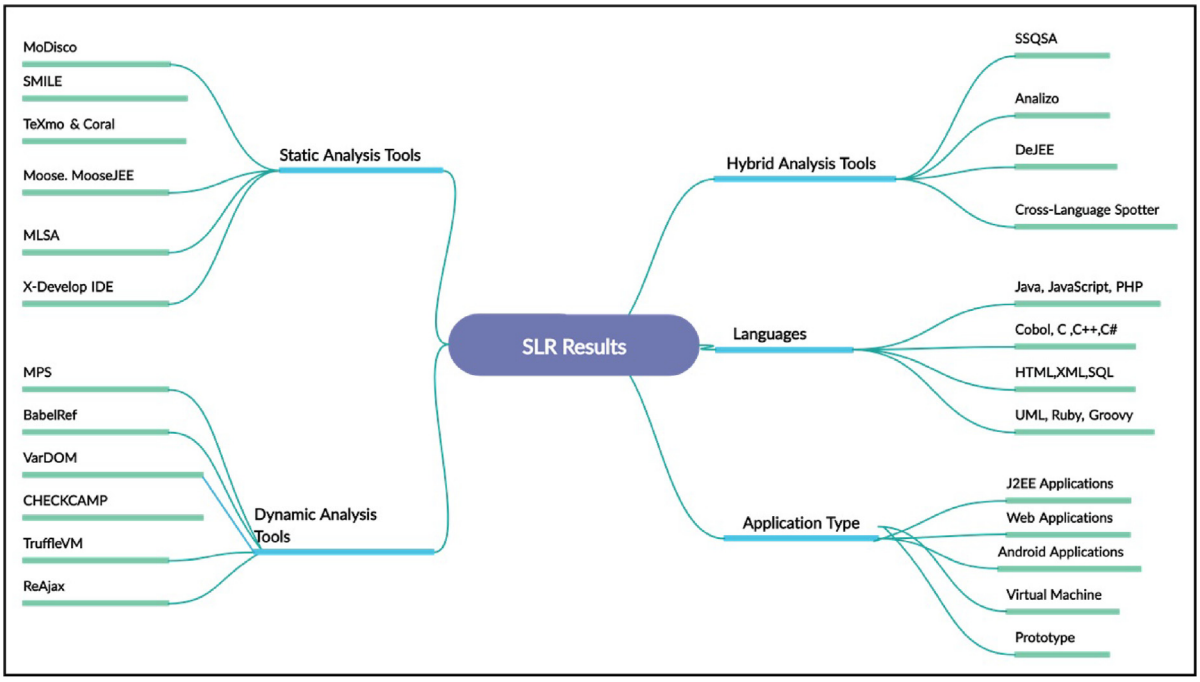


Fig. 17. SLR findings: cross-language links detection tools, techniques, and applications.

5.2. RQ2 and RQ3: Research contributions and limitations

The 'static analysis' mechanism constitutes 59% of primary studies and is the most widely used technique. Its overview reveals that it is not possible to successfully obtain the run-time dependencies. These run-time dependencies are the prerequisites for understanding the behavior and response of a given software application. The tools available for static analysis only support domain-specific languages.

The 'dynamic analysis' is used to counter the issues faced during static analysis and makes up 13% of primary studies. Dynamic analysis requires error-free source code in an executable form. Supporting analysis on client code while it is still embedded in the server code is daunting and generates errors. Similarly, the 'hybrid analysis' mechanism which employs a combination of static and dynamic analyses, mandates concrete background knowledge, domain knowledge, and expertise in all related tools. These combined requirements increase the complexity of the analysis and are not very common to find. 28% of the primary studies focus on hybrid analysis.

It is found that the tools used in most of these studies support limited languages. The tools are evaluated based on case studies employing toy examples, limited to a few hundred lines of code, and are domain-specific like web applications, Android, and enterprise applications to name a few. In the real world of contemporary software applications, however, numerous programming languages are interacting with each other across several domains through millions of lines of code. This software fails at data retrieval and visualization support. Only 5% of the primary studies support the visualization aspect. It is therefore concluded that until these problems are addressed, a centralized guide for reverse engineering activities cannot be created.

The research is organized and several features of XLA are classified that are shown in Tables A.4, A.5, and A.6. Every contribution is characterized with E# (number of Excel sheets). Contributions are segregated into 48% tools and techniques, 20% strategies, 10% of the studies consist of data gathering techniques such as questionnaires, surveys, interviews of formal and informal types, focus group plus case studies, 5% studies are based on systematic literature review, and 4% are taxonomy based. They focus on different kinds of language representations that are integrated into multiple software engineering domains. Each research offers individual support to resolve the requisite tasks. The analysis is supported in the form of different tools that are subdivided into tools of dynamic and static analysis and semantic/hybrid analysis. They are appraised via frameworks, prototypes, case studies, development environments, simulators, and plugins. Results of the prototype tools provide proof of the concept which is proposed in the models and approaches. Frameworks, models, and algorithms have been used to present analysis techniques. These also comprise UML, graphical, mathematical, or semantic models. Case studies or frameworks have been employed to evaluate the techniques mathematically. Surveys underscore the significance of multilingual IDEs, source code analysis, refactoring, XLA, and empirical appraisal of cross-language links.

5.3. RQ4: Development of schema and taxonomy

To assist the researchers and practitioners, a general schema (Fig. 15) representing the analysis process along with definitions of XLLs and associated concepts is developed. The contribution of this paper is a systematic literature review on XLLs and a comprehensive taxonomy named XLA, which incorporates all XLA components.

5.4. RQ5: Research challenges and potential future contributions

As the world is advancing, its needs and requirements are evolving along in line with the fiercely growing competition. Practically, all organizations and businesses are benefiting from various enterprise software applications. Efficient management of resources in a corporate or profession is a difficult task and corporate organizations across the globe are striving to devise strategies to successfully extract the desired material from an enormous quantum of data (big data) that is both, productive and inexpensive. Therefore, rather than developing a new application, software engineers prefer to re-engineer the existing ones.

However, it has been observed that most organizations hire third parties to perform this pivotal task where the original developers of the application are often left out of the picture. This is where the problem surfaces.

This research is an attempt to sift through the state-of-the-art literature focused on addressing the issue of upgradation/maintenance of software applications and summarizing the findings as a centralized guide. Consequently, the reverse engineering team must comprehend the source code as a first step and crack the structural behavior of the system. State-of-the-art software applications comprise countless languages and components where interactions between discrete entities of the applications could be concealed. Using present-day tools of reverse engineering in this perspective does not aid the analysis of MLAs because their focus is on specific domains, components, and languages. Analysis of large MLAs is getting increasingly complicated and traditional methodologies lack such capabilities. The following research challenges are found in three analysis approaches (see Fig. 18).

5.4.1. Patterns of dependencies

There are patterns of dependencies codification in MLAs due to run-time dependencies grounded on runtime events like sensor values and user actions to name a few. Dependencies related to communication protocols include HTTP, RMI, etc.

5.4.2. Diversity of applications

There are many types of applications and dependencies related to application context e.g. Web, client/server, the server only, the client only, embedded, desktop, and mobile.

5.4.3. Diversity of languages

There is a variety of dependencies among programming language versions. There are also combinations of XLLs among domain-specific languages and languages of general purpose.

GPL/DSL: Java/XML, Java/SQL, JS/HTML, Java/HTML.
GPL/GPL: Java/PY, Java/C, Java/JS.
DSL/DSL: HTML/CSS, XML/HTML, XML/SQL.

5.4.4. Diversity of frameworks

We should comprehend these frameworks in order to ascertain which callback methods are employed at particular times/for particular events, for example when the button is pressed by a user. Moreover, we should explicitly represent the dependencies that are managed by the frameworks; calls related to the application life-cycle management. We need to understand configuration files describing applications functionalities and callback methods offered by frameworks.

5.4.5. Identification and maintenance of XLLs

A comprehensible querying mechanism along with its GUI for visualization of XLLs in a multilingual source code is missing. XLLs are not part of any language but rather stand aloof which indicates that they are not assessed at the design stage for accuracy. Thus, such a GUI would help developers in the identification and maintenance of XLLs for upgradation.

5.4.6. Unavailability of compilers and editors

The available platforms for source code analysis have minimal flexibility in cross-language representation and MLDEs. In a multilingual environment, it is necessary to have platforms that provide support for all individual languages because only a few team members manage to comprehend the problems that occur because of multiple languages.

5.4.7. Lack of full test coverage of source code

Test writing is a crucial task that requires special care when dealing with XLLs in MLSAs. Automated tools cannot guarantee the full test coverage of the source code producing false positives and false negatives.

5.4.8. Incapability of data storage and retrieval

Existing workbenches have minimal data repositories for dynamically examining multilingual software corpora and object-oriented languages.

5.4.9. Meta-models diversity

Unique programming languages are designed using different meta-models and describe different syntactic, lexical, and semantic guidelines.

- Object-oriented paradigm C++, C#, Java.
- XML meta-models ASP, HTML, and JSP.
- Procedural meta-models, such as Basic, C, COBOL.

A better methodical investigation of dependency types is necessitated. Also, there is a need to extend existing approaches for highly refined couplings with dynamic UML models.

5.4.10. Tools inadequacy

The XLLs tools need support for generalization, dynamic and low-level languages as well as the number of supporting languages but in most cases, they are developed for Java-related technologies. Tools based on hybrid analysis have interoperability issues and portability issues. Additionally, in comparison to fully static detection, hybrid (dynamic/static) analysis does not offer consistent improvement. Moreover, they need to extend across different application domains. The MLSCA tools lack continuous reporting (documentation) or a graphical user interface to examine information flow. Some of the tools and techniques presented in the literature are not implemented for any case study to properly evaluate the findings or are evaluated on smaller projects of less than a thousand lines of code. The accuracy of an XLL detection tool is a key aspect of its validity. It requires an empirical appraisal and additional enhancements for persistence and categorization in XLLs.

5.4.11. Lack of design recovery techniques

Current tools and techniques only focus on the analysis or transformation of source code but do not support additional design recovery techniques.

5.4.12. Incomprehensiveness of source code

Present approaches lack identifying dependencies that reflect run-time behavior which is fruitful for architectural purposes. Moreover, it is not possible to extract from incomplete source code.

5.4.13. Multilingual code in source files

Certain programming languages allow fusing multi-language code inside a single source code file, e.g. an instance of reusing the custom tag in a JSP page. This tag is configured in an XML file which describes tags and their relevant characteristics and also maps the tags to their implementing tag handlers, scripted in Java. Different pieces of code may possess internal dependencies inside the same file and external dependencies with code in other files.

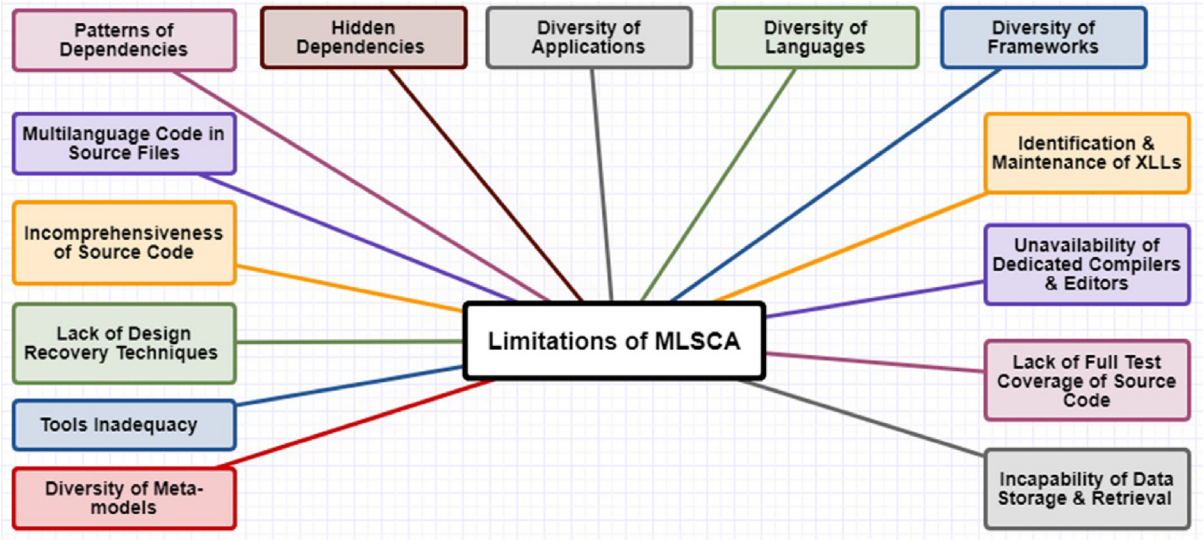


Fig. 18. Outcome of SLR: research gaps.

5.5. Potential future contribution

The MLDEs have been reported in many research studies that somehow support refactoring or clone code identification. Yet very few efforts have been made for a multilingual maintenance environment that supports reverse engineering today's large enterprise applications. A distributed environment or open-source tool that can offer source code analysis and manipulation of a sizeable set of languages and domains, along with recognition and visualization of XLLs is required. Refactoring, metrics calculation, clone code detection, etc. should also be provided as supporting maintenance activities.

With the combination of a highly scalable graph database and machine learning algorithms that leverage not only source code but also source code relationships (XLLs), models can be trained. Such models can effectively learn the patterns of XLLs in a multilingual application. This has the potential of opening up new avenues for developers to shape applications that are intelligent enough to traverse the present day's humungous and intertwined code in real-time (Kanade et al., 2020).

5.5.1. Large source code repositories

Popular repositories on the internet like GitHub, SourceForge, and BitBucket [https://software.ac.uk] host a multitude of open-source projects. Such repositories are gaining popularity and augmenting in size. Countless research projects mine the aforementioned repositories for valued data and evidence. If projects from such repositories are compiled, it may create an opportunity of altering the fruitful ones. In addition, this will assure the syntactical correctness of source code and the project dependencies either being self-contained or accessible on the Internet. However, environmental factors taken by the developers like versions, the existence of outside dependencies and build tools, etc. mostly go uncaptured by these repositories. This heterogeneity of the projects makes the efficient extraction of links an extremely daunting task. This is because it is inappropriate to apply a universal or single solution to all.

5.5.2. Enterprise resource planning software

Oracle NetSuite, Microsoft Dynamics, Openbravo, and SAP [htttps://www.selecthub.com/erp-software] are famous ERP software containing very large sets of code and are customizable applications. According to the need of the organization, the ERP software is customized and then deployed. The software is tailored for others but the companies owning such software are always maintaining and handling very large code of ERP software. Thus, traditional source code analysis techniques presented in the literature are incapable of dealing with such a huge magnitude of code.

5.5.3. Cloud computing and graph databases

Relational databases are now obsolete, because of their incapability of handling gazillion lines of code. Software developers are adopting cloud computing services like Amazon Web Services (AWS) [https://aws.amazon.com] and graph-based databases (Neo4J) [https://neo4j.com/] to store and retrieve extensive relationships (XLLs) for maintenance of cross-language applications, instead of buying, owning, and maintaining large databases and servers.

5.5.4. Machine learning and natural language processing

As software continues to encompass the world, it is accumulating gazillions of lines of code, and billions of applications developed using diversified programming languages, infrastructure, and frameworks. More frequently the analysis of source code requires a stronger understanding of statistical attributes to obtain healthier metrics for a machine learning model. Machine learning techniques can help companies in streamlining the codebase and software delivery processes and offer a better understanding and management of their engineering talents. Business groups have a distinctive prospect of gaining a competitive edge by handling software artifacts as data and employing machine learning and advanced data science techniques (Mili et al., 2019).

5.5.5. Limitation

It is an interdisciplinary field of research pertaining to the processing of Natural language, the structure of programming language, and historic analysis. It focuses on acquiring large-scale source code datasets to perform software engineering tasks on automation. Applying ML on source code has its own limitations like requiring zero error rate. A tiny, single misprediction may lead to the whole program's compilation failure. It can be very costly to manually label datasets, so researchers typically have to develop correlated heuristics. MLonSourceCode problems comprise a myriad of data mining tasks, which from a theoretical perspective may be trivial yet technically puzzling because of the scale or intricacies, for instance, code clone detection and similar developer clustering.

5.5.6. Abstract syntax tree

AST nodes are used in conventional Natural language processing algorithms, containing sequence-to-sequence deep learning models, which is the most powerful representation in addition to being the toughest to handle. The relevant ML models include various graph embeddings and gated graph neural networks. The usefulness of big code: greater the frequency of source code analysis, the more reliable the statistical properties underscored leading to refined metrics of a trained machine learning model (Mili et al., 2019; Saha et al., 2020).

With software consuming the world, there is an accumulation of gazillions of lines of code, and uncountable applications being developed from a plethora of programming languages, frameworks, and infrastructure. MLonSourceCode can help companies reverse engineer their multilingual software applications. By considering software artifacts as data and implementing advanced data science, in combination with techniques of machine learning, businesses, and enterprises can easily gain a competitive edge over their peers.

6. Study implications for readers groups

6.1. Language technology researchers and practitioners

6.1.1. Challenges

Language technology researchers can benefit from exploring the challenges related to identifying patterns of dependencies and diversity in multilingual source code analysis. Understanding the complexities of cross-language links and communication protocols can guide the development of advanced language technology tools.

6.1.2. Future directions

Researchers in this group can further investigate the potential of machine learning and graph databases for scalable analysis of multilingual applications. They can explore the application of advanced data science techniques to enhance language technology models and improve source code analysis.

6.2. Software developers and engineers

6.2.1. Challenges

For software developers and engineers, the data highlights the difficulties in managing cross-language dependencies in various application contexts and diverse frameworks. Understanding the limitations of existing tools can help developers be aware of potential pitfalls in source code analysis.

6.2.2. Future directions

This group can explore the potential of GUI-based querying mechanisms for identifying and maintaining cross-language links in multilingual source code. They can consider adopting cloud computing and graph databases to improve the storage and retrieval of dependencies for large-scale applications.

6.3. Academics and students in computer science

6.3.1. Challenges

Academics and students can delve into the data to understand the complexities of different programming languages, metamodels, and abstract syntax trees in cross-language analysis. They can explore the limitations of MLonSourceCode problems and the challenges in design recovery techniques.

6.3.2. Future directions

This group can conduct research on developing methodologies for investigating diverse dependency types and improving source code comprehension using machine learning models. They can also explore the potential of advanced data science and statistical attributes for better metrics in machine learning applications.

6.4. Policy and decision makers

6.4.1. Challenges

Policy and decision-makers can gain insights into the challenges faced by software development teams in managing multilingual codebases. Understanding the limitations of existing tools can help in making informed decisions about adopting advanced technologies.

6.4.2. Future directions

This group can explore the potential of supporting language technology research and development in the context of cross-language source code analysis. They can consider investing in machine learning and graph database technologies to improve software development practices.

6.5. General readers and enthusiasts

6.5.1. Challenges

General readers interested in software technology can gain an overview of the complexities involved in cross-language source code analysis. They can understand the challenges in identifying dependencies and links in multilingual applications.

6.5.2. Future directions

General readers can explore the potential of emerging technologies, such as machine learning and cloud computing, in improving software development and maintenance. Understanding the significance of advanced data science techniques can provide insights into the evolving landscape of software engineering. By mapping the research work to different research groups, the findings and future directions are tailored to their specific interests and needs. This approach ensures that each group can derive meaningful and relevant insights from the research, fostering further investigation and advancement in the field of cross-language source code analysis and manipulation.

7. Implementation guidelines for integrating XLLs detection

7.1. Identify relevant languages and frameworks

Before integrating cross-language link detection, identify the programming languages and frameworks commonly used in your software projects. Understand the interactions and dependencies between these languages to focus on the essential areas for link detection.

7.2. Select suitable tools or develop custom solutions

Evaluate existing tools that offer cross-language link detection capabilities. Choose tools that align with your project's needs and support the languages and frameworks in use. If no suitable tools are available, consider developing custom solutions tailored to your specific requirements. Collaborate with language technology researchers or practitioners to build efficient detection mechanisms.

7.3. Integrate cross-language analysis in the development workflow

Integrate cross-language link detection as part of the regular software development workflow. Consider integrating it with code analysis and review processes to identify and address dependencies early in the development cycle.

7.4. Provide training and resources

Offer training sessions and resources to software developers and engineers on using cross-language link detection tools effectively. Ensure that team members are proficient in interpreting and acting upon the detected links.

7.5. Establish code review guidelines

Develop code review guidelines that emphasize the identification and management of cross-language links. Encourage developers to review and refactor code to reduce unnecessary dependencies and enhance code modularity.

7.6. Visualize cross-language dependencies

Implement a graphical user interface or visualization tool that presents cross-language dependencies clearly and understandably. Visual representations can aid developers in comprehending complex relationships.

7.7. Automate dependency tracking

Automate the tracking of cross-language dependencies to reduce manual effort and ensure consistent analysis across the codebase. Continuous monitoring of dependencies can help detect changes and potential issues promptly.

7.8. Address false positives and negatives

Address false positives and false negatives generated by the detection tools. Fine-tune the detection algorithms to minimize inaccurate results and improve the accuracy of cross-language link identification.

7.9. Monitor codebase evolution

Regularly monitor the codebase's evolution to identify new cross-language links that might emerge during the development process. Implement mechanisms to keep the link detection up to date with the code changes.

7.10. Collaborate with language technology researchers

Foster collaboration with language technology researchers and practitioners to stay updated on advancements in cross-language analysis tools and methodologies. Research partnerships can lead to the development of more robust and efficient link detection solutions.

7.11. Evaluate effectiveness and impact

Continuously evaluate the effectiveness and impact of cross-language link detection in software development practices. Collect feedback from developers and measure improvements in code quality, modularity, and maintenance efficiency. By following these implementation guidelines, software development teams can effectively integrate cross-language link detection into their practices and tools. This integration will enable them to identify and manage dependencies across diverse languages, leading to more efficient and reliable multilingual software systems.

8. Conclusions and future work

This systematic literature review is based on a total of 76 primary studies, which span over 15 years. The selected studies investigated methods and approaches to detect cross-language links and their dependencies in multilingual source code environments. One key objective for this systematic literature review is to create a 'go-to' literature database at the end, where professionals from software engineering could find all the content pertinent to the analysis and XLL detection for all major types of multilingual applications like Java enterprise applications, Android applications, web applications, etc. However, it has been observed that traditional source code analysis mechanisms which are employed to reverse engineer contemporary software applications face scores of problems and limitations that need to be addressed in the future.

Software maintenance and upgradation are the backbones of today's fastly growing technological evolution. As a result, almost every modern-day software application, at some point in its life cycle, comes to a point where it requires refining and updating. Decoding the body of a software application is done through various analyses which can be broadly categorized into 'static analysis', 'dynamic analysis, and 'hybrid analysis' Since most state-of-the-art software applications are heterogeneous i.e. they comprise multilingual source codes, backtracking means that it must be able to fully comprehend and unravel the cross-language links and their dependencies. However, research reveals that most organizations hire third parties to perform this pivotal task whereas the original developers of the application are often left out of the picture. The survey's limitations include its focus on specific multilingual applications, such as dynamic web applications, Java enterprise applications, Multilingual Development Environments (MLDEs), ERP solutions, etc, which may limit the exploration of other relevant areas. The survey does not delve into the implementation details of specific tools or techniques, as its primary goal is to provide an overview and synthesis of existing research. Despite these limitations, the survey offers valuable insights, and guidelines, serving as a foundational resource for researchers and practitioners in the field of multilingual source code analysis and manipulation.

CRediT authorship contribution statement

Saira Latif: Conceptualization, Data curation, Writing – original draft. **Zaigham Mushtaq:** Conceptualization, Formal analysis, Writing – original draft. **Ghulam Rasool:** Methodology, Project administration, Resources. **Furqan Rustam:** Software, Visualization, Investigation. **Naila Aslam:** Investigation, Validation, Formal analysis. **Imran Ashraf:** Supervision, Validation, Writing – review & editing.

Declaration of competing interest

All authors have participated in (a) conception and design, or analysis and interpretation of the data; (b) drafting the article or revising it critically for important intellectual content; and (c) approval of the final version.

This manuscript has not been submitted to, nor is under review at, another journal or other publishing venue.

The authors have no affiliation with any organization with a direct or indirect financial interest in the subject matter discussed in the manuscript

Data availability

Data will be made available on request.

Appendix. Tables

See Tables A.3–A.6.

Table A.3
Quality assessment of selected primary studies.

Reference	Year	QA 1	QA 2	QA 3	QA 4	Total
Polychniatis et al. (2013)	2013	1	0.5	1	0	2.5
Mayer (2017)	2016	1	1	0.5	1	3.5
Strein et al. (2006)	2012	1	0.5	0	1	2.5
Lozano et al. (2016)	2016	1	0.5	1	0.5	3
Linos et al. (2007)	2007	1	0.5	1	0	2.5
Mushtaq et al. (2017b)	2017	1	1	1	0	3
Mushtaq and Rasool (2015)	2015	1	1	1	0	3
Mushtaq et al. (2017a)	2017	0.5	0.5	1	1	3
Chikofsky and Cross (1990)	2017	1	1	1	1	4
Aslam and Ashraf (2014)	2018	1	1	1	0	3
Rashid et al. (2013)	2019	0	1	0.5	1	2.5
Mushtaq et al. (2017a)	2019	1	1	1	1	4
Khan et al. (2017)	2014	1	0	1	0	2
Kitchenham et al. (1999)	2012	1	0.5	1	1	4
Kitchenham (2004)	2012	0.5	0	1	0.5	2
Kitchenham et al. (2002)	2012	0	0	1	1	2
Afzal et al. (2009)	2013	0	0	1	1	2
Binkley (2007)	2013	0.5	0.5	1	1	3
Cerulo (2006)	2011	0	0	1	1	2
Pfeiffer and Wasowski (2011)	2013	1	0	1	0	2
Mayer et al. (2017)	2013	0.5	0.5	0.5	1	2.5
Nguyen et al. (2012)	2015	0.5	1	1	0.5	3
Shatnawi et al. (2018a)	2012	0	1	1	0.5	2.5
Rakić et al. (2013)	2019	1	0.5	1	0.5	3
Mayer and Schroeder (2014)	2013	0	1	0	0	1
Mayer and Schroeder (2013a)	2018	0.5	1	1	1	3.5
Van Der Storm and Vinju (2015)	2007	0	0	0.5	0.5	1
Li et al. (2017)	2013	1	0.5	1	0	2.5
Lyons et al. (2017)	2013	1	1	1	0.5	3.5
Albertsson (2006)	2017	1	1	0	0.5	3.5
Xia et al. (2014)	2014	1	1	1	1	4
Jiang et al. (2013)	2017	0	1	1	0	2
Misra et al. (2012)	2017	0	0.5	0	0	0.5
Perin (2012)	2013	1	1	0.5	1	3.5
Grimmer et al. (2018)	2015	0.5	0	1	1	2.5
Pfeiffer and Wasowski (2012b)	2015	0	0	1	1	2
Pfeiffer and Wasowski (2015)	2019	1	1	1	1	4
Klint et al. (2009)	2012	0	0.5	1	1	2.5
Perin et al. (2010)	2019	1	1	1	0	3
Savić et al. (2014)	2014	0	0.5	1	0.5	2
Yazdanshenas and Moonen (2011)	2006	1	1	0.5	0	2.5
Kargar et al. (2020)	2015	0	0.5	1	0.5	2
Bui et al. (2019)	2010	0	0	0.5	0	0.5
Lehnert et al. (2013)	2005	1	0	1	1	3
Savić et al. (2012)	2018	0.5	0	0	0	0.5
Caracciolo et al. (2014)	2019	1	0.5	1	1	3.5
Rakić and Budimac (2011)	2018	0	1	1	1	3
Pribela (2012)	2018	0.5	1	1	1	3.5
Gosain and Sharma (2015)	2018	0	0.5	1	0	1.5
Pfeiffer et al. (2014)	2019	0	0	0.5	0.5	1
Rakić and Budimac (2013)	2008	0	0.5	1	1	2.5
Bruneliere et al. (2010)	2010	0	1	0.5	0.5	2
Aarssen et al. (2019)	2010	1	0	1	0	2
Hecht et al. (2018)	2016	1	1	1	0	3
Mili et al. (2019)	2010	0	1	1	0	2
Shatnawi et al. (2019)	2009	0	1	1	0	2
Moise and Wong (2005)	2017	1	1	0.5	0	2.5
Tomassetti et al. (2013b)	2007	1	0	1	0	2
Savić et al. (2013)	2007	1	1	0.5	0	2.5
Kolek et al. (2013)	2008	1	0.5	1	1	3.5
Marchetto et al. (2012)	2018	0	0	0	0	0
Gerlec et al. (2012)	2019	0	1	0.5	0.5	2
Tomassetti et al. (2014a)	2006	0	0.5	1	1	2
Hadjidj et al. (2008)	2014	1	0	1	1	3
Nguyen et al. (2014)	2010	1	0.5	1	1	3.5
Grimmer et al. (2015)	2013	0	0	1	0	1
Shatnawi et al. (2018b)	2019	0	0.5	0.5	0.5	1.5
Joorabchi et al. (2015)	2013	1	0.5	0.5	1	3

(continued on next page)

Table A.3 (*continued*).

Nair et al. (2015)	2011	1	0.5	1	0.5	3
Shatnawi et al. (2017)	2015	0.5	0	1	1	2.5
Bogar et al. (2018)	2014	0.5	0	1	1	2.5
Terceiro et al. (2010)	2010	0.5	0	0.5	0.5	1.5
Tomassetti et al. (2014b)	2015	0	0	1	1	2
Marinescu and Jurca (2006)	2010	0	0	0.5	0	0.5
Tomassetti et al. (2013a)	2012	1	1	1	1	4
Kontogiannis et al. (2006)	2018	1	0.5	1	1	3.5
Schink (2013)	2013	0	0	1	1	2
Jinan et al. (2017)	2014	1	0.5	1	0	2.5
Mayer and Schroeder (2013b)	2010	0	0	1	1	2
Janes et al. (2013)	2014	0	1	1	0.5	2.5
Boughanmi (2010)	2011	0.5	0	1	1	2.5
Sultana et al. (2016)	2013	1	0.5	1	1	3
Jenkins and Kirk (2007)	2014	0	0	1	1	2
Pfeiffer and Wasowski (2012a)	2013	0	0	1	1	2
Angerer (2014)	2011	0	0	1	0	1
Kienle and Müller (2010)	2014	1	0	1	1	3
ANTRL (2021)	2018	0.5	0	1	1	2.5

Table A.4
Static analysis mechanism.

Sr #	Ref.	Technique	Tool	Application type	Meta-model	Supporting languages	Model description
1	Jiang et al. (2013)	Augmented call graph	MoDISCO	J2EE applications	OO	Java	KDM Meta-model
2	Misra et al. (2012)	Program dependency graph	MoDISCO	JEE applications	Procedural SOA MM	JavaHTML	OMG's KDM
3	Kitchenham et al. (2002)	Static dependency call graph	MoDISCO/DeJEE	JEE application	OO/XML	Java/Html	OMG's KDM
4	Grimmer et al. (2018)	eCST generation on software repositories	SMILE/ANTLR	Generic	OO	Java/C#	Special software
5	Pfeiffer and Wasowski (2012b)	eCST representation of source code	SNEIPL prototype	Generic universal SWNE	OO	Java/C#	eCST Generator, ANTLR parser generator
6	Klint et al. (2009)	eCST generation framework	SMILE	Generic	Procedural/OO	Java, Modula-2, COBOL	Framework for automatic assessment
7	Perin et al. (2010)	Language independent static call graph	SMILE/SNEIPL/SSCA	Generic	OO/Procedural	C#, Java, Delphi, Cobol etc	SSQSA framework: a set of static analyzers
8	Kargar et al. (2020)	Parser based eCST generator	SSQSA	Generic	Procedural	Java, C#, WSL, OW etc	eCST Generator, parser generator ANTLR
9	Mayer et al. (2017)	Search based & universal representation	MLDE prototypes	Web/Prototype	OO/XML	Java, HTML, XML & Groovy	Coral: Search-based RM, TexMo: universal language representation.
10	Pfeiffer and Wasowski (2011)	Explicit model, tags, search & interface based techniques	TeXmo prototype	MLDE	Explicit RM	Syntactic, Universal: Java, Java Script, HTML and XML	ML IDE for classification & representation
11	Bui et al. (2019)	Open-source meta-programming framework & workbench	Rascal API	IDE	OO	C++, java, JavaScript, RASCAL, Json	A lightweight API, support concrete syntax using external parsers
12	Pribela (2012)	XLA binding and refactoring.	Wicket, Hibernate, SPRING	IDE	OOP	Java, DSLs (Spring, HBM, HQL, Wicket)	Automated cross language references
13	Youtube (2021)	MLSA: XLL approach on XLAs	Eclipse Hybrid Framework	IDE	XLL Hybrid Design	Ruby, XML Android Java	Contextual connections: Objects of different languages using QVT R.
14	Caracciolo et al. (2014)	XL linking and refactoring	Framework	Web IDE, XLA	EMF, Modisco	Java, Spring, Hibernate, HTML & Wicket	Automated XLL with Hibernate & Wicket environment.
15	Rakić and Budimac (2011)	Identification of XLL patterns	Standard patterns for XLLs	IDE	OO	Android, Spring, JBeans, XML, HQL, OpenMeetings	Identified eight patterns of XLL
19	Shatnawi et al. (2019)	Framework of Bilateral Neural Networks (Bi-NN)	Bi-NN Framework	Generic	OO	Java, C++	Graph-based machine learning algorithm and token-based techniques

(*continued on next page*)

Table A.4 (*continued*).

20	Youtube (2021)	MM for XL analysis & refactoring	X-Develop	OO IDE, Web application	OO	ASP, HTML, C#, VB, J#	Language independent MM from XL systems
21	Polychniatis et al. (2013)	Standardized approach for detecting XLDs	Matching common tokens	Generic	OO/XML	JSF, Java, SQL & Spring specific XML	Spring web flow framework or SWF2
22	Kitchenham (2004)	Facts extraction for MLAs.	Source Navigator (SN)	Heterogeneous SW	Procedural, OO	C. C++, Java. COBOL, TCL, Fortran	Used fact extractors in Source Navigator
23	Gerlec et al. (2012)	Source code modularization	Genetic algorithm	Generic (Firefox Case study)	OO	C/C++, Java, LISP/Perl	Source code modularizing using the genetic algorithm assumed semantic, nominal, CDG
24	Tomassetti et al. (2014a)	Call- graph dependency	Lightweight MLSA architecture	IDE	Procedural, Scripting	C/C++, Python, JScript	Architecture of inter-language, intra-language calls and dynamic calls
25	Shatnawi et al. (2018b)	Source verification environment for static analysis and model checking.	MOPED	Open source generic SW	Procedural	C, C++, JAVA, ADA, FORTRAN, Remopla	GCC compiler which is known to be an opensource compiler
26	Khan et al. (2017)	A model browser, supports extension, stomization & visualization.	MoDicso	IDE	OO	Java, XML, Struts, Hibernate	Standard Meta-models: KDM & SMM
27	Joorabchi et al. (2015)	Virtual call Identification & interconnection	Modified BSD	Heterogeneous SW	Procedural, Script, OO	C, Java, Perl, Apache	Identification of virtual calls, interconnections.
28	Nair et al. (2015)	XL refactoring techniques.	OpenRefactory C and Fortran	Prototype	Procedural	C, Fortran	Data flow and dependence analysis
29	Shatnawi et al. (2017)	Transaction analysis in JEAs	Enriched FAMIX metamodel	Java Enterprise Application (JEA)	OO	Java, EJB, JSP, Applets	FAMIX language independent meta-model
30	Bogar et al. (2018)	RASCAL is a domain-specific language for source code analysis	RASCAL	Heterogeneous applications	OO	Generally applicable to all object-oriented languages	Creation of intermediate language
31	Tomassetti et al. (2014b)	Measure program structure's instability	Metric Icc overlaid on the instability metric I, of Martin.	Web application	OO	Hipergate, Jedit, Toscanaj, J2EE	Graphs as complex networks
32	Marinescu and Jurca (2006)	Valuating dependencies: SEA/SEB algorithm	CodeSurfer using ICFG	Architecture	Procedural	C/C++	System Dependence Graph (SDG)
33	Kontogiannis et al. (2006)	Analyze multilingual call graphs	Pybind11 FFI	MLSA SW architecture	Script, OO	C++, python	Used Lakosian design (Lakos 1996) for C++
34	Jinan et al. (2017)	Model-based approach	MOF & OCL	Enterprise applications	Unifying meta-model	Java, uflow: Upl, & DSL	AST grammar & ANTLR parser generator
35	ANTRL (2021)	Multiple graphs and slicing technique.	Prototype	Prototype	OO, Procedural	C, C++, java	Knowledge Discovery Metamodel (KDM)
36	ANTRL (2021)	Evaluate XL support mechanisms.	Jtrac, TexMo	JTrac Web application	XML, OO	Java, HTML, XML, JScript	Cross-language support & eveluation
37	ANTRL (2021)	I2SD: RE tool for EJB with interceptors.	I2SD + SQuAVisiT	EJB applications	OO, XML	XML, JavaCC	EJB with UML sequence diagram interceptors
38	Pang et al. (2018)	Parsing with modern multilingual features	Eclipse tool	IDE	Procedural, OO	C, C++, C#, Java, Java Script	Open Source Parser: generates metrics
39	Kanade et al. (2020)	EJB modeling	MOOSE	JEAs	OO	Java, XML, JEAs languages	MOOSE JEE on FAMIX platform
40	Saha et al. (2020)	Setting & running multilingual empirical studies: workbench	Pangea	XL SW corpora	OO	Java, XML	Multilingual software corpora static review is achieved through object model snapshots
41	[106]	A library Spot XLR automatically	AST Library:	Angular JS app	OO/XML	Ruby, JScript, HTML, XML, Java & properties files	Language agnostic approach: Spot XLR automatically, Summarize set of parsers.

(*continued on next page*)

Table A.4 (*continued*).

42	Hecht et al. (2018)	Model-driven interoperability: Algorithm	Xtext2Sonar Tool (Text & SonarQube)	Link smart devices: IoT gadgets	Xtext (dedicated language MM)	Java & text-based DSLs	ANTLR Parser on EBNF grammar, accuracy tool (SonarQube) on EBNF/PEGsXtext.
43	ANTRL (2021)	Information flow: Program slicing.	Grammatech CodeSurfer	Component-based system	OO	C and C++ programs	OMG's KDM Meta-model
44	Binkley (2007)	Translate JSP pages to java servlets.	Implementation of JSP pages in a KDM model.	JEE/JSP	The MM of JSP: MoDisco	Java, JBeans, Jscript, XML, JSP & JSF.	Jasper: Translate JSP to Java, MoDisco: extract KDM from Java.

Table A.5
Dynamic analysis mechanism.

Sr #	Ref.	Technique	Tool	Application type	Meta-model	Supporting languages	Model description
1	Eclipse Foundation (2021)	Trace link the preservation of source code.	MaTraca	Web application prototype	XML, OO	XML, Java	CHAQ meta-model, REST web service. Xform XML way of defining web interface
3	Rakić and Budimac (2013)	Seamless language integration method	MPS	EJB, Interceptors.	OO/XML	Java/XML	Language independent editors, Utilize MPS & Eclipse with full platform support.
4	Mili et al. (2019)	Identification & renaming XL entities.	BabelReF	Web application	XML, D-model	PHP, HTML	Single tree-based framework, identify client-side entities/connections directly.
5	Moise and Wong (2005)	WALA: Build call graphs	VarDOM	Dynamic web application	XML, IDEs	HTML, CSS, PHP, JS	Symbolic execution & variability-aware parsing/call-graph to build a VarDOM model.
6	Savić et al. (2013)	Abstract model creation	CHECKCAMP	Android, IOS	OO	C, Java	Collection of run-time code-based and GUI related metrics
7	Angerer (2014)	Dynamic compilers: automate access	TruffleVM	Virtual machine	OO, Procedural	JavaScript, Ruby, C	A multilingual runtime helps Consistent writing of various language implementations.
8	Moonen (2001)	Benchmark: Execution traces identification:	API for graph repre-sentation.	OO software applications	OO APIs	Java, XML ACL, API 1, 2 & 3, ASM, bytecode	Clustering algorithm based on graphs, Cluster API approaches that use the quality method.
9	Schink et al. (2011)	Compile source code to Java byteCode.	Lässig	Prototype virtual machine	Machine level	bytecode	Standard traceability: Compile languages to virtual machine.
10	ANTRL (2021)	Ajax based Reverse engineering	ReAjax	Ajax web application	OO, XML	Java, PHP	GUI-based state models from Ajax applications.

Table A.6
Hybrid analysis mechanism.

Sr #	Ref.	Technique	Tool	Application type	Meta-model	Supporting languages	Model description
1	Binkley (2007)	Static dependency call graph	DeJEE	JEE application	OO, KDM	Java, XML	Language independent model on MoDisco.
2	Misra et al. (2012)	MLE Network extractor	SNEIPLE	Generic	SSQSA framework	Java, Modula-2, and Delphi.	ANTLR parser generator
3	Pfeiffer and Wasowski (2015)	Automatic assessment framework	SMILE	Generic	XML	MPL.	Automated assessment: SW metrics
4	Savić et al. (2014)	Metrics extractor for MLE	SMILE	Generic	OO/Procedural	Java/Modula-2/COBOL	Call graph and Ecst
5	Yazdan-shenas and Moonen (2011)	eCST framework	SSQSA	Ontological systems.	OWL-2 to SSQSA	OWL-2 Intermediate language.	Integrating OWL2 to SSQSA
6	Angerer (2014)	Dynamic compiler	TruffleVM	Virtual machine	OO	Java, Ruby, C	Programming language composition, standard access to data & code
7	Varanasi and Belida (2015)	Iterative methodology: Taxonomy	GPL and DSL, Hibernate, Spring MVC.	Iterative methodology	XML, OO	C/C++, JScript, Java, PHP. YAML, JSON & XML.	Taxonomy: Iterative 7-step methodology

(*continued on next page*)

Table A.6 (*continued*).

9	Bruneliere et al. (2010)	Hybrid algorithm to test CFG	Static Code Analysis	Safety-related applications	Hybrid analysis framework	DSLS	ST/FBD parser
10	Mayer (2017)	Automatic detection collection & visualization of MLAs	MMT	Visual studio applications	Visual studio.NET SDE	UML	Multilingual complexity analysis at intermediate stage (like MSIL)
11	Tomassetti et al. (2014a)	Compiler-generated AST	PAIGE, Clang AST	IDE	Procedural, OO	C, C++	MLSA architecture
12	Terceiro et al. (2010)	SW vulnerability enumeration of intelligent data processing.	Methodology	Intelligent data application	XML, OO	C, C++, Java, C#	Use of RE, AST, CFG, DFG for Software Vulnerability Analysis
13	Mushtaq et al. (2017b)	Identified research challenges of source code analysis	N/A	Generic	OO	Java	Use CFG, Call graph, AST, VDG
14	Shatnawi et al. (2018a)	Model for RE process	Meta model DATES	OO enterprise applications.	OO	C++, C#	Intermediate representation model
15	Boughanmi (2010)	Taxonomy (Classification of Language Interactions)	Identification of possible semantic interactions	Polyglot applications, IDE	OO, Procedural	Xml, Java	Use of commits to analyze crosslanguage pairs.
16	Jenkins and Kirk (2007)	Markov Chains methods: Instantly identify XLLs, semantic of container artifacts	Cross-LanguageSpotter	IDE, web frameworks	OO, representation of the ASTs called code models	Java, Ruby, JavaScript, XML HTML, and Properties	Classifier is based on Weka5, a well-known Java machine learning library.
17	Pfeiffer and Wasowski (2011)	Novel propagation method analyzes SW dependency SW based on the type of change.	Prototype tool EMFTrace.	Generic	Unifying meta-model	Java, JUnit, UML models, URN /OWL models	Impact propagation rules
18	Mili et al. (2019)	Eclipse based database connectivity with Java	SQL-schema-comparer library	Library IDE development	DB	Java, SQL, XML	Eclipse tool modifications
19	Saha et al. (2020)	Change impact analysis (CIA) in constructing dependency graphs (CDG), (SDG).	Computing & annotating to CSDGs and linking multiple CSDGs to express XLDs.	Prototype	Multilingual environments	Used for dependency analysis of cross-language SPLs	Computing SDGs for each PL, annotating SDGs to CSDGs and then linking to XLDs.
21	Cerulo (2006)	Generation of dependency graphs, and software evolution analysis	Analizo visualization toolkit	UNIX	OO, Procedural	C, C++, Java	Doxygen Parser

References

Aarssen, R., Vinju, J., van der Storm, T., 2019. Concrete syntax with black box parsers. arXiv preprint arXiv:1902.00543.

Adams, J., Kear, K., 2003. Method and system for estimating software maintenance.

Afzal, W., Torkar, R., Feldt, R., 2009. A systematic review of search-based testing for non-functional system properties. Inf. Softw. Technol. 51 (6), 957–976.

Albertsson, L., 2006. Holistic debugging–enabling instruction set simulation for software quality assurance. In: 14th IEEE International Symposium on Modeling, Analysis, and Simulation. IEEE, pp. 96–103.

Angerer, F., 2014. Variability-aware change impact analysis of multi-language product lines. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. pp. 903–906.

ANTRL, 2021. ANTLR (Another Tool for Language Recognition (Ph.D. thesis).

Aslam, S., Ashraf, I., 2014. Data mining algorithms and their applications in education data mining. Int. J. 2 (7).

Binkley, D., 2007. Source code analysis: A road map. In: Future of Software Engineering (FOSE'07). IEEE, pp. 104–119.

Bogar, A.M., Lyons, D.M., Baird, D., 2018. Lightweight call-graph construction for multilingual software analysis. arXiv preprint arXiv:1808.01213.

Boughanmi, F., 2010. Multi-language and heterogeneously-licensed software analysis. In: 2010 17th Working Conference on Reverse Engineering. IEEE, pp. 293–296.

Bravenboer, M., Visser, E., 2004. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 365–383.

Bruneliere, H., Cabot, J., Jouault, F., Madiot, F., 2010. Modisco: a generic and extensible framework for model driven reverse engineering. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. pp. 173–174.

Bui, N.D., Yu, Y., Jiang, L., 2019. Bilateral dependency neural networks for cross-language algorithm classification. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 422–433.

Canfora, G., Di Penta, M., 2007. New frontiers of reverse engineering. In: Future of Software Engineering (FOSE'07). IEEE, pp. 326–341.

Caracciolo, A., Chis, A., Spasojevic, B., Lungu, M., 2014. Pangea: A workbench for statically analyzing multi-language software corpora. In: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation. IEEE, pp. 71–76.

Cerulo, L., 2006. On the use of process trails to understand software development. In: 2006 13th Working Conference on Reverse Engineering. IEEE, pp. 303–304.

Chikofsky, E.J., Cross, J.H., 1990. Reverse engineering and design recovery: A taxonomy. IEEE Softw. 7 (1), 13–17.

Eclipse Foundation, 2021. The community for open innovation and collaboration. https://www.eclipse.org/.

Gerlec, Č., Rakić, G., Budimac, Z., Heričko, M., 2012. A programming language independent framework for metrics-based software evolution and analysis. Comput. Sci. Inf. Syst. 9 (3), 1155–1186.

Gosain, A., Sharma, G., 2015. A survey of dynamic program analysis techniques and tools. In: Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014. Springer, pp. 113–122.

Grimmer, M., Schatz, R., Seaton, C., Würthinger, T., Luján, M., Mössenböck, H., 2018. Cross-language interoperability in a multi-language runtime. ACM Trans. Programm. Lang. Syst. (TOPLAS) 40 (2), 1–43.

Grimmer, M., Seaton, C., Schatz, R., Würthinger, T., Mössenböck, H., 2015. High-performance cross-language interoperability in a multi-language runtime. In: Proceedings of the 11th Symposium on Dynamic Languages. pp. 78–90.

Hadjidj, R., Yang, X., Tlili, S., Debbabi, M., 2008. Model-checking for software vulnerabilities detection with multi-language support. In: 2008 Sixth Annual Conference on Privacy, Security and Trust. IEEE, pp. 133–142.

Hecht, G., Mili, H., El-Boussaidi, G., Boubaker, A., Abdellatif, M., Guéhéneuc, Y.-G., Shatnawi, A., Privat, J., Moha, N., 2018. Codifying hidden dependencies in legacy J2EE applications. In: 2018 25th Asia-Pacific Software Engineering Conference. APSEC, IEEE, pp. 305–314.

Janes, A., Piatov, D., Sillitti, A., Succi, G., 2013. How to calculate software metrics for multiple languages using open source parsers. In: IFIP International Conference on Open Source Systems. Springer, pp. 264–270.

Jawawi, D.N., Deris, S., Mamat, R.B., 2007. Software reuse for mobile robot applications through analysis patterns.. Int. Arab J. Inf. Technol. 4 (3), 220–228.

Jenkins, S., Kirk, S.R., 2007. Software architecture graphs as complex networks: A novel partitioning scheme to measure stability and evolution. Inform. Sci. 177 (12), 2587–2601.

Jiang, L., Zhang, Z., Zhao, Z., 2013. AST based JAVA software evolution analysis. In: 2013 10th Web Information System and Application Conference. IEEE, pp. 180–183.

Jinan, S., Kefeng, P., Xuefeng, C., Junfu, Z., 2017. Security patterns from intelligent data: A map of software vulnerability analysis. In: 2017 IEEE 3rd International Conference on Big Data Security on Cloud (Bigdatasecurity), IEEE International Conference on High Performance and Smart Computing (Hpsc), and IEEE International Conference on Intelligent Data and Security (Ids). IEEE, pp. 18–25.

Joorabchi, M.E., Ali, M., Mesbah, A., 2015. Detecting inconsistencies in multi-platform mobile apps. In: 2015 IEEE 26th International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 450–460.

Kanade, A., Maniatis, P., Balakrishnan, G., Shi, K., 2020. Learning and evaluating contextual embedding of source code. In: International Conference on Machine Learning. PMLR, pp. 5110–5121.

Kargar, M., Isazadeh, A., Izadkhah, H., 2020. Improving the modularization quality of heterogeneous multi-programming software systems by unifying structural and semantic concepts. J. Supercomput. 76 (1), 87–121.

Khan, A.A., Keung, J., Niazi, M., Hussain, S., Ahmad, A., 2017. Systematic literature review and empirical investigation of barriers to process improvement in global software development: Client–vendor perspective. Inf. Softw. Technol. 87, 180–205.

Kienle, H.M., Müller, H.A., 2010. Rigi—An environment for software reverse engineering, exploration, visualization, and redocumentation. Sci. Comput. Programm. 75 (4), 247–263.

Kitchenham, B., 2004. Procedures for Performing Systematic Literature Reviews. Keele University, Newcastle, UK.

Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., El Emam, K., Rosenberg, J., 2002. Preliminary guidelines for empirical research in software engineering. IEEE Trans. Softw. Eng. 28 (8), 721–734.

Kitchenham, B.A., Travassos, G.H., Von Mayrhauser, A., Niessink, F., Schneidewind, N.F., Singer, J., Takada, S., Vehvilainen, R., Yang, H., 1999. Towards an ontology of software maintenance. J. Softw. Maint. Res. Pract. 11 (6), 365–389.

Klint, P., Van Der Storm, T., Vinju, J., 2009. Rascal: A domain specific language for source code analysis and manipulation. In: 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation. IEEE, pp. 168–177.

Kolek, J., Rakic, G., Savic, M., 2013. Two-dimensional extensibility of SSQSA framework. In: SQAMIA. Citeseer, pp. 35–43.

Kontogiannis, K., Linos, P., Wong, K., 2006. Comprehension and maintenance of large-scale multi-language software applications. In: 2006 22nd IEEE International Conference on Software Maintenance. IEEE, pp. 497–500.

Lehnert, S., Riebisch, M., et al., 2013. Rule-based impact analysis for heterogeneous software artifacts. In: 2013 17th European Conference on Software Maintenance and Reengineering. IEEE, pp. 209–218.

Li, L., Bissyandé, T.F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., Traon, L., 2017. Static analysis of android apps: A systematic literature review. Inf. Softw. Technol. 88, 67–95.

Linos, P., Lucas, W., Myers, S., Maier, E., 2007. A metrics tool for multi-language software. In: Proceedings of the 11th IASTED International Conference on Software Engineering and Applications. pp. 324–329.

Lozano, A., Noguera, C., Jonckers, V., 2016. Managing traceability links with matraca. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering. SANER, 1, IEEE, pp. 665–668.

Lyons, D., Bogar, A., Baird, D., 2017. Lightweight multilingual software analysis. In: Challenges and Opportunities in ICT Research Projects - EPS Madrid 2017. INSTICC, SciTePress, pp. 35–51. http://dx.doi.org/10.5220/0007904900350051.

Marburger, A., Westfechtel, B., 2010. Graph-based structural analysis for telecommunication systems. In: Graph Transformations and Model-Driven Engineering. Springer, pp. 363–392.

Marchetto, A., Tonella, P., Ricca, F., 2012. Reajax: a reverse engineering tool for ajax web applications. IET Softw. 6 (1), 33–49.

Marinescu, C., Jurca, I., 2006. A meta-model for enterprise applications. In: 2006 Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. IEEE, pp. 187–194.

Mayer, P., 2017. A taxonomy of cross-language linking mechanisms in open source frameworks. Computing 99 (7), 701–724.

Mayer, P., Bauer, A., 2015. 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. pp. 1–10.

Mayer, P., Kirsch, M., Le, M.A., 2017. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. J. Softw. Eng. Res. Dev. 5 (1), 1–33.

Mayer, P., Schroeder, A., 2013a. Patterns of cross-language linking in java frameworks. In: 2013 21st International Conference on Program Comprehension. ICPC, IEEE, pp. 113–122.

Mayer, P., Schroeder, A., 2013b. Towards automated cross-language refactorings between java and dsls used by java frameworks. In: Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools. pp. 5–8.

Mayer, P., Schroeder, A., 2014. Automated multi-language artifact binding and rename refactoring between java and DSLs used by java frameworks. In: European Conference on Object-Oriented Programming. Springer, pp. 437–462.

Mili, H., El-Boussaidi, G., Shatnawi, A., Guéhéneuc, Y.-G., Moha, N., Privat, J., Vatlchev, P., 2019. Service-oriented re-engineering of legacy JEE applications: Issues and research directions. arXiv preprint arXiv:1906.00937.

Misra, J., Annervaz, K., Kaulgud, V., Sengupta, S., Titus, G., 2012. Software clustering: Unifying syntactic and semantic features. In: 2012 19th Working Conference on Reverse Engineering. IEEE, pp. 113–122.

Moise, D.L., Wong, K., 2005. Extracting and representing cross-language dependencies in diverse software systems. In: 12th Working Conference on Reverse Engineering (WCRE'05). IEEE, pp. 10–pp.

Moonen, L., 2001. Generating robust parsers using island grammars. In: Proceedings Eighth Working Conference on Reverse Engineering. IEEE, pp. 13–22.

Moshenska, G., 2016. Reverse engineering and the archaeology of the modern world. In: Forum Kritische Archäologie, Vol. 5. pp. 16–28.

Mushtaq, Z., Rasool, G., 2015. Multilingual source code analysis: State of the art and challenges. In: 2015 International Conference on Open Source Systems & Technologies. ICOSST, IEEE, pp. 170–175.

Mushtaq, Z., Rasool, G., Shahzad, B., 2017a. Detection of J2EE patterns based on customizable features. Int. J. Adv. Comput. Sci. Appl. 8 (1), 361–376.

Mushtaq, Z., Rasool, G., Shehzad, B., 2017b. Multilingual source code analysis: A systematic literature review. IEEE Access 5, 11307–11336.

Nair, S., Jetley, R., Nair, A., Hauck-Stattelmann, S., 2015. A static code analysis tool for control system software. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering. SANER, IEEE, pp. 459–463.

Nguyen, H.V., Kästner, C., Nguyen, T.N., 2014. Building call graphs for embedded client-side code in dynamic web applications. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 518–529.

Nguyen, H.V., Nguyen, H.A., Nguyen, T.T., Nguyen, T.N., 2012. BabelRef: detection and renaming tool for cross-language program entities in dynamic web applications. In: 2012 34th International Conference on Software Engineering. ICSE, IEEE, pp. 1391–1394.

Oliva, G.A., Santana, F.W., Gerosa, M.A., De Souza, C.R., 2011. Towards a classification of logical dependencies origins: a case study. In: Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution. pp. 31–40.

Pang, A., Anslow, C., Noble, J., 2018. What programming languages do developers use? A theory of static vs dynamic language choice. In: 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). IEEE, pp. 239–247.

Perin, F., 2012. Reverse Engineering Heterogeneous Applications (Ph.D. thesis). University of Bern.

Perin, F., Girba, T., Nierstrasz, O., 2010. Recovery and analysis of transaction scope from scattered information in java enterprise applications. In: 2010 IEEE International Conference on Software Maintenance. IEEE, pp. 1–10.

Pfeiffer, R.-H., Reimann, J., Wasowski, A., 2014. Language-independent traceability with lässig. In: European Conference on Modelling Foundations and Applications. Springer, pp. 148–163.

Pfeiffer, R.-H., Wasowski, A., 2011. Taming the confusion of languages. In: European Conference on Modelling Foundations and Applications. Springer, pp. 312–328.

Pfeiffer, R.-H., Wasowski, A., 2012a. Cross-language support mechanisms significantly aid software development. In: International Conference on Model Driven Engineering Languages and Systems. Springer, pp. 168–184.

Pfeiffer, R.-H., Wasowski, A., 2012b. Texmo: A multi-language development environment. In: European Conference on Modelling Foundations and Applications. Springer, pp. 178–193.

Pfeiffer, R.-H., Wasowski, A., 2015. The design space of multi-language development environments. Softw. Syst. Model. 14 (1), 383–411.

Polychniatis, T., Hage, J., Jansen, S., Bouwers, E., Visser, J., 2013. Detecting cross-language dependencies generically. In: 2013 17th European Conference on Software Maintenance and Reengineering. IEEE, pp. 349–352.

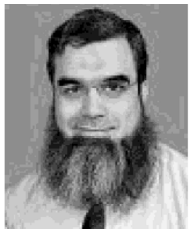
Pribela, I., 2012. First experiences in using so ware metrics in automated assessment.

Rakić, G., Budimac, Z., 2011. Smiile prototype. In: AIP Conference Proceedings, Vol. 1389. American Institute of Physics, pp. 853–856.

Rakić, G., Budimac, Z., 2013. Introducing enriched concrete syntax trees. arXiv preprint arXiv:1310.0802.

Rakić, G., Budimac, Z., Savić, M., 2013. Language independent framework for static code analysis. In: Proceedings of the 6th Balkan Conference in Informatics. pp. 236–243.
Rashid, A., Asif, S., Butt, N.A., Ashraf, I., 2013. Feature level opinion mining of educational student feedback data using sequential pattern mining and association rule mining. Int. J. Comput. Appl. 81 (10).
Saha, A., Denning, T., Srikumar, V., Kasera, S.K., 2020. Secrets in source code: Reducing false positives using machine learning. In: 2020 International Conference on COMmunication Systems & NETworkS. COMSNETS, IEEE, pp. 168–175.
Savić, M., Budimac, Z., Rakić, G., Ivanović, M., Heričko, M., 2013. SSQSA ontology metrics front-end. In: Proceedings of the 2nd Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, Novi Sad, Serbia. pp. 95–101.
Savić, M., Rakić, G., Budimac, Z., Ivanović, M., 2012. Extractor of software networks from enriched concrete syntax trees. In: AIP Conference Proceedings, Vol. 1479. American Institute of Physics, pp. 486–489.
Savić, M., Rakić, G., Budimac, Z., Ivanović, M., 2014. A language-independent approach to the extraction of dependencies between source code entities. Inf. Softw. Technol. 56 (10), 1268–1288.
Schink, H., 2013. Sql-schema-comparer: Support of multi-language refactoring with relational databases. In: 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation. SCAM, IEEE, pp. 173–178.
Schink, H., Kuhlemann, M., Saake, G., Lämmel, R., 2011. Hurdles in multi-language refactoring of hibernate applications. In: ICSOFT (2). pp. 129–134.
Shatnawi, A., Mili, H., Abdellatif, M., Boussaidi, G.E., Privat, J., Guéhéneuc, Y.-G., Moha, N., 2018a. Identifying kdm model of jsp pages. arXiv preprint arXiv:1803.05270.
Shatnawi, A., Mili, H., Abdellatif, M., Guéhéneuc, Y.-G., Moha, N., Hecht, G., Boussaidi, G.E., Privat, J., 2019. Static code analysis of multilanguage software systems. arXiv preprint arXiv:1906.00815.
Shatnawi, A., Mili, H., El Boussaidi, G., Boubaker, A., Guéhéneuc, Y.-G., Moha, N., Privat, J., Abdellatif, M., 2017. Analyzing program dependencies in java ee applications. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories. MSR, IEEE, pp. 64–74.
Shatnawi, A., Shatnawi, H., Saied, M.A., Shara, Z.A., Sahraoui, H., Seriai, A., 2018b. Identifying software components from object-oriented APIs based on dynamic analysis. In: Proceedings of the 26th Conference on Program Comprehension. pp. 189–199.
Srinivas, M., Ramakrishna, G., Rao, K.R., Babu, E.S., 2016. Analysis of legacy system in software application development: A comparative survey. Int. J. Electr. Comput. Eng. 6 (1), 292.
Strein, D., Kratz, H., Lowe, W., 2006. Cross-language program analysis and refactoring. In: 2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation. IEEE, pp. 207–216.
Sultana, N., Middleton, J., Overbey, J., Hafiz, M., 2016. Understanding and fixing multiple language interoperability issues: the c/fortran case. In: Proceedings of the 38th International Conference on Software Engineering. pp. 772–783.
Terceiro, A., Costa, J., Miranda, J., Meirelles, P., Rios, L.R., Almeida, L., Chavez, C., Kon, F., 2010. Analizo: an extensible multi-language source code analysis and visualization toolkit. In: Brazilian Conference on Software: Theory and Practice (Tools Session).
Tomassetti, F., Rizzo, G., Torchiano, M., 2014a. Spotting automatically cross-language relations. In: 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE). IEEE, pp. 338–342.
Tomassetti, F., Rizzo, G., Troncy, R., 2014b. Crosslanguagespotter: a library for detecting relations in polyglot frameworks. In: Proceedings of the 23rd International Conference on World Wide Web. pp. 583–586.
Tomassetti, F., Torchiano, M., Vetro, A., 2013a. Classification of language interactions. In: 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. IEEE, pp. 287–290.
Tomassetti, F., Vetró, A., Torchiano, M., Voelter, M., Kolb, B., 2013b. A model-based approach to language integration. In: 2013 5th International Workshop on Modeling in Software Engineering (MiSE). IEEE, pp. 76–81.
Van Der Storm, T., Vinju, J.J., 2015. Towards multilingual programming environments. Sci. Comput. Programm. 97, 143–149.
Varanasi, B., Belida, S., 2015. Spring Rest. A Press.
Xia, X., Lo, D., Wang, X., Zhang, C., Wang, X., 2014. Cross-language bug localization. In: Proceedings of the 22nd International Conference on Program Comprehension. pp. 275–278.
Yazdanshenas, A.R., Moonen, L., 2011. Crossing the boundaries while analyzing heterogeneous component-based software systems. In: 2011 27th IEEE International Conference on Software Maintenance. ICSM, IEEE, pp. 193–202.
2021. Youtube. https://www.youtube.com/.

Saira Latif received an MSCS (Master of Science in Computer Science) from COMSATS University Islamabad, Lahore Campus (2018-2020), and a BSCS (Hons.) in Computer Science from Capital University of Science & Technology (CUST), Islamabad (2012-2016), She is currently working as a Lecturer at the University of Central Punjab (FOIT). Her interests lie in software development, reverse engineering, multilingual source code analysis, cross-language link detection, and machine learning in software engineering..



Zaigham Mushtaq received the M.S. degree in computer science from the COMSATS Institute of Information Technology, Lahore, in 2010, where he also received the Ph.D. degree in computer science. He is currently working as an Assistant Professor with the Department of Computer Science, The Islamia university of Bahawalpur, Bahawalpur. He was involved in software process improvement and semantic web-based SQL statements. He is involved in design recovery of multilingual applications through recognition of J2EE Pattern. His research interests include source code analysis especially cross language dependence analysis, program comprehension, and source code documentation.

Ghulam Rasool received the M.Sc. degree in computer science from BZU, Multan, Pakistan, in 1998, the M.S.C.S. degree from the University of Lahore, Pakistan, in 2008, and the Ph.D. degree in reverse engineering from the Technical University of Ilmenau, Germany, in 2011. In 2006, he joined the University of Lahore. He has teaching and research experience of 15 years at national and international levels. He is currently an Associate Professor with the COMSATS Institute of Information Technology, Lahore, Pakistan. His research interests include reverse engineering, design pattern recovery, program comprehension, and source code analysis.

Furqan Rustam received the Master of Science degree in computer science from the Department of Computer Science, Khwaja Fareed University of Engineering and Information Technology (KFUEIT), Rahim Yar Khan, Pakistan. He worked as a Research Assistant with the Fareed Computing & Research Center, KFUEIT. He is currently pursuing his Ph.D. degree from University College Dublin, Ireland. His current research interests include data mining, machine learning, and artificial intelligence, mainly working on creative computing and supervised machine learning.

Naila Aslam received the B.S. and M.S. degrees in computer science from The Islamia University of Bahawalpur, Pakistan. She is currently pursuing a Ph.D. degree from the Hebei University of Technology, Tianjin, China. Since, she has been working as a Lecturer with the Department of Computer Science, University of Management Technology, Lahore. Her main research interests include artificial intelligence, data mining, and machine learning.

Imran Ashraf received his Ph.D. in Information and Communication Engineering from Yeungnam University, South Korea in 2018, and the M.S. degree in computer science from the Blekinge Institute of Technology, Karlskrona, Sweden, in 2010 with distinction. He has worked as a postdoctoral fellow at Yeungnam University, as well. He is currently working as an Assistant Professor at the Information and Communication Engineering Department, Yeungnam University, Gyeongsan, South Korea. His research areas include positioning using next-generation networks, communication in 5G and beyond, location-based services in wireless communication, smart sensors (LIDAR) for smart cars, and data analytics.