Master Thesis

December 16, 2021

Overcoming the gap between structured dependencies and change coupling

Towards an Incremental Call Graph Generator from source code

Gabriela Eugenia López Magaña

of No Home given, <Mexico> (17-744-830)

supervised by

Prof. Dr. Harald C. Gall
Dr. Carol Alexandru

Dr. Sebastiano Panichella

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Master Thesis

Overcoming the gap between structured dependencies and change coupling

Towards an Incremental Call Graph Generator from source code

Gabriela Eugenia López Magaña

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Master Thesis

Author: Gabriela Eugenia López Magaña, gabrielaeugenia.lopezmagana@uzh.ch

URL: https://www.ifi.uzh.ch/en/seal.html

Project period: 16.06.2021 - 16.12.2021

Software Evolution & Architecture Lab

Department of Informatics, University of Zurich

Acknowledgements

I would like to thank...

- Prof. Dr. Harald C. Gall for allowing and approving this thesis.
- Dr. Sebastiano Panichella for co-supervising this thesis, providing invaluable support, feedback and guidance, continuously. Especially for his motivation to disseminate and transmit the scientific approach to software engineering to his students, industrial partners and the general public.
- Dr. Carol Alexandru for co-supervising this thesis, his expert opinions, supply of related work and feedback on methodology and the shape of the thesis.

To my family and friends for their support, care and encouragement.

Abstract

In this thesis, we conduct an empirical analysis of the history of object-oriented complex cyber-physical systems to discover contextual metrics computed on top of software changes, based on call graph analysis and evolution of software entities (e.g., call graph changes of a given calling functions at a specific commit, release, or time frame). These metrics serve as proxies to measure how high (or low) the change coupling of subsequent software changes will be. Additionally, they should reveal if the coupled changes happened within the call graph or outside of it.

Specifically, we conjecture that such metrics are valuable indicators of complex types of changes that directly impact the maintainability of the system code.

For the support of our investigation and future research, we developed an automatic approach to compute the designed metrics. To validate our research questions we carry on a case study involving four open-source projects in the domains of house automation, health devices, small robots controlling, and robot real-time visual processing.

The results of this study highlight how these metrics, accompanied by a user-friendly tool, provide to practitioners quantitative views of dependencies and evolution, based on call graph analysis.

As future work, we plan to quantitatively and qualitatively assess the change-proneness exposed by our metrics in further projects and organizations from different industrial domains.

As motivation for this thesis, we inquire three fundamental research questions:

- *RQ1: To what extent is it possible to build evolutionary call-graphs based on software version management information?*
- *RQ2: What is the relation between structural coupling (on a function level) and the call-graphs software evolution?*
- *RQ3: Is there a relation between conceptual (non-structural) coupling and the call-graphs software evolution?*

Zusammenfassung

In dieser Arbeit führen wir eine empirische Analyse der Geschichte objektorientierter komplexer cyber-physikalischer Systeme durch. Durch Analyse von Aufrufdiagrammen und durch Betrachtung der Evolution von Software-Entitäten (z. B. Änderungen des Aufrufdiagramms einer bestimmten aufrufenden Funktion bei einem bestimmten Commit, Release oder Zeitrahmen) hoffen wir kontextuelle Metriken zu entwickeln.

Diese Metriken dienen als Proxys, um zu messen, wie hoch (oder niedrig) die Änderungskopplung nachfolgender Softwareänderungen sein wird. Außerdem sollen sie zeigen, ob die gekoppelten Änderungen innerhalb oder außerhalb des Aufrufdiagramms stattfanden.

Insbesondere vermuten wir, dass solche Metriken für komplexe Arten von Änderungen wertvolle Hilfestellung bei der Bewertung der Wartbarkeit des Systemcodes bieten können. Um unsere Untersuchung und zukünftige Forschung zu unterstützen, haben wir einen automatischen Ansatz zur Berechnung der entworfenen Metriken entwickelt. Dann haben wir unsere Forschungsfragen anhand einer Fallstudie validiert, die vier Open-Source-Projekte in den Bereichen Hausautomatisierung, Gesundheitsgeräte, Steuerung kleiner Roboter und visuelle Echtzeitverarbeitung von Robotern umfasst. Die Ergebnisse dieser Studie zeigen, wie diese Metriken Anwendern(??) quantitative und auf Aufrufdiagrammen basierende Sichten von Abhängigkeiten und Evolution zur Verfügung stellen können, die aufbereitet durch ein anwenderfreundliches Tool eine Bewertung der Codequalität zulassen und so zu einer Verbesserung der Effizienz führen können. Als zukünftige Arbeit planen wir die quantitative und qualitative Bewertung der Änderungsanfälligkeit, die durch unsere Metriken in weiteren Projekten und Organisationen aus verschiedenen industriellen Bereichen aufgedeckt wird. Als Motivation für diese Arbeit stellen wir uns drei grundlegende Forschungsfragen:

- *RQ1: Inwieweit ist es möglich, evolutionäre Aufrufdiagramme auf Basis von Software-Versionsmanagement-Informationen zu erstellen?*
- *-RQ2: Welcher Zusammenhang besteht zwischen struktureller Kopplung (auf der Ebene von Funk-*

tionen) und der Software-Evolution der Aufrufdiagramme?

- *-RQ3: Gibt es einen Zusammenhang zwischen konzeptioneller (nicht-struktureller) Kopplung und der Evolution von Aufrufdiagramm-Software?*

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Complex systems investigated in the context of this thesis are systems composed of many, heterogeneous components that interact with each other and the system environment. In the case of *software systems*, such components and its related resources can have a different nature, such as microservices, compiled code, databases, external libraries called via APIs, etc. In this context, Cyber-physical Systems (CPSs) represent a more complex set of systems, which present both hardware and software components and make physical tasks/decisions on the basis of data from sensors, event-camera from the surrounding environment. CPSs characterize a large set of critical domains from space exploration, smart home, autonomous driving systems (e.g., drones and self-driving cars), e-health systems and devices [3].

The interaction of a CPS with hardware devices, as well as with humans [4–6] as well as other systems, makes the nature and effect of faults in CPS environments very specific and non-predictable [7, 8]. For example, a broken sensor [9] or a security attack [10] can lead to (unexpected) inputs that mislead autonomous systems' behavior. Hence, CPSs behaviour is intrinsically more difficult to monitor, due to the dependencies and interactions of their components with the environment. For example, a large range of inputs from the real world and that affect or impact CPSs' components might include noise duo to the seamless connectivity for IoT devices and the hardware heterogeneity [11]. This complexity hampers the reproduction of test scenarios because the environment constantly changes, making it hard to recreate the relevant or original conditions [12].

Similar to any other software system, CPSs evolve through frequent changes. Due to their complexity, a single change can alter many parts of the system and, as consequence, its behaviour. If one of the modules presents a defect and is not tested correctly, this defect might be executed in the production environment, causing a failure in the system's behaviour and potentially generating damages to the environments or the humans involved in such environmental context. This is indeed and important aspect of such systems: CPSs interact with humans and the environment in the real world, and defects can lead to fatal damages sometimes causing also humans fatalities [13–15].

Software maintainability is essential to deliver dependable systems. Chen *et al.* [1] showed in an empirical study, based on the ontology of Software Qualities (SQ) defined by Boehm, *et al.*

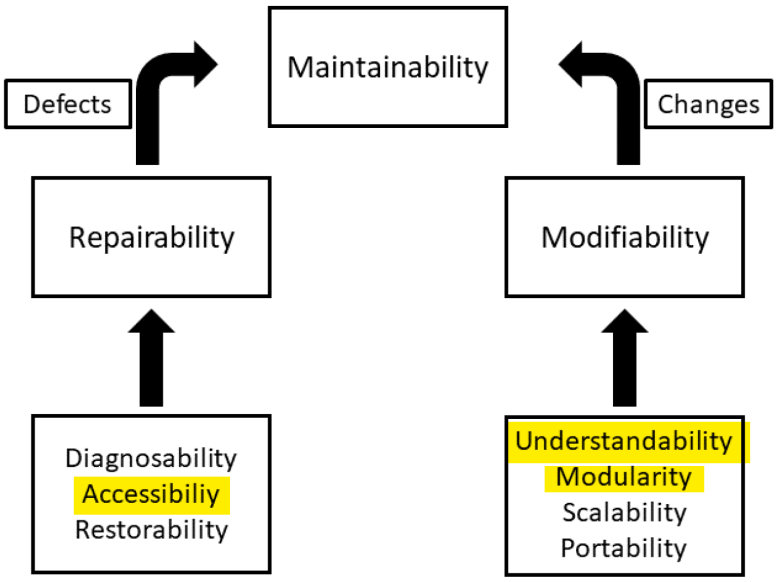


Figure 1.1: Maintainability Hierarchy (Chen *et al.* [1])

[16, 17] that high-severity bugs contain more maintainability issues, as compared to low-severity ones. In Figure 1.1 we show the relationships of the Software Qualities subgroups. Chen *et al.* found that some subgroups present larger failure incidences. For example, *modularity* issues presented the highest number of source bugs and were mostly dependent on bugs from different modules. *Accessibility* and *understandability* [18, 19] provoke the largest number of reopened bugs [1]. Regarding the causes, these issues are predominantly caused by failures in the implementation of data design, usage and functionality.

Dependability in the software development process is a crucial quality to reduce the risk of failures and damages. Developers need to understand the source code they are working with, and this code comprehension tasks on average takes up to 58% of their time [18, 20, 21]. Furthermore, developers and other stakeholders are required to understand the impact of the introduced changes in the system. Managers typically allocate resources in areas where more defects or severe ones are foreseen [22]. Often this resource allocation is based on the manager or the team's experience.

Chen *et al.* [1] showed that accessibility, understandability and modularity are key Software Qualities highly related to maintainability issues and bugs, and many studies have proven that complexity increases defect-proneness [23–26]. Like explained in the section of Related Work, researchers have searched for indicators and created prediction models to detect quality deficiencies, but did not found a universal metric, and one shortcoming is that most of the complexity metrics focus on analyzing single elements. In addition, several studies have shown that process metrics can accurately predict defects in source code [27, 28].

Since the 1970's there has been applications using call graphs on compiled code mainly to analyze data flows and analyzing the behaviour of the variables and calls [29] and research effort

have been focused on improving the precision of the algorithms that generate the call graphs [30,31].

Based on the works of Watts and Strogatz (1998) and Albert & Barabási (2002) [32] in the field of complex networks, network science and social networks researchers started applying this concepts to exhibit properties of software systems [33,34].

In recent years the study of graph, or network, analysis has been applied to software engineering to address different challenges. Some challenges are descriptive, like measuring software complexity, detecting communities [35]; and some others address a specific problematic in the software code, such as malware detection [36], clone and similarity detection [37] and unreachable code detection [38].

1.1 Motivation

In Chapter 2 we discuss the current work on software evolution and analysis of software as networks. We highlight the gap between the subjects of analyzing code changes and studying the graph structures of the software. Specifically, the changing of one line of code might have a very different impact in the software behaviour, depending on *where* in the system this change happens. We believe that analyzing the incremental changes of the underlying call graphs of the system, helps to yield an overview of the system as it evolves and shows how its different components are impacted by changes. We believe that this complementary view of software evolution might support the understanding testers' and developers' maintainability and risk assessment tasks.

Much of the research and tools on software evolution are based on a meticulous analysis of the code, either static or dynamic analysis, and the usual main goal is to supports the program understanding by focusing on the information's need of the developer. But in industrial scenarios, specially in complex systems, there are other stakeholders that need different levels of granularity, summarizing and abstraction [39]. Furthermore, many studies have focus their attention in analyzing and predicting failures [40]. For this objective an accurate and representative database of bug reports is necessary, but from our experience in the software development process, as reported by Ambros *et al.* [40], defect corrections might be reported in the code change as new or updated features and not always written in the form of bug reports [41]. Additionally, inadequate management of requirements at the moment of programming are often not transparent, and they repeatedly might represent non-crashing bugs; so, their behavioural flaws are often not tracked as software failures. Because of these reasons we focus on change evolution and not in defect prediction.

1.2 Goal

The goal of this thesis is to design and implement a tool for the analysis of software systems, which complements existing change analysis tools, and integrates the ongoing research on network structure of the code, this via the understand of the system evolution through analyzing the

history of changes. The tool we implemented in this thesis, called *callgraphCA*, is designed to be easily adaptable to different programming languages and repository sources. This is an important requirement since we aim to provide information to different stakeholders, that might not have a matching code compiler or, researchers that might be interested in comparing different systems written in different languages. The *callgraphCA* tool does not require the compilation of code, since it is based on openly available information. Specifically, we validated our tool in CPS relevant domains, involving four CPS open source projects, that fulfills our requirements of data quality, size and availability. The *callgraphCA* specifically models methods and function calls, in a deeper granularity level than most of the literature studies, and analyze their relations to change coupling. The focus on this research direction, since we aim to identify change behaviours, with an emphasis in change coupling. This is because requirements and design errors or omissions are often not documented in socio-technical repositories (e.g., in a bug reports of issue tracking systems), and might only be discovered until late testing phases or post-release [26, 40, 42].

We aim to define complexity metrics that allow to monitor software changes based on call graph analysis of the changed functions and detects the level of change coupling of subsequent software changes. The metric will also show if the changes happened within the call graph and outside of it, representing respectively structural and conceptual or logical changes. We conjecture that this complexity impacts the maintainability of the code and we will evaluate this by assessing the change-proneness exposed by the changed call graphs. The usefulness of this metric lays in supporting people assessing the risks of software changes, and who posses a certain amount of experience. We aim to analyze how transferable our approach is, when applied to different projects [23].

1.3 Research questions

In this thesis, we conjecture that the proposed *contextual complexity* metrics are valuable indicators of complex types of changes that directly impact the maintainability of the system code. To investigate this assumption, we empirically investigated three fundamental research questions.

The **research questions** that guide this thesis are:

- *RQ1: To what extent is it possible to build evolutionary call-graphs based on software version management information?* For bench-marking the reliability of our solution we propose to use SourceTrail as an oracle to detect the graph structure of the code and to compare our outputs.

- *RQ2: What is the relationship between structural coupling (on a function level) and the call-graphs software evolution?* We propose to investigate how many times commit changes on a *function A* impact the functions that are connected in the form of method calls to it (i.e., the functions that are calling *function A* as well as the ones that *function A* calls)..

- *RQ3: Is there a relationship between conceptual (non-structural) coupling and the call-graphs software evolution?* Change-coupling consists in co-changes of functions in the same time win-

dow. In this context we propose to investigate the relations between change-coupling and call-graph changes (e.g., How many times co-changes of functions impact a call graph?).

1.4 Contribution

In this thesis we propose a prototype tool, *callgraphCA*, for the empirical analysis of changes in the context of complex (e.g., -physical) systems, with the focus of providing overview metrics that have the following characteristics:

- they combine graph analysis with code change analysis
- they analyze the system's call graph of the changed functions
- they can be used as proxy to measure how high the change coupling of subsequent software changes is within the call graph and outside of it
- they can be computed at commit or release (e.g., Github tags) level
- they can be applied to different programming languages

1.5 Outline

This thesis is structured in eight parts. We are motivated to discover the relationships between change proneness and the underlying network structure of the software based in call-graph analysis. In Chapter Section 2 we discuss related work in the areas of analysing software evolution, and applying network analytics to the understanding of software. In Chapter Section 3 we present the methodology of this thesis, the main concepts for this research, the challenges when implementing some algorithms and presenting the selection of systems under study 3.3. Our motivation is to automatize and facilitate the access to our analytics approach, we present our tool *callgraphCA* In Chapter Section 4. In Chapter Section 5 we present the results of the study, and discuss the shortcomings of developing complex systems for the study of software evolution. In Chapter Section 6 we list threats to validity and in Chapter Section 7 we present the limitations of our approach and future work. Chapter Section 8 closes this part of our work.

In this work, we use the word *function* as an umbrella term to refer to actual functions as well as methods from classes.

Chapter 2

Related Work

In 1999 Kemer and Slaughter [43] explained that the discipline of analyzing software evolution dates back to the 70's, which has the main goal of understanding software evolution dynamics, provide measurable metrics of software evolution [44], to predict or avoid future failures in the code [45–49], and to improve users satisfaction on software systems [5, 50]. Many different fault (or defect) prediction techniques have been proposed such as analyzing software evolution, using change logs, versioning history, code metrics from static analysis, cyclomatic complexity, object-oriented metrics, process metrics, differences between binaries and past bugs history [23, 25].

As explained in the Chapter 1, we propose an empirical analysis of software evolution based on repository commit changes, and call graph analysis. Therefore this section discusses the literature concerning (i) previous work on software metrics in the context of defect prediction; (ii) recent studies on graph analysis of software systems, as well as (iii) change coupling analysis.

Software Metrics for Defect Prediction

A number of metrics, including object-oriented metrics suggested by Chidamber and Kemerer (CK) [51, 52], process metrics [53] and complexity of change metrics [54] have been proposed to predict bugs. A comparative study of several machine learning approaches (e.g., Logistic Regression [55], Naive Bayes , Regression Trees, K-nearest neighbours Decision trees such as J48) proposed to predict bugs on top of such metrics have been presented by D'Ambros *et al.* [47]. Khoshgoftaar *et al.* [56] has used Neural networks as alternative algorithm for bug prediction.

Complementary to such previous studies, more recent work investigated the use of different metrics or different strategies to improve applicability of proposed metrics in different context. Gyimothy *et al.* [52] presented a comprehensive study on the use of code metrics for defect prediction, while Menzies *et al.* [48] revealed that a high precision of prediction models cannot be achieved using just a single project. Hence, Zimmermann *et al.* [57] performed a large scale experiment to determine the best way to improve accuracy of cross-project defect prediction strategies.

Ball *et al.* [58] experimented with the usage of code-churn as a strong indicator of errors, while Pinzger *et al.* [59] provided a retrospective comparison of fine-grained source code changes and code churn metrics for bug prediction. Khoshgoftaar *et al.* [56] proposed a code churn based approach that counts the lines of code inserted or removed for fixing a bug in a program file. Recent

works proposed approaches where defect prediction as modeled as a multi-objective optimization algorithms [60, 61], conducted in-depth assessments of using traditional metrics (e.g, class size) in software defect prediction [62], discussed agile methods for defect prediction [63], and proposed bug prediction approaches leveraging source code embedding based on Doc2Vec [64].

Software evolution Metrics for Change coupling

Zimmermann *et al.* [55] and Nachiappan *et al.* [65] experimented with evolution (or historical based) approaches mining software archives to compute code related metrics as well as process and historical metrics, with the goal of predicting fault-prone modules. Since then, several tools [66] for computing software change metrics [67,68] have been proposed and studies investigating software changes have been conducted [68–70].

In this context, the term *change coupling* was introduced by Fluri *et al.*and Gall [69] [71] and reflects an evolutionary dependency of two artifacts that frequently change together and might not be structurally linked. Furthermore, this couplings might reflect logical dependencies that should be drafted at the moment of designing the changes and testing strategies. Hassan and Holt [72] demonstrated that change coupling analysis delivers high performance in predicting change propagation. Hence, Zhou *et al.* [73] proposed a bayesian network based approach for change coupling prediction. Oliva and Gerosa [74] explained several advantages of analyzing change coupling when studying a software system. Change coupling analysis reveals evolutionary relationships that might not be discovered by structural coupling analysis and might be better suited for certain applications. As consequence, Poshyvanyk and Marcus [75], and other researchers [76–80], investigated and proposed conceptual and logical couplings metrics for object-oriented systems and other types of software systems.

Network and Graph Analysis for Software Engineering

Network science is defined as *"the study of network representations of physical, biological, and social phenomena leading to predictive models of these phenomena"* [81]. In this context, a call graph is a representation of calling relationships between the methods in a software system [82]. The graph is composed by nodes, that represent the functions or methods, and the directed edges represent which function calls another. Function Call Graphs (FCG) provide syntactic, topological and symbolic information of the system [37].

Researchers, in recent years, have used the approach of analyzing software systems as graphs or networks. Alexandru et. al. developed an approach for analyzing the fine-grained history of software projects by merging graph representations of thousands of program revisions and computing metrics on the merged graph representation [83]. In a sample of 80 software systems written in Java and C++, Valverde and Sole [34] analyzed the class membership of methods and subclasses and found that dependencies in such systems present the same small world behaviour discovered in many natural and human-made systems, such as the internet hubs and social networks. Another important characteristic, discovered by Potanin *et al.* [33] when studying object-oriented programs, is that graphs of objects show also the scale-free network properties. Here, they evaluated the objects created by the program at run time, representing the nodes. Such

graphs change constantly through the execution of the program, but the behaviour is that the vast majority of the objects have low connectivity, and there is a relatively small number of high-connectivity nodes, or “hubs". In such hubs the vast majority of nodes are poorly connected to the rest of the graph [84–86]. As consequence, deleting some of them can have negligible effect on the connectivity of those remaining, while deleting some others (e.g., the one very highly connected) can be far more destructive. Such behaviours can be relevant for approaching software engineering and development budgets, because they imply that concentrating debugging approaches on more relevant well-connected objects could be more effective, for instance when eliminating bugs from the most important hubs [84, 87].

Researchers have quickly adopted the approach to study software systems under the concept of small-world properties, for example, when analyzing the interactions and dependencies between classes, methods, packages and libraries [85–87]. However, most tools developed based on such concepts consists mainly in visual exploration of dependency graph in source code [88, 89] (e.g., via embedding-based similarity [90]), with very few approaches proposing tools for a graph dependency analysis of software systems [91], and, to our best knowledge, focusing on the current state of the software and having no link to the evolution of the system.

Chapter 3

Methodology

This chapter discusses the study methodologies of this thesis. In Section 3.1 we provide an overview of the change coupling definitions available from the literature, then we discuss which definition we used in the context of our work. In Section 3.2 we explain the pillar concepts and challenges of analyzing software for call graph evolution that will need to be implemented for answering our research questions. Finally, in Section 3.3 we describe the targeted systems, the data collection process applied, including the the methodology followed to select them.

3.1 Change coupling

In this section we describe the concepts behind the definition of traditional coupling metrics, possibly describing how they complement each other, then, we discuss which metrics we used in the context of our work.

Oliva *et al.* [74] defined that a dependency between two elements is a directional semantic relation where changes performed in an element may affect the other element in the relation. In previous work, this dependency relations have been separated in four main categories:

- *structural coupling*: two code elements are structurally coupled if code/structural dependencies exist among them [92]. Higher is the number of dependencies among these code elements, higher is the level of coupling. A low structural coupling is important to allow changes in an individual code entity without propagating them in other part of the system.

- *semantical coupling* is based on interpretation of *"meaning"* of unstructured text in the source code. Ajienka2018 *et al.* [93] points out that for some authors this relations are reduced to the similarity of identifiers, for some others the similarity is extended to comments and even to domain concepts and features that can be extracted from the artefact. More formally, a software module (or artifact) A is semantically coupled with another module B, if their code are semantically similar [93], i.e., they talk about the same concepts (i.e., report similar terms). This semantic similarity is measured by means of information retrieval techniques, such as Vector Space Model (VSM), or Latent Semantic Indexing (LSI) [94–96], that generates similarity values between 0 (min. similarity) and 1 (max. similarity). Thus, higher is the

number of similar artifacts (e.g., similarity measure > 0.75, where 0 is the min. value and 1 is the max value possible) among these modules, higher is the level of coupling among them. The rationale behind the concept of semantic coupling is that if two artifacts are similar (e.g., there are textual code clones among them), then, they tend to experience similar changes. Thus, in this case, in the corresponding collaboration graph, the nodes correspond to the system modules and the weighted edges represent the number of semantically similar artifacts among these nodes. Also in this case, a low semantic coupling is important to allow changes in an individual module without propagating them in other modules (high maintainability). Vice versa, a high semantic coupling led to bugs and changes propagation among modules of a system (low maintainability).

- *logical coupling*: Co-evolution of classes can be represented with their change, logical or evolutionary coupling [97] therefore, the logical coupling of any two classes is based on their change history, and is a measure of the observation that the two classes always co-evolve or change together [97,98]. More formally, a software module (or artifact) *A* is conceptually or logically coupled with another module *B*, if their code tends to change in the same commit or temporary closed commits [75], i.e., tend to co-change together or in close time intervals. This conceptual coupling is measured by means of historical information from commit repositories. Thus, higher is the number of times artifacts co-change together among two modules, higher is the level of coupling among them. The rationale behind the concept of conceptual coupling is that if two artifacts are conceptually related (e.g., they tend to co-change often together), then they tend to experience conceptually related changes. Thus, in this case, in the corresponding collaboration graph.

For our research, we will focus on the structural and logical coupling of software artifacts. Previous approaches to discover change coupling in software engineering have presented different algorithms, granularity level and a wide palette of information presenting, that matches the diverse purposes of the research and targeted stakeholders. The basic idea to detect coupling is to discover association rules between items happening in a large set of events, or *transactions*. The usual metaphor to explain this relationships is the supermarket basket, where the aim is to discover which set of objects that are bought implies the buying of a third one: *people who bough milk and bread also bought eggs*. In the original definition by Agrawal *et al.* [99] there are two sets: I containing *n items*, and a set *D* of *transactions* called database. Each transaction has a unique identification and contains a subset of items, called an itemset. In data mining, the number of items in a transaction is expected to be small in comparison to the number of existing items. A set of items or items set is frequent if it is contained in many transactions. The *support* of the itemset I indicates the number of transactions in which is contained. Association rules are created to understand the relationships of itemsets in the universe of transactions. An association rule I J, states that when I is present in a transaction, J will likely be present. Association rules are directed, the left-hand-side of the rule (LHS) is the antecedent and the right-hand-side (RHS) is the consequent. In a rule {A, B} $\rightarrow$ {C} the antecedent A and B denote that if a transaction contains A and B, it is likely that it will also contain C. The number of transactions containing both A and B is the support (A $\rightarrow$ B) = support(A$\cup$B). A useful concept is *confidence*, which is basically the

percentage of all transactions satisfying X that also satisfy Y. In our field of research of software systems evolution, items represent changes in the source code and the transactions can be defined as process events or time windows. The coupling is stronger when the set of artifacts are changed together and not independently from each other. Support and confidence and other metrics can be applied to discover coupling between items, but in practical applications the focus is on the discovery of many sets of items that appear frequently in the given database and are relevant to make decisions or reflect behaviours of the system, this is called association rule learning. In 1994 Agrawal *et al.* [100] presented an implementation for a fast algorithm for mining association rules. The Apriori algorithm is frequently used because it reduces processing by preemptively pruning of non-frequent (irrelevant) items that do not satisfy a required support threshold. The apriori algorithm requires a parameter for threshold. In the field of software engineering, finding a relevant threshold for the association rules mining has proven to be a complex topic. Depending on the systems under study and the purpose of the research, different opinions have been documented. Zimmermann *et al.* [97] studied coupling between files in the C language and included both the main source files and header files. Having this constellation of files greatly increases the confidence of the association rules, hence he and his team set the, comparatively, high support threshold of 1 and a confidence of 0.5. On the other hand, after studying the perception of coupling by the software developers, Bavota *et al.* [101] set much lower thresholds, where they included elements that co changed with minimum support of 0.02 and confidence larger than 0.8. For mining for association rules, in our research, we will use the *apriori* algorithm. The main reasons for this decision are its efficiency, the hability to process transactions containing irregular numbers of items, and because it searches for both LHS and RHS rules. We will also provide a practical solution for finding adequate support thresholds for the different systems under study.

3.2 Call graph evolution

The concept of call graph evolution can be understood as the adding and deleting of function and function calls within the system. Because specific ranges of the repository history can be analyzed, we traverse the commit history in inverse time order. In this section, we use the terms

- *start_hash*: referring to the commit hash and commit date where the function call was added, meaning, the first time that Function *A* called Function *B*,
- and *end_hash*: referring to both the commit hash and commit date where Function *A* stopped calling Function *B*.

all of this fields are saved on the database tables.

In Figure 3.1 we have a simple example of *File1*, in which the function Function *FX* is calling functions Function *{A,B,C D, E}* at some point in time. Function A is always present during the whole history under analysis, from commit *'h-4'* to commit *'h2'* . Function B was added at commit *'h-3'* and remains until commit *'h2'*. When stopping, at a given commit in the history, the values in the database are the ones matching the commit color. When analyzing each previous commit, the *start_hash* are updated to the oldest commit in which the function call was existing in the

file, respectively, being called by function Function *FX*. We can observe that, when the function is no longer existing in the current commit, the *end_hash* will be set to show the date on which the function call was removed. Because of the inverse order, each processing cycle updates the *start_hash* of the, at this time, existing function calls. So we can see that the initial insert of the function Function *A* is done at the time of analyzing *h2* and because it is still existing, the *end_hash* is set empty (*()*). When analyzing the previous state, at commit *'h1'* the *start_hash* is set to *h1* and so consecutively until reaching the oldest commit *h-4*. An important case is when functions are deleted in previous commits but reinserted later. We cover this case by making sure that current commits only update the *end_hash* on entries that do not already possess an *end_hash*, meaning, that they were deleted by more recent commits. In this thesis, we refer to added and deleted software elements, like files, functions, methods and function calls referring to added or deleted with respect to the previously stored version, generally the previous commit.

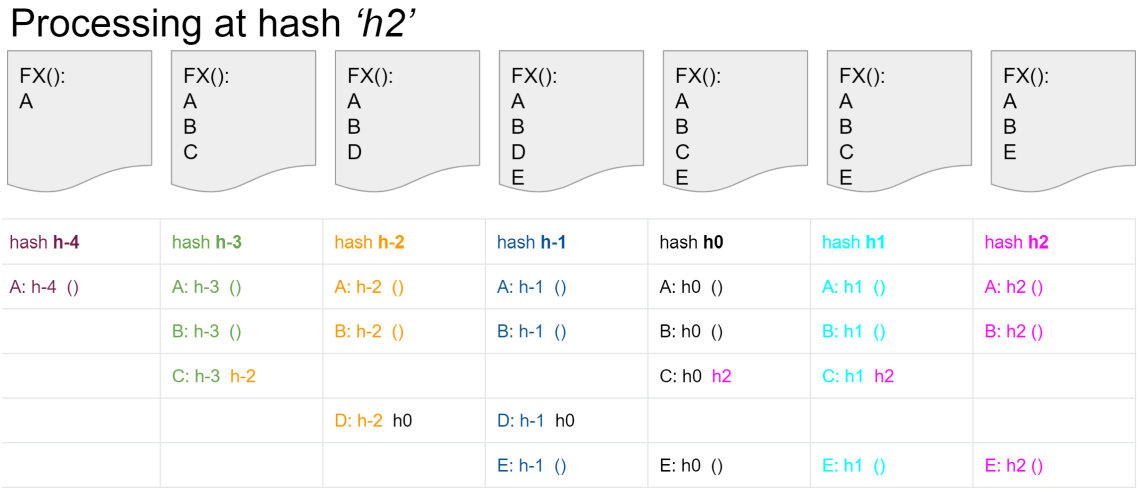


Figure 3.1: Evolution of function calls within a file, respectively function. Next to the called function name is the starting commit hash or *start_hash* and the ending commit hash or *end_hash*.

3.2.1 Light-weighted graph creation

Defining call graphs

We define a call graph as a directed graph $D = (N, C)$. Where N is a set of methods or functions (nodes), and E is the set of relationships (edges). For our study, we defined the relationship as a method call $C = (ni, nj)$ that reads as "ni calls nj". For this research we approach the retrieval of identifiers by two methods, the first time is by matching of artifact identifiers and the second is by parsing of Abstract Syntax Trees (AST).

Artifact identifiers. Ajienka *et al.* stated that when analysing the corpora of the software classes the identifier could be retrieved for classes with strong semantic coupling. We proposed an approach of matching identifiers according to their package tree locations and matching over parameters and arguments. Additionally, an initial parsing of the source code to generate the relationships between file, package and method should support the matching accuracy.

AST parsing. Abstract syntax trees (AST) are tree representations of the syntactic structure of code. They represent the code structure in a higher level than the compilator trees and don't include every element of the code, for example, punctuation elements like parenthesis, etc. AST parsing is the action of visiting the nodes of the tree to retrieve relevant information, and it is usually employed for semantic analysis. We employ parsing libraries like *srcML* for parsing the tree and transform it into other readable representations like xml. Based on heuristic rules we extract xml nodes that we use to build our model representation of the system. The model we propose for our research is similar to the metamodel FAMIX of Moose *et al.* [102] Figure 3.2. The part of our research that deals with call-graph analysis implements a model that contain methods, invocations or calls, packages and the package hierarchy and import relationships between the files.

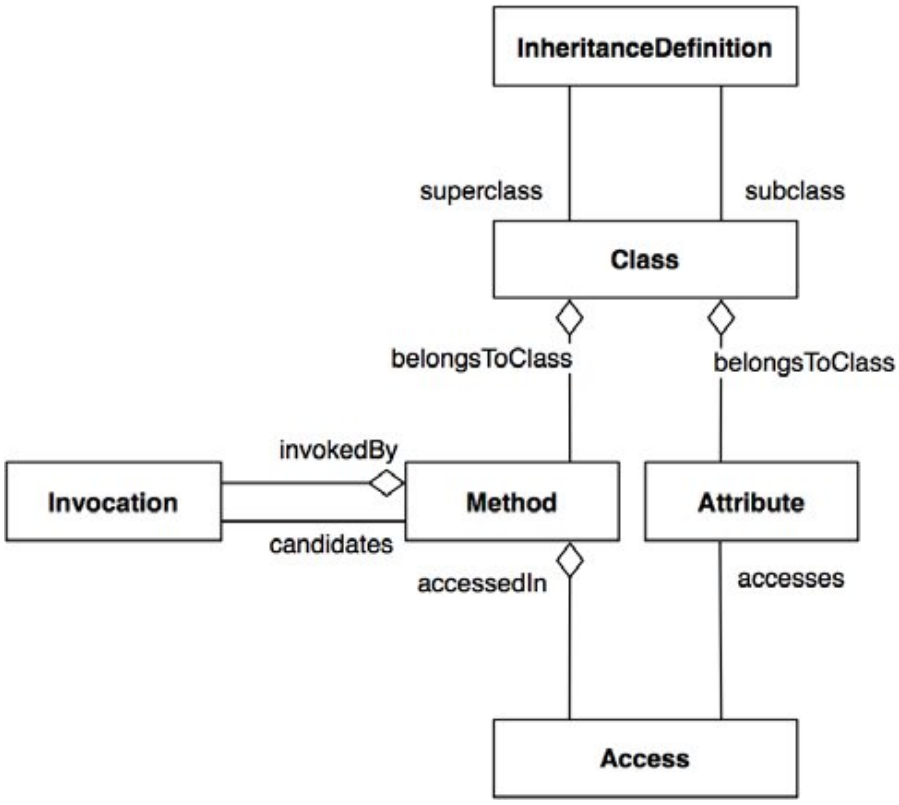


Figure 3.2: Famix Model (Lanza. [2])

3.2.2 Challenges

Stability. As previously explained in the Call Graph evolution section 3.2, the *start_hash* and *end_hash* are updated inverse time order, starting from the newest commit in the range. Because software systems are in constant evolution, it might be necessary to analyze the project consecutively with *callgraphCA* to keep the information updated. In Figure 3.3 we can observe that the first processing was done at *'h-1'*, and a consecutively processed at commit *'h2'*. The methodology and the tool must ensure that independently of the range of changes being analyzed, even if overlapping, the most accurate and complete information will be stored. This leads to the challenge to only update the *start_hash* at the, at this point, oldest known occurrence of the function call. In our example, when processing from *'h-1'* we can see that functions Function *B, C, D* and Function *E* were all inserted during the analyzed time period; however, because the commit range starts at *h-4*, we cannot know if function Function *A* existed previously or not, hence we set the *start_hash* to *h-4*. When processing the system at *h2* we need to make sure not to overwrite the known oldest insert times of the function calls. So for functions Function *A, B* and Function *E* we take the oldest existing *start_hash*, based on the commit date, and keep it for the currently existing function call, in the example, at commit *h2* the functions Function *A, B* and Function *E* become the oldest known *start_hash*.

When analyzing *h1* we find that function Function *C* was added, in relation to *h2*, however, this function was already existing previously and hence there is a record of it. The algorithm must make sure that it only updates function calls that have not been previously *closed*, meaning that the function was both added and deleted in the past. When this case arises, the algorithm must insert a new record for this function, like for Function *C* at hash *h1*.

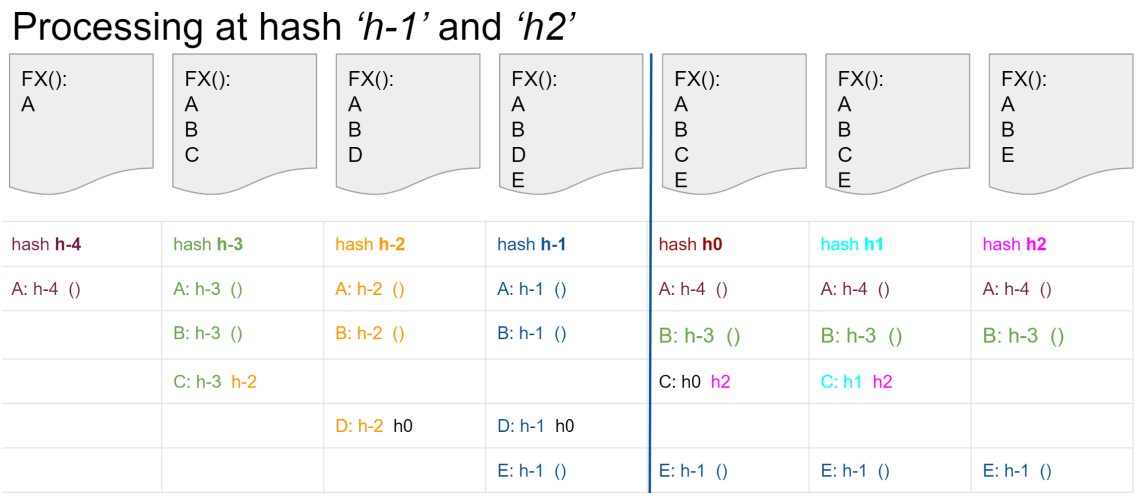


Figure 3.3: Secondary processing of the same project at commit hash 'h2'.

3.3 Systems selection

In the following section, we detail the methodology adopted to select the projects to answer our research questions. Specifically, for this master thesis we used data from Git repositories. For this purpose we chose projects in Github[1], whose characteristics are summarized in Table 3.1. We searched for relevant projects, as detailed later in this section, considering a specific selection criteria, tailored to the domains of cyber-physical systems. Using the GitHub query search feature, we applied the following selection criteria based on a "criterion sampling" [103], ending up with a sample of four projects.

- *Domain selection*: We focus on software that supports cyber-physical systems (CPSs). Since our goal is to identify projects belonging to different CPS domains, we experimented with specific GitHub queries: *"IoT"*, *robot*, and *health*. In addition, we also considered *home-automation* to explicitly target projects aimed to design and manufacture hardware and software for smart homes.
- *Project Popularity*: We sort the results by stars to focus on popular repositories. Note that, while selecting projects solely based on stars has been considered inaccurate [104], this selection criterion is enhanced with other criteria.
- *Language selection*: We selected projects having as object oriented languages main programming languages, mainly Java and C++, since, while querying GitHub for projects belonging to different domains, we realized that most of them use those languages. While the choice of Java can be considered obvious, the selection of C++ projects is pretty consistent with the finding from previous work that shows that most CPS development is performed in C++ program language [105], however, for the results presentation of this thesis, we concentrate on Java systems, and in the tool limitation section 7.0.1 we explain our decision.
- *Projects size*: Due to the scope of the project, we searched for systems that were not larger than 1.5 million LOC.

Below is a description of the selected projects, detailed in Table 3.1:

- **Glucosio for Android** - An open source diabetes tracker app for controlling blood glucose, HB1AC, Cholesterol, Blood Pressure, Ketones, Body Weight and more for diabetes type 1 and type 2. It can connect to Bluetooth devices for enabling the built of closed loop insulin management. [2]
- **OpenBot** - A software stack for an Android smartphones to be connected to robots, that might be self made, and serve as the robot body for the smartphone. It allows complex workloads like following a person and autonomous navigation. [3]
- **Eclipse Concierge** - Concierge is a small-footprint implementation of the OSGi Core Specification R5 standard optimized for mobile and embedded devices. [4]

[1] https://github.com/
[2] https://github.com/Glucosio/glucosio-android
[3] https://github.com/isl-org/OpenBot
[4] https://github.com/eclipse/concierge

- **GRIP Computer Vision Engine** - GRIP (the Graphically Represented Image Processing engine) is an application for rapidly prototyping and deploying computer vision algorithms, primarily for robotics applications. [5]

Table 3.1: Systems under study

Project Name	**Main programming language**	**LOC**[6]	Nr. of commits [7]	Stars	Nr. Tags	Topic
Glucosio/glucosio-android	java	59K	1'166	332	31	health automation
isl-org/OpenBot	java	191K	730	2K	7	robots
eclipse/concierge	java	62K	502	32	2	iot
WPIRoboticsProjects/GRIP	java	54K	1'170	334	26	robotic vision

[5]https://github.com/WPIRoboticsProjects/GRIP
[6]According to github data at 06.10.2021
[7]Ibid.

Chapter 4

callgraphCA - call graph evolution and code change analytics

The motivation for developing *callgraphCA* is to support the analysis of software systems evolution to different stakeholders; for example, researchers that conduct studies on multiple software systems, or development teams that require a "big picture" of the system network structure and want to study its evolution. *callgraphCA* aims to bridge the gap between the subjects of analysis of code evolution and network analytics applied to software engineering that complements other analysis tools. It provides data for a high level of abstraction, to gain perspectives of structural and conceptual changing, considering changes in the call graph. For this thesis, we focused on the evaluation of complex cyber-physical systems.

callgraphCA is a source code analysis and data management tool, that discovers the incremental changes of the underlying call graphs of the system and its relation to change coupling, and it´s solely based on data from the software's version controlling repositories.

In this section we describe the *callgraphCA*'s software architecture, the requirements it serves, and discuss the components and technologies behind it.

4.1 Requirements

callgraphCA is aimed to support the research of multiple software systems in scenarios where the user does not posses the compiled code and might not have the purpose of compiling the systems. The literature presented in Chapter 2 describes two approaches that have been used to study software systems, change coupling, and network analysis. *callgraphCA* should deliver information linking the software evolution and the relationship to the network structure based on call-graphs.

Specifically, it is required that the data can be imported from various repositories, pre-processed consistently, exported into a persistent database for further analysis. The current pilot study in-

tends to cover cases of cyber-physical systems based on Java, however the tool should be easily extensible to process other programming languages.

Functional requirements. We can sort the functional requirements in six core sections: data import, data management, data processing, data export and storage, analytics, and user parametrization.

The data import requirements consist in the ability to import the data into *callgraphCA* from diverse repositories. Because of our future work with industry partners, it should be easily extensible to manage other repository types like Microsoft Team Foundation Version Control (TFVC), CSV and SVN, which are still being used for current projects and would provide the capacity to evaluate legacy systems that might still be in production.

For the purpose of manual validation, one data management requirement is that the tool must enable to save the source code and source code differences as well as parsing results on a cache folder in the processing machine. Because it is intended to be used for the study of different projects at the time, the creation of a database for each project, as well as the storage of cache source data, must proceed in an automatic way.

The data processing requirements include stability and completeness of processing within the limits described in the chapter 7.0.1. The user should be able to run the analysis of specific data ranges, like explained in section 3.2.2 and obtain the same results if the ranges have not changed, most importantly, the tool should only update records that are relevant, for example, to change a newly deleted function or function call. Due to the link to AST parsing, a complete language agnosticism might not be granted, but the tool should provide easily extensible support for the analysis of systems in different programming languages. The processing and saving of data is able to run as background-tasks, enabling the constant access to the data for visualization.

The data export and storage requirements include the ability to change the location where the cache source and parse files are stored, and the possibility of not storing cache data. Additionally, to be able to write in diverse databases for the project under study, the tool must be easily extendable to change the type of database systems where the data results from processing are stored, like Sqlite, PostgreSQL, Azure DB, MySQL or Oracle.

In the requirements for the support to analytics, the users might have different information needs, so, the tool must provide different levels of granularity at the level of code and network analytics. The information should be subject to slicing and dicing, for example, to set different time windows and other process or time related parameters, like commits, tags and releases.

Non-functional requirements. The Non-functional requirements are regarding the extensibility and therefor modularity, for different data import and export requirements. Additionally, because the users might use diverse systems for their work, there is a portability requirement, so the tool can run in diverse platforms.

4.2 Features

callgraphCA is a portable tool that can be used to analyze diverse systems. It currently offers the following features:

User Parametrizing. In the following paragraphs, we will explain the parameters that the user can give when launching the tool, so that it can: configure the sources of information, program the language of the system, select data or tag ranges, define whether the source code and AST parsing files will be saved in cache directories and the paths for saving the outputs in a database. *callgraphCA* provides a CLI interface for launching the execution, but when downloading the source code, it can also be triggered within the development environment (IDE). In the section 7 we explain planed interface features.

Data import. Because of the parametrizability of *callgraphCA* the users can set different Git repositories as sources for their study. The modular architecture of the repository import adapters currently allows connections to Git repositories, through the given repository path parameter. The import adapters are designed for easy future adaptation of the tool to new repository type requirements. Additionally, the tool allows date and tag range restrictions, targeted commits and it filters out merge commits and branches.

Data processing. Like explained in the section of 3.2.2, the users are able to process the same system with different time frame parameters and the results remain consistent. It means that *callgraphCA* will only overwrite data that extends the previous knowledge of the system, but will not change data that was known to be closed. The data transformation component allows different programming languages to be parsed into AST and afterward, these trees are examined to extract the function and method calls for each declared function in the file, respectively, class.

Data management, export and storage. The current default data storage is a local Sqlite[1] database. This feature can be extended to include connections to PostgreSQL[2], and others, depending on the needs of our future research industrial partners. On the command line, the user can reset the database an determine if cache files should be kept or deleted. Furthermore, the paths to where the database and cached files are stored can be easily changed. In the architecture section we will explain how the export adapter can be extended to accommodate other database types.

Data Analysis and Visualizations For the ease of use, *callgraphCA* has separated analytics tables that might be seen as a data warehouse. This ensures decoupling and the ability to distribute this information to other channels that are not necessarily the ones who created the data.

[1] https://sqlite.org/
[2] https://www.postgresql.org/

Currently, the tool provides data for assessing change proneness, coupling and call-graph analysis. Additionally, the visualization library provides, to our understanding, common visualizations where related parameters can be given, e.g. start and end date for displaying the function changes.

Portability. The current state of the software can be executed in Windows, Linux (Ubuntu) and iOS platforms.

Extensibility. The processing of new programming languages should be accomplished with minimum changes in the software, for this intention, artifacts might be employed and their interface should be adaptable through a change of parameters.

4.3 Architecture and system design

In this section, we present a short description of how our tool matches the requirements above mentioned and implements a flexible framework for extending its functionalities.

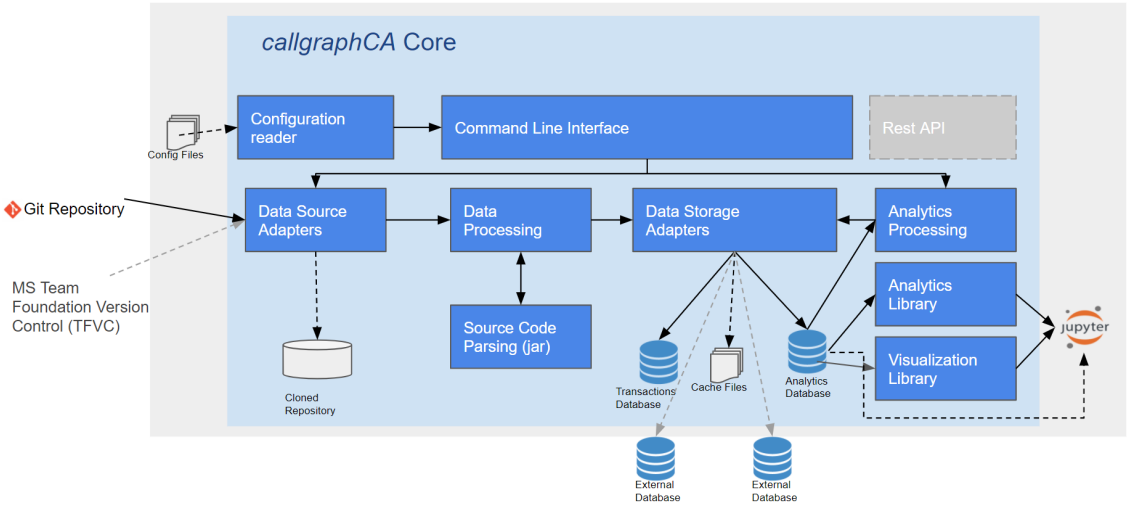


Figure 4.1: Current *callgraphCA* Architecture

4.3.1 Overview

Our architecture resembles reasonably the explained model by Alexandru *et al.* [83] of an archetypical software analysis framework. One innovation from our approach is that for our tool it is not required that source code data is saved in the local machine, this improves the performance by avoiding i/o processing. In Figure 4.1 we draw the features that are not required with intermittent

arrows, and features that are under construction are signaled with light gray intermittent arrows and lines. Through the implementation of abstract data models, *callgraphCA* provides ease by expanding functionality to adapt to different environments.

Launching. Through a command line interface (CLI) the user can start the analysis of a project. It is possible to send the parameters directly in the starting command, or when launching the application, it can read the parameters from configuration files laid for the selected projects.

Data Collection. The first mission for analyzing software is to extract and transform the raw data into a robust model. *callgraphCA* collects the repository information through a repository mining adapter. We provide a default adapter for Git repositories, but the functionality can be adapted for different repository types, in our case TFVC will be required for the analysis with an industrial partner.

Data Processing. On launching the app, you may download, analyze and capture both the relevant static software structure as well as the accompanying versioning process information. We traverse through the repository commit history and parse the change sets, as well as the related source code, both of the current as the previous state of the file. For the scope of our thesis, we only process commits that were done to the relevant file types, in this case .java files. The package hierarchy can be either built incrementally as the commits are being processed, but the tool also offers the possibility of building it from downloaded sources. Because each language has its own syntax *callgraphCA* implements an interface where different self-contained, executable Java applications can be *plugged in* to execute the parsing of the code files and return a standard syntax to the tool. The path to the .jar file for the parsing of the specific language can be set through the configuration file or the CLI. The structural information of the code is processed by the parser and injected back for further processing. This information is enriched by data generated from the mining of the repository.

4.3.2 astChangeAnalyzer

We searched for a multi-purposed tool to analyze the source code differences between two files. Because we aimed to search for function calls within the changed functions, we needed a fine grained tool, so for this task we built a wrapper tool based on *gumtreediff*[3] [106]. Our wrapper, *astChangeAnalyzer*, applies the visitor pattern when an AST node change is detected, for finding the parent function, or finding the *calling function* and *called function*. It returns an array of records containing these relationships. Additionally, the *astChangeAnalyzer* offers the option to save the parsed code in the local storage, with the parameters that activate this function and give the destination path.

[3]https://github.com/GumTreeDiff/gumtree

4.3.3 Data model

Our data model is split into transactional and analytic models. The transactional model contains fourteen tables that are divided in *commit* data, *structural* data, like the package hierarchy, and the *graph* data containing call relations between the functions and file imports. The analytics model contains tables on the level of file, package and function related to *change proneness*, *change coupling*, mainly the association rules, and the evolution of *function calls*, which offer the basis to build the network models. Our data model is directed to object oriented program languages, but we expect that it is flexible and abstract enough to seek for comparability of unpaired systems.

4.3.4 Technology

callgraphCA is written in Python 3.9. A list containing the employed libraries can be seen in the Appendix section **??**.

4.4 Outputs

An important part of our system design are the provided outputs for the users or researchers. There are two different types of outputs: the data and the services, in this case, the libraries. Figure 4.1 exhibits how these components interact with the external systems. The main output, is the analytics database that can be accessed from diverse systems. Visualization and analytics libraries provide straightforward execution commands for the support of standard analysis. Two examples of such standard functionalities are:

Threshold finding. As explained previously in the methodology section, the topic of finding relevant thresholds is a basic start up point for the analysis and might not be a trivial tasks, especially when applying the algorithms to different projects. A simple command from a library from *callgraphCA* supports the practitioner to explore thresholds by observing the possible aggregated results. This functionality will allow the researchers to decide the range of thresholds that best fits each different situation.

Change proneness visualization In the section of Section 5, in Figure 5.2 we show an extract of the visualisation of change proneness. The flexible input of data allows the users to slice the desired time frames and ranges of files, functions or calls, to be analyzed.

Chapter 5

Results and Discussion

In this section, we discuss the results obtained from the analysis of our systems under study. As explained in Section 3.3, these are Java projects in the field of cyber-physical systems. We define a *relevant commit* one that contains source code files, in our case .java files. The projects have a larger number of commits than the reported here, the divergence is given by the commits that were excluded because there were no source code file changes in them.

One of the motivations for our research, and tool development, is to bridge the gap between the detection and understanding of logical and structural coupling. D'Ambros *et al.* [40] found certain metrics that correlate with software defects in OO open source systems, we will present two of them in this chapter.

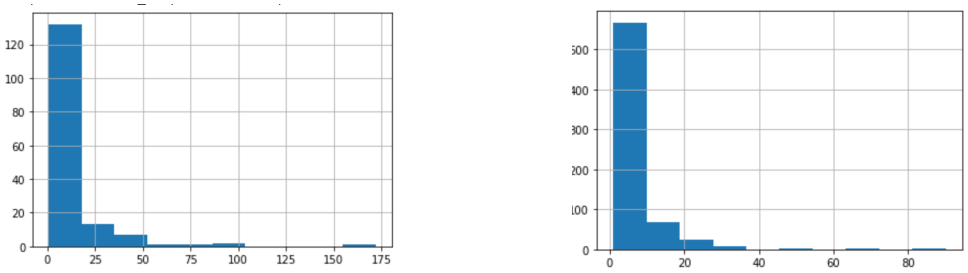
5.1 Change proneness

Before the introduction of the term *change coupling* by Fluri *et al.*, in 2003, Bieman *et al.* [107] pointed out that frequent changes in clusters of classes might reflect functional coupling or chronic problems in the architecture of the system; both Nagappan *et al.* [108] and D'Ambros *et al.* [40] found that change proneness correlates stronger that coupling in the projects they studied back then. In this sense, and following Bieman's *et al.* proposal, in *callgraphCA* we provide functionalities to identify change-prone clusters that may promote the better steering of the the development process.

The summary of results analyzing the change proneness trough out the life of the systems under study is shown in Table 5.1. For a visual aid, in Figure 5.1 we present the distribution of number of commits per file for the projects *glucosio-android* and *GRIP*. The skewed distribution comes from the many files changing rarely, and very few files who are often updated, this distribution is similarly present in all of our projects. These results align with previous research, like discussed in Section 2, that has found that many social artifacts, like social and economical networks, present long tailed behaviours with scarce matrices. The identification of co-change patterns in complex systems with such exponential behaviours is not a task for intuitive pattern detection and needs support from appropriate tools.

Table 5.1: Number of commits aggregated on file and function level.

Nr commits per file				Nr commits per function			
glucosio-android	*openBot*	*eclipse-concierge*	*GRIP*	*glucosio-android*	*openBot*	*eclipse-concierge*	*GRIP*
1779	1002	997	3620	4786	3078	4998	9461
11,33	9,54	3,88	5,4	2.62	2.53	2.48	2.23
5	5	2	3	2	2	1	2
1	1	1	1	1	1	1	1
172	76	140	90	64	29	40	47
20	12,38	11,08	7,58	3.55	2.49	2.59	1.93
4,66	2,57	9,53	4,76	8.84	04.05	4.83	6.36



(a) *glucosio-android* (b) *GRIP*

Figure 5.1: Distribution of file commits

5.2 Change coupling

When adopting association rule learning to discover co-changing patterns, it is important to find a balance between highly restrictive thresholds that eliminate *loose* couplings but reduce the number of discovered rules to small sets that might be not actionable, and on the other side, where so much association rules are found, with low support, the user finds them irrelevant. In Table 5.2 we display the amount of generated rules when applying the *Apriori* algorithm with the given support threshold. We can observe that, the lower the threshold, the more rules will be found, respectively more itemsets. Referring again to the study by Bavota *et al.* [101] on developers perception, and taking into account the results from the table, we opted for using a support threshold of 0.02.

In Table 5.3 we compare the logical and structural coupling at the level of file commit for the association rules found for the projects. The first column is the number of rules found with a threshold of 0.02. The second column displays the number of rules with itemsets that present a structural dependency. For each project there are two rows. The first row shows rules generated with the threshold and no limit of items, the second row shows the numbers having itemsets larger than two. In 2018 Ajienka *et al.* [93] found that when two objects had a dependency, 70% of the time they were also semantically linked. Despite our small sample and risking to leave the project *eclipse-concierge* aside as an outlier, we are inclined to say that our values of the association

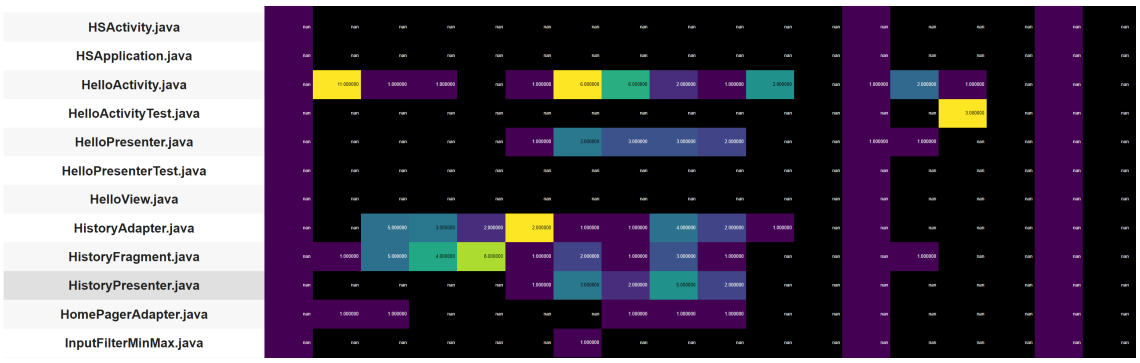


Figure 5.2: Visualization of change proneness at file level.

Table 5.2: Association rule learning on file commit level. Number of rules from thresholds.

glucosio-android	*openBot*	*eclipse-concierge*	*GRIP*
[0.5, 0]	[0.5, 0]	[0.5, 0]	[0.5, 0]
[0.451, 0]	[0.451, 0]	[0.451, 0]	[0.451, 0]
[0.304, 0]	[0.304, 0]	[0.304, 1]	[0.304, 0]
[0.255, 1]	[0.255, 0]	[0.255, 2]	[0.255, 0]
[0.156, 3]	[0.156, 1]	[0.156, 2]	[0.156, 0]
[0.059, 13]	[0.059, 15]	[0.059, 9]	[0.059, 6]
[0.01, 824]	[0.01, 340]	[0.01, 234]	[0.01, 189714]

rules that are found to have a structural coupling seem in a range of the expected. At this point, we need to mention the limitations and threats to validity of not slicing over time windows. For the current implementation we compare just the existence of a structural dependency without filtering for time periods.

With the output of the association rules, the user can easily know the number of occurrences of each of the rule's *item*'s within the set of *transactions*. For such purpose our analytics library offers support functions. Figure 5.3 *show_transactions_containing_items( )* displays the number of times that items *1,2* and *1,2,3,4* existed in the *transactions* set, furthermore, it explains the directional number of occurrences where the first item is the predecesor and so forward until the last item.

```
[ ]  1  l_elem = ['AddCholesterolActivity.java', 'AddA1CActivity.java']
     2  show_transactions_containing_items(df,'files_in_hash', l_elem, print_elems=False)

Element count. Df len 582. 1ind: 22, 2dep: 14, 2ind: 16

[ ]  1  l_elems = ["'AddKetoneActivity.java'", "'AddWeightActivity.java'", "'AddGlucoseActivity.java'", "'AddPressureActivity.java'"]
     2  show_transactions_containing_items(df, l_elems, print_elems=False)

Element count. Df len 582. 1ind: 25, 2dep: 20, 3dep: 17, 4dep: 17,
    2ind: 26, 3ind: 47, 4ind 21
```

Figure 5.3: Transaction support values

Table 5.3: Association rules and their positive structure rate.

Nr. Rules	Nr. Rules with positive structural dependency	P
glucosio-android		
140	83	0.59
39	38	0.97
openBot		
55	7	0.13
20	7	0.35
eclipse-concierge		
57	35	0.61
36	35	0.97
GRIP		
185	90	0.49
105	90	0.86

5.3 Call graph evolution

In the methodology section Section 3 we presented the abstract concept of call graph evolution and explained the functional requirements for the processing algorithm. Figure 5.4 shows an example from the project *glucosio-android* that would result in an addition of a new *function_call* record being *AboutActivity::onPreferenceClick(param)* the calling function, and *External LinkActivity.launch(arg, arg, arg)* the called function. Accompanying, the hash value of this commit will be set as the *end_hash* to the *function_call* that was previously existing in the method *AboutActivity::onPreferenceClick(param)*.

When running the whole history of commits we register the *function_call* changes (additions or deletions) that are reported in the Table 5.4. With this data we can build an answer the to the *RQ1*.

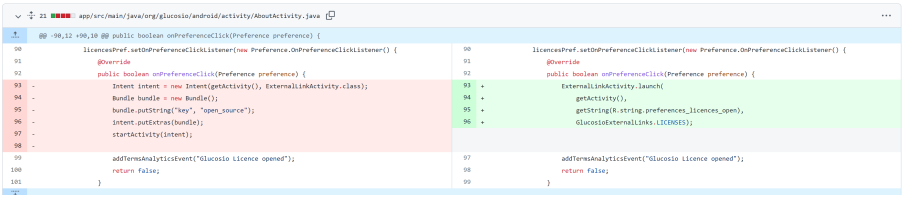


Figure 5.4: Addition and deletion of function calls.

5.3.1 Building the call-graph

"The empirical results show that:(1) identifier based methods have more computational efficiency but cannot always be used interchangeably with corpora-based methods of computing semantic coupling of classes" (Ajienka *et al.* [109])

Table 5.4: Number of changes aggregated on call graph level.

	glucosio-android	*openBot*	*eclipse-concierge*	*GRIP*
sum	113316	19390	164326	39535
mean	94.04	32.75	130.73	23.99
median	18	12.5	12	9
min	1	1	1	1
max	6129	854	5867	319
std	308.96	65.45	421.32	40.05
skewness	11.36	6.89	07.01	3.82
count	1205	592	1257	1648

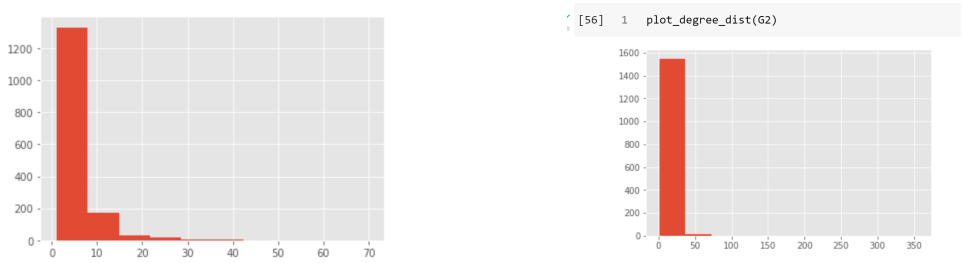
For the scope of the thesis, we initially used the gumtreediff tool to implement the visitor patternfor the nodes that reported source changes, unfortunately they are already isolated AST's the root method could not be always found, additionally, getting the inner calls proved to be challenging. Our newest attempt was a methodology mixing structured information retrieval by parsing the source files and transforming them in the xml and enriching this information with the import dependencies of the file where the calls reside.

We needed an efficient tool to benchmark the graphs we generated. For this purposes we found *sourceTrail*[1], it is a portable tool that generates references from the source and supports the navigating between files. A convenient functionality is that the underlying information is saved in a Sqlite database.

The concept for building the graph is first to build the relationships between files, packages, classes and its methods. A limitation is the detection of nested classes, for example. Then we visit the function and search for the nodes that are calls and retrieve the name of the functions including its parameters. When building the call relations we search in the class imports and join the functions that this imports have to make a sub set of the candidate functions in the imported file. Functions like 'toString', 'instanceOf', 'init' are pruned from the set.

The initial attempts without pruning resulted in magnitudes of distance in the network metrics compared to the benchmark database. The base measure we use is the node and edge count differences and distance between the degree distributions. We applied the Jenson-Shannon Distance and the Kolmogorov–Smirnov test to reject the hypotesis that the degree distributions of the graphs are the same. We can appreciate that they both present a long tailed behaviour, as in Figure 5.5 but the statistical similarity was never achieved. The nodes in the SourceTrail database are saved with a serialized name, a reliable parsing of their qualified names was challenging and to the date not completed. When comparing the nodes one to one we notice that our algorithm did not found many function calls, mainly due to the complexity of the implementation, on the other hand, we obtained to many nonexistent calls because our similarity pattern based on a class and package level.

[1]https://www.sourcetrail.com/



(a) *glucosio-android* (b) *GRIP*

Figure 5.5: Graph degree distributions, benchmark vs *callgraphCA* generated.

Chapter 6

Threats to Validity and Limitations

6.1 Construct and Internal validity

Threats to *construct validity* are about the relationship between theory and observation. In this study, threats can be mainly due to the imprecision of dependencies extraction from software artifacts. We mitigated both threats by leveraging standard and well-known tools for code change analysis and graph dependencies extraction, as detailed in Section 3 and Section 4.

Threats to *internal validity* are about the cause-effect relationships between the investigated dependencies and their root causes (e.g., fine-grained changes). We looked at the fine-grained changes as well as changes in the call-graph evolution, to ensure that our analysis and root causes are valid for the investigated projects.

The study is focused on implementing a tool that performs a top-down analysis: the tool analyzes different levels of granularity of changes of CPSs, from higher level (e.g., commit) to lower levels (e.g., function or method calls). Another strategy that could be investigated in future work, which is typical in change analysis tools, is a bottom up approach, to compare the stability of the results. Moreover, our tool is based on AST parsing and retrieval of static information from the source code, which can be less accurate than dynamic information or does not have the advantages provided by some compilers. We use AST parsing to retrieve the function calls and then, we match them based on the imports from the given file. For this study, we limit ourselves to direct imports, and not relationships that might be attained due to inheritance or interface implementations, this impacts the rate for finding the matching graph relations. For bench-marking our generated graphs we use the tool SourceTrail. SourceTrail can analyze the code with or without compilation libraries. We comparison building the code graphs on compiled code that there is a large improvement in the information retrieval. However, this information retrieval was a specific design decision, explained in our motivation, to support multiple comparison of systems written in different languages, where the stakeholders do not posses compilation skills or resources. To ensure a reliable evaluation when assessing change coupling, we used the Apriori algorithm.

However, it is possible that the other algorithms briefly introduced in Section 3 might provide other results. Within the future steps of the conducted research, we plan to perform further procedures for addressing the internal validity threats, which deal sampling biases, methodology for measuring and comparing results and variation of algorithms and thresholds.

6.2 External validity

Finally, threats to *external validity* concern the generalization of our findings. Although the number of analyzed projects belonging to different domains is relatively large, by no means they can be generalized to the universe of open-source CPSs. Therefore, further replications are desirable, in both open-source but also in industrial contexts, to assess the generalizability of the proposed approach.

In this thesis we propose a prototype tool, *callgraphCA* to make a empirical analysis of open source projects, but our selected projects were in the scope of cyber-physical systems, and were constraint in size and language, these are reasons that largely affect the generalizability of the results. It might also be the case, that the selected systems might not be a representative sample for CPS systems. Specifically, the study conducted in this thesis is limited to open source projects from GitHub, in the topic of Cyber-Physical Systems, with all projects primarily written in Java. To limit any bias on the results discussed in this thesis, we applied a strict selection criteria for selecting the project to analyze, as detailed in Section 3. We expect to extend our research to other programming languages and other kind of systems.

It is unclear if our approach would be equally suited for industrial software or other types of systems, having languages and organization structures as well as different evolutionary patterns. However, the concepts of analyzing change coupling and the structure of the code should be broadly applicable. Because we retrieve information by parsing the code and forming ASTś, one important root-cause and potential threat to the precision and coverage of our tool is the constant evolution of the programming languages. We can observe in the history of java versions [1], that there are constantly new features and syntax being added, as well as others become deprecated. This impacts the stability of the systems under study and comparability of the projects.

[1] https://www.java.com/releases/

Chapter 7

Limitations and Future Work

We have built a light-weighted tool that aims to support the understanding of software evolution, especially focused in tackling the difficulties of identifying, simplifying and understanding the change-coupling patterns or rules underlying in complex systems. *callgraphCA* is currently able to analyze Java projects from git repositories and it provides partial functionalities for C++ and C systems. In the course of this masters thesis we discover several approaches that have faced the same challenges as us, like for example the limitations of depending from language specific parsers. In our case, our AST visiting approach is dependent in heuristic rules to explain which parts of the code are relevant for translating into the abstract data model.

Many tools collect the software symbols and their relations, function calls, by accessing compilation databases, in the case of Java the pre-compiled header and flag files, and for C++ the compilation database.

7.0.1 Limitations

Soundness . Our approach was to provide a "big picture" perspective that would allow rough estimations when evaluating the risk of changes, it was not our aim to be absolutely precise in all of the cases, but to develop a useful to the end user and responsive tool that was not aiming at program understanding at a source code level but in focusing on the dynamics of the function calls evolution that might present patterns that are coupled to changes in the functionality of the software or reflect architecture or implementation instabilities.

Additionally to internal challenges while developing algorithms to get more precision, there are also external factors that influence the precision of the results. For example, we found out that *callgraphCA* was constantly adding and removing functions and function calls despite the real code not changing, in Figure 7.1 we can see an example of this situation. The precision of the language parsing tools for popular object oriented languages is adequate for our purposes but there is a limitation when the languages evolve, that the parser don't catch the new syntactical structures. This risk exists as well when analyzing legacy systems.

Git sensitivity .

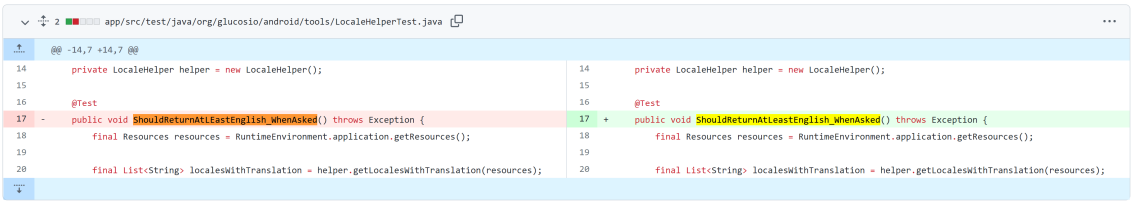


Figure 7.1: Git reports change in unchanged function declaration

7.1 Future work

During the course of this thesis, specially when researching previous work, I acquired new perspectives regarding the research of software engineering, how tools in the past have tackled certain needs and how the systems have evolved in the last years, as well as noticing that certain topics are still providing challenges for this area.

Our main concern in the following months will be to improve the functionality of *callgraphCA* so that it builds reliable call-graphs from the repository mining information, for this goal we will extend our work in the parsing of either ASTs or improve the heuristic rules for node extraction of xml representations.

We are aware that the scope of the thesis and the intensive effort in development did not allowed us to reach best practices in research methodology. After *callgraphCA* is stable we will extend our bench-marking to C++ systems like the ones listed in the Table Table 7.1. Additionally we propose to generate synthetic benchmarks of automatically generated projects diverse number of methods. We believe that this can be a great contribution for the community.

The previous endeavours have a *vertical* component, that is, we compare diverse systems but under the scope of change proneness and call-graph analyitics, We want to extend the comparison by measuring up our approach to other existing metrics, like process and semantic ones.

We plan to use *callgraphCA* in a project with an industrial research partner that is in the field of medical devices. For this we will need to expand the parsing capabilities to C and generate other source access adapters for Microsoft Team Foundation Versions Controller (TFVC) and containerize our system.

Last, we would like to replicate the study of Bavota *et al.* [101] that researches the developers ability to find the different types of change. We would add a group that will be assisted by our tool.

- **OpenHAB Core** might delete too large- A framework to build smart home products solutions via OSGi bundles. [1]
- **JKQtPlotter** - An extensive Qt5 Plotter framework without external dependencies. [2]
- **Blynk library** - Blynk library for embedded hardware. Blynk offers a fully integrated suite of IoT software. [3]

[1] https://github.com/openhab/openhab-core
[2] https://github.com/jkriege2/JKQtPlotter
[3] https://github.com/blynkkk/blynk-library

- **RF24** - An optimized high speed driver for wireless transceiver provides the OSI Layer 2 driver for nRF24L01 on Arduino Raspberry Pi/Linux Devices. [4]
- **Free Gait** - A software framework for the control of legged robots. Defines a whole-body abstraction layer and tracks the motion tasks with a feedback controller to ensure accuracy. [5]
- **Syropod High-level Controller** - A controller for quasi-static multilegged robots. This Robot Operating System (ROS) can be deployed on robots with different sensor, leg and joint configurations; it can generates trajectories, step clearance, etc. It handles input sensors feedback that can be used to control the robot in, for example, inclined or uneven terrains. [6]
- **tzapu WiFiManager** - A WiFi Connection manager for esp8266 devises with web management portal. [7]
- **CHAMP** might delete no Tags :(- An open source development framework for building new quadrupedal robots and developing new control algorithms. [8]

Table 7.1: Future systems under study

Project Name	**Main programming language**	**LOC**[9]	Nr. of commits [10]	Stars	Nr. Tags	Topic
openhab/openhab-core might remove	java	284K	1'539	590	54	home-automation
jkriege2/JKQtPlotter	c++	1.2M	476	311	6	iot
blynkkk/blynk-library	c++,c	36K	1'831	3.3K	35	iot
nRF24/RF24	c++	31K	870	1.8K	26	iot
leggedrobotics/free_gait	c++	24.5K	907	292	7	legged robots
csiro-robotics/syropod_highlevel_controller	c++	11.6K	638	86	18	legged robots
tzapu/WiFiManager	c++	8.1K	1'095	4.9K	19	iot
chvmp/champ might remove	c++	13.5K	708	800	0	legged robots

TODO

[4] https://github.com/nRF24/RF24
[5] https://github.com/leggedrobotics/free_gait
[6] https://github.com/csiro-robotics/syropod_highlevel_controller
[7] https://github.com/tzapu/WiFiManager
[8] https://github.com/chvmp/champ

Chapter 8

Conclusion

We have built a light-weighted tool that support the understanding of software evolution, especially focused on tackling the difficulties of identifying, simplifying, and understanding the change-coupling patterns or rules underlying complex systems. *callgraphCA* is currently able to analyze the complete revision history from Java projects in Git repositories, and it provides partial functionalities for C++ and C systems. The current functionality provides support for process metrics that already catch much of the change coupling patterns of the projects we studied. In the course of this master thesis, we discovered several previous research projects that have faced similar challenges. For example, the dependence from language-specific parsers. In our case, our parsing approach depends on heuristic rules to obtain the relevant parts of the code to build our abstract data model. We constructed a model to follow the evolution of the call graph, unfortunately, the reconstruction of structure dependencies on the level of package and file import and inheritance, blurred our efforts to build a reliable graph, additionally, the unqualified name matching turned has strong risks even if the package dependency is solved, because many classes within the package override the same method. We conclude that to build a reliable call graph it is necessary to increase the language syntax dependency to build a more solid source to model matches. Despite the current state of the tool regarding the call graph evolution functionality, we keep confident that in the next months a stable and sound tool can be developed for the intended purposes.

Bibliography

[1] C. Chen, S. Lin, M. Shoga, Q. Wang, and B. Boehm, "How do defects hurt qualities? an empirical study on characterizing a software maintainability ontology in open source software," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 226–237, 2018.

[2] O. Nierstrasz, M. Kobel, T. Gîrba, M. Lanza, and H. Bunke, "Example-driven reconstruction of software models," pp. 275–286, 01 2007.

[3] H. Chen, "Applications of cyber-physical system: A literature review," *Journal of Industrial Integration and Management*, vol. 02, no. 03, p. 1750012, 2017.

[4] J. Wilhelm, C. Petzoldt, T. Beinke, and M. Freitag, "Review of digital twin-based interaction in smart manufacturing: Enabling cyber-physical systems for human-machine interaction," *Int. J. Comput. Integr. Manuf.*, vol. 34, no. 10, pp. 1031–1048, 2021.

[5] S. Panichella, A. D. Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall, "How can i improve my app? classifying user reviews for software maintenance and evolution," in *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015* (R. Koschke, J. Krinke, and M. P. Robillard, eds.), pp. 281–290, IEEE Computer Society, 2015.

[6] A. D. Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. C. Gall, "What would users change in my app? summarizing app reviews for recommending software changes," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pp. 499–510, 2016.

[7] D. Zhang, G. Feng, Y. Shi, and D. Srinivasan, "Physical safety and cyber security analysis of multi-agent systems: A survey of recent advances," *IEEE/CAA Journal of Automatica Sinica*, vol. 8, no. 2, pp. 319–333, 2021.

[8] Y. Zhou, F. R. Yu, J. Chen, and Y. Kuo, "Cyber-physical-social systems: A state-of-the-art survey, challenges and opportunities," *IEEE Communications Surveys Tutorials*, vol. 22, no. 1, pp. 389–425, 2020.

[9] J. Park, R. Ivanov, J. Weimer, M. Pajic, S. H. Son, and I. Lee, "Security of cyber-physical systems in the presence of transient sensor faults," *ACM Trans. Cyber Phys. Syst.*, vol. 1, no. 3, pp. 15:1–15:23.

[10] H. Zhou, W. Li, Z. Kong, J. Guo, Y. Zhang, B. Yu, L. Zhang, and C. Liu, "Deepbillboard: systematic physical-world testing of autonomous driving systems," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pp. 347–358, ACM, 2020.

[11] M. Törngren and U. Sellgren, *Complexity Challenges in Development of Cyber-Physical Systems*, pp. 478–503. Cham: Springer International Publishing, 2018.

[12] J. Kim, S. Chon, and J. Park, "Suggestion of testing method for industrial level cyber-physical system in complex environment," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, IEEE, Apr. 2019.

[13] T. Guardian, "Self-driving uber kills arizona woman in first fatal crash involving pedestrian," 2018.

[14] The-Washington-Post, "Uber's radar detected elaine herzberg nearly 6 seconds before she was fatally struck, but "the system design did not include a consideration for jaywalking pedestrians" so it didn't react as if she were a person.," 2019.

[15] F. Ingrand, "Recent trends in formal validation and verification of autonomous robots software," in *3rd IEEE International Conference on Robotic Computing, IRC 2019, Naples, Italy, February 25-27, 2019*, pp. 321–328, 2019.

[16] B. Boehm and N. Kukreja, "An initial ontology for system qualities," *INCOSE International Symposium*, vol. 25, no. 1, pp. 341–356, 2015.

[17] B. Boehm, C. Chen, K. Srisopha, and L. Shi, "The key roles of maintainability in an ontology for system qualities," *INCOSE International Symposium*, vol. 26, no. 1, pp. 2026–2040, 2016.

[18] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, "The impact of test case summaries on bug fixing performance: an empirical investigation," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016* (L. K. Dillon, W. Visser, and L. A. Williams, eds.), pp. 547–558, ACM, 2016.

[19] P. Rani, S. Panichella, M. Leuenberger, A. D. Sorbo, and O. Nierstrasz, "How to identify class comment types? A multi-language approach for class comment classification," *J. Syst. Softw.*, vol. 181, p. 111047, 2021.

[20] Y. Tymchuk, A. Mocci, and M. Lanza, "Code review: Veni, vidi, vici," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 151–160, 2015.

[21] S. Panichella, "Supporting newcomers in software development projects," in *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany,*

September 29 - October 1, 2015 (R. Koschke, J. Krinke, and M. P. Robillard, eds.), pp. 586–589, IEEE Computer Society, 2015.

[22] G. Grano, C. Laaber, A. Panichella, and S. Panichella, "Testing with fewer resources: An adaptive approach to performance-aware test case generation," *IEEE Trans. Software Eng.*, vol. 47, no. 11, pp. 2332–2347, 2021.

[23] T. Zimmermann, N. Nagappan, and A. Zeller, *Software Evolution*, ch. Predicting Bugs from History. Springer Berlin Heidelberg, 2008.

[24] R. Subramanyam and M. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects," *IEEE Transactions on Software Engineering*, vol. 29, no. 4, pp. 297–310, 2003.

[25] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pp. 31–41, 2010.

[26] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, pp. 9–9, 2007.

[27] T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting fault incidence using software change history," *Software Engineering, IEEE Transactions on*, vol. 26, pp. 653 – 661, 08 2000.

[28] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan, "Understanding the impact of code and process metrics on post-release defects: A case study on the eclipse project," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, (New York, NY, USA), Association for Computing Machinery, 2010.

[29] M. A. Jenkins, "Translating apl, an empirical study," in *Proceedings of Seventh International Conference on APL*, APL '75, (New York, NY, USA), p. 192–200, Association for Computing Machinery, 1975.

[30] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call graph construction in object-oriented languages," in *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, (New York, NY, USA), p. 108–124, Association for Computing Machinery, 1997.

[31] D. Grove and C. Chambers, "A framework for call graph construction algorithms," *ACM Trans. Program. Lang. Syst.*, vol. 23, p. 685–746, Nov. 2001.

[32] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks," *Rev. Mod. Phys.*, vol. 74, pp. 47–97, Jan 2002.

[33] A. Potanin, J. Noble, M. Frean, and R. Biddle, "Scale-free geometry in object-oriented programs," *Communications of The ACM - CACM*, vol. 48, 01 2004.

[34] S. Valverde and R. Sole, "Hierarchical small-worlds in software architecture," *Dynamics of Continuous Discrete and Impulsive Systems: Series B; Applications and Algorithms*, vol. 14, p. 1, 01 2007.

[35] Y. Qu, X. Guan, Q. Zheng, T. Liu, L. Wang, Y. Hou, and Z. Yang, "Exploring community structure of software call graph and its applications in class cohesion measurement," *Journal of Systems and Software*, vol. 108, pp. 193–210, 2015.

[36] J. Kinable and O. Kostakis, "Malware classification based on call graph clustering," *Journal in Computer Virology*, vol. 7, pp. 233–245, Feb. 2011.

[37] P. Wu, J. Wang, and B. Tian, "Software homology detection with software motifs based on function-call graph," *IEEE Access*, vol. 6, pp. 19007–19017, 2018.

[38] S. Romano, G. Scanniello, C. Sartiani, and M. Risi, "A graph-based approach to detect unreachable methods in java software," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, (New York, NY, USA), p. 1538–1541, Association for Computing Machinery, 2016.

[39] S. Panichella, "Summarization techniques for code, change, testing, and user feedback (invited paper)," in *2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests, VST@SANER 2018, Campobasso, Italy, March 20, 2018* (C. Artho and R. Ramler, eds.), pp. 1–5, IEEE, 2018.

[40] M. D'Ambros, M. Lanza, and R. Robbes, "On the relationship between change coupling and software defects," in *2009 16th Working Conference on Reverse Engineering*, pp. 135–144, 2009.

[41] S. Panichella, M. I. Rahman, and D. Taibi, "Structural coupling for microservices," in *Proceedings of the 11th International Conference on Cloud Computing and Services Science, CLOSER 2021, Online Streaming, April 28-30, 2021* (M. Helfert, D. Ferguson, and C. Pahl, eds.), pp. 280–287, SCITEPRESS, 2021.

[42] R. Lutz, "Analyzing software requirements errors in safety-critical, embedded systems," in *[1993] Proceedings of the IEEE International Symposium on Requirements Engineering*, pp. 126–133, 1993.

[43] C. Kemerer and S. Slaughter, "An empirical approach to studying software evolution," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 493–509, 1999.

[44] M. M. Lehman, J. F. Ramil, P. Wernick, D. E. Perry, and W. M. Turski, "Metrics and laws of software evolution - the nineties view," in *4th IEEE International Software Metrics Symposium (METRICS 1997), November 5-7, 1997, Albuquerque, NM, USA*, p. 20, IEEE Computer Society, 1997.

[45] A. E. Hassan and R. C. Holt, "The top ten list: dynamic fault prediction," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 263–272, 2005.

[46] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pp. 1–10, 2010.

[47] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 531–577, 2012.

[48] T. Menzies, Z. Milton, B. Turhan, B. Cukic, and Y. J. A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Automated Software Engineering*, vol. 17, pp. 375–407, 2010.

[49] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, p. 7, ACM, 2009.

[50] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall, "How can i improve my app? classifying user reviews for software maintenance and evolution," in *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ICSME '15, (Washington, DC, USA), pp. 281–290, IEEE Computer Society, 2015.

[51] M. Aniche, "Java code metrics calculator (ck)." , Apr. 2020.

[52] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. Softw. Eng.*, vol. 31, p. 897–910, Oct. 2005.

[53] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th international conference on Software Engineering*, pp. 181–190, ACM, 2008.

[54] A. E. Hassan, "Predicting faults using the complexity of code changes," in *2009 IEEE 31st International Conference on Software Engineering*, pp. 78–88, 2009.

[55] T. Zimmermann, N. Nagappan, and A. Zeller, "Predicting bugs from history," *Software Evolution*, pp. 69–88.

[56] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan, "Detection of software modules with high debug code churn in a very large legacy system," in *Proceedings of the The Seventh International Symposium on Software Reliability Engineering*, ISSRE '96, (USA), p. 364, IEEE Computer Society, 1996.

[57] T. Zimmermann, N. Nagappan, H. C. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009* (H. van Vliet and V. Issarny, eds.), pp. 91–100, ACM, 2009.

[58] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pp. 284–292, 2005.

[59] M. Pinzger, E. Giger, and H. C. Gall, "Comparing fine-grained source code changes and code churn for bug prediction - A retrospective," *ACM SIGSOFT Softw. Eng. Notes*, vol. 46, no. 3, pp. 21–23, 2021.

[60] G. Canfora, A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella, "Defect prediction as a multiobjective optimization problem," *Softw. Test., Verif. Reliab.*, vol. 25, no. 4, pp. 426–459, 2015.

[61] A. Panichella, C. Alexandru, S. Panichella, A. Bacchelli, and H. Gall, "A search-based training algorithm for cost-aware defect prediction," 07 2016.

[62] A. Tahir, K. E. Bennin, X. Xiao, and S. G. MacDonell, "Does class size matter? an in-depth assessment of the effect of class size in software defect prediction," *Empir. Softw. Eng.*, vol. 26, no. 5, p. 106, 2021.

[63] S. Wang, J. Wang, J. Nam, and N. Nagappan, "Continuous software bug prediction," in *ESEM '21: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 11-15, 2021* (F. Lanubile, M. Kalinowski, and M. T. Baldassarre, eds.), pp. 14:1–14:12, ACM, 2021.

[64] T. Aladics, J. Jász, and R. Ferenc, "Bug prediction using source code embedding based on doc2vec," in *Computational Science and Its Applications - ICCSA 2021 - 21st International Conference, Cagliari, Italy, September 13-16, 2021, Proceedings, Part VII* (O. Gervasi, B. Murgante, S. Misra, C. Garau, I. Blecic, D. Taniar, B. O. Apduhan, A. M. A. C. Rocha, E. Tarantino, and C. M. Torre, eds.), vol. 12955 of *Lecture Notes in Computer Science*, pp. 382–397, Springer, 2021.

[65] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, (New York, NY, USA), p. 452–461, Association for Computing Machinery, 2006.

[66] B. Fluri, M. Wursch, M. PInzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[67] H. C. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *IEEE Software*, vol. 26, no. 1, pp. 26–33, 2009.

[68] B. Fluri, E. Giger, and H. C. Gall, "Discovering patterns of change types," in *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pp. 463–466, IEEE Computer Society, 2008.

[69] B. Fluri, H. Gall, and M. Pinzger, "Fine-grained analysis of change couplings," in *Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pp. 66–74, 2005.

[70] S. Panichella and N. Zaugg, "An empirical investigation of relevant changes and automation needs in modern code review," *Empir. Softw. Eng.*, vol. 25, no. 6, pp. 4833–4872, 2020.

[71] B. Fluri and H. Gall, "Classifying change types for qualifying change couplings," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pp. 35–45, 2006.

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

[73] Y. Zhou, M. Würsch, E. Giger, H. C. Gall, and J. Lu, "A bayesian network based approach for change coupling prediction," in *WCRE 2008, Proceedings of the 15th Working Conference on Reverse Engineering, Antwerp, Belgium, October 15-18, 2008* (A. E. Hassan, A. Zaidman, and M. D. Penta, eds.), pp. 27–36, IEEE Computer Society, 2008.

[74] G. A. Oliva and M. A. Gerosa, "Chapter 11 - change coupling between software artifacts: Learning from past changes," in *The Art and Science of Analyzing Software Data* (C. Bird, T. Menzies, and T. Zimmermann, eds.), pp. 285–323, Boston: Morgan Kaufmann, 2015.

[75] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *22nd IEEE Int. Conf. on Software Maintenance (ICSM 2006), 24-27 September 2006, Philadelphia, Pennsylvania, USA*, pp. 469–478, 2006.

[76] B. Ujhazi, R. Ferenc, D. Poshyvanyk, and T. Gyimóthy, "New conceptual coupling and cohesion metrics for object-oriented systems," in *Tenth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2010, Timisoara, Romania, 12-13 September 2010*, pp. 33–42, IEEE Computer Society, 2010.

[77] M. Revelle, M. Gethers, and D. Poshyvanyk, "Using structural and textual information to capture feature coupling in object-oriented software," *Empir. Softw. Eng.*, vol. 16, no. 6, pp. 773–811, 2011.

[78] H. H. Kagdi, M. Gethers, and D. Poshyvanyk, "Integrating conceptual and logical couplings for change impact analysis in software," *Empir. Softw. Eng.*, vol. 18, no. 5, pp. 933–969, 2013.

[79] S. Tiwari and S. S. Rathore, "Coupling and cohesion metrics for object-oriented software: A systematic mapping study," in *Proceedings of the 11th Innovations in Software Engineering Conference, ISEC 2018, Hyderabad, India, February 09 - 11, 2018* (Y. R. Reddy, V. Varma, J. Cleland-Huang, U. Bellur, S. Sengupta, N. Sharma, R. Loganathan, R. Sharma, and S. Sarkar, eds.), pp. 8:1–8:11, ACM, 2018.

[80] L. B. L. de Souza and M. de Almeida Maia, "Do software categories impact coupling metrics?," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013* (T. Zimmermann, M. D. Penta, and S. Kim, eds.), pp. 217–220, IEEE Computer Society, 2013.

[81] N. R. Council, *Network Science*. Washington, DC: The National Academies Press, 2005.

[82] B. G. Ryder, "Constructing the call graph of a program," *IEEE Trans. Software Eng.*, vol. 5, no. 3, pp. 216–226, 1979.

[83] C. V. Alexandru, S. Panichella, S. Proksch, and H. C. Gall, "Redundancy-free analysis of multi-revision software artifacts," *Empirical Software Engineering*, vol. 24, pp. 332–380, February 2019.

[84] A. Potanin, J. Noble, M. R. Frean, and R. Biddle, "Scale-free geometry in OO programs," *Commun. ACM*, vol. 48, no. 5, pp. 99–103, 2005.

[85] G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella, "How the apache community upgrades dependencies: an evolutionary study," *Empir. Softw. Eng.*, vol. 20, no. 5, pp. 1275–1317, 2015.

[86] G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: The case of apache," in *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, pp. 280–289, IEEE Computer Society, 2013.

[87] S. D. Nikolopoulos and I. Polenakis, "A graph-based model for malicious code detection exploiting dependencies of system-call groups," in *Proceedings of the 16th International Conference on Computer Systems and Technologies, CompSysTech, Dublin, Ireland, June 25 - 26, 2015* (B. Rachev and A. Smrikarov, eds.), pp. 228–235, ACM, 2015.

[88] J. P. S. Alcocer, H. C. Jaimes, D. Costa, A. Bergel, and F. Beck, "Enhancing commit graphs with visual runtime clues," in *2019 Working Conference on Software Visualization, VISSOFT 2019, Cleveland, OH, USA, September 30 - October 1, 2019*, pp. 28–32, IEEE, 2019.

[89] A. Bergel, S. Maass, S. Ducasse, and T. Gîrba, "A domain-specific language for visualizing software dependencies as a graph," in *Second IEEE Working Conference on Software Visualization, VISSOFT 2014, Victoria, BC, Canada, September 29-30, 2014* (H. A. Sahraoui, A. Zaidman, and B. Sharif, eds.), pp. 45–49, IEEE Computer Society, 2014.

[90] H. Liu, Y. Tao, W. Huang, and H. Lin, "Visual exploration of dependency graph in source code via embedding-based similarity," *J. Vis.*, vol. 24, no. 3, pp. 565–581, 2021.

[91] O. Goonetilleke, D. Meibusch, and B. Barham, "Graph data management of evolving dependency graphs for multi-versioned codebases," in *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pp. 574–583, IEEE Computer Society, 2017.

[92] M. Savic, M. Ivanovic, and M. Radovanovic, "Analysis of high structural class coupling in object-oriented software systems," *Computing*, vol. 99, no. 11, pp. 1055–1079, 2017.

[93] N. Ajienka, A. Capiluppi, and S. Counsell, "An empirical study on the interplay between semantic coupling and co-change of software classes," *Empir. Softw. Eng.*, vol. 23, pp. 1791–1825, June 2018.

[94] G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella, "Traceability recovery using numerical analysis," in *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*, pp. 195–204, 2009.

[95] G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella, "On the role of the nouns in ir-based traceability recovery," in *The 17th IEEE Int. Conf. on Program Comprehension, ICPC 2009, Vancouver, British Columbia, Canada, May 17-19, 2009*, pp. 148–157, 2009.

[96] A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using IR methods for labeling source code artifacts: Is it worthwhile?," in *IEEE 20th Int. Conf. on Program Comprehension, ICPC 2012, Passau, Germany, June 11-13, 2012*, pp. 193–202, 2012.

[97] T. Zimmermann, S. Diehl, and A. Zeller, "How history justifies system architecture (or not)," in *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings.*, pp. 73–83, 2003.

[98] H. Gall, M. Jazayeri, and J. Krajewski, "Cvs release history data for detecting logical couplings," in *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings.*, pp. 13–23, 2003.

[99] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, (New York, NY, USA), p. 207–216, Association for Computing Machinery, 1993.

[100] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, (San Francisco, CA, USA), p. 487–499, Morgan Kaufmann Publishers Inc., 1994.

[101] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "An empirical study on the developers' perception of software coupling," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 692–701, 2013.

[102] S. Ducasse, M. Lanza, and E. Tichelaar, "Moose: an extensible language-independent environment for reengineering object-oriented systems," 04 2000.

[103] M. Patton, *Qualitative Evaluation and Research Methods*. Newbury Park: Sage, 2002.

[104] H. Borges and M. T. Valente, "What's in a github star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018.

[105] P. Soulier, Depeng Li, and J. R. Williams, "A survey of language-based approaches to cyber-physical and embedded system development," *Tsinghua Science and Technology*, vol. 20, no. 2, pp. 130–141, 2015.

[106] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pp. 313–324, 2014.

[107] J. Bieman, A. Andrews, and H. Yang, "Understanding change-proneness in oo software through visualization," in *11th IEEE International Workshop on Program Comprehension, 2003.*, pp. 44–53, 2003.

[108] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 580–586, 2005.

[109] N. Ajienka, A. Capiluppi, and S. Counsell, "An empirical study on the interplay between semantic coupling and co-change of software classes," *Empir. Softw. Eng.*, vol. 23, pp. 1791–1825, June 2018.