SOFTWARE ARCHITECTURE MINING FROM SOURCE CODE

Krystallia Savvaki

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes, Heraklion, GR-70013, Greece

Thesis Advisor: Prof. *Anthony Savidis*

UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

SOFTWARE ARCHITECTURE MINING FROM SOURCE CODE

Thesis submitted by
Krystallia Savvaki
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____

Krystallia Savvaki

Committee approvals: _____

Anthony Savidis
Professor, Thesis Supervisor

Antonios Savvidis
Digitally signed by Antonios Savvidis
Date: 2021.04.26 12:25:44 +03'00'

Panos Trahanias
Professor, Committee Member

PANAGIOTIS TRACHANIAS
Digitally signed by PANAGIOTIS TRACHANIAS
Date: 2021.04.27 10:57:12 +03'00'

Polyvios Pratikakis
Assistant Professor, Committee Member

Polyvios Pratikakis
Digitally signed by Polyvios Pratikakis
Date: 2021.05.06 11:34:39 +03'00'

Department approval: _____

Polyvios Pratikakis
Assistant Professor, Director of Graduate Studies

Polyvios Pratikakis
Digitally signed by Polyvios Pratikakis
Date: 2021.05.06 11:35:03 +03'00'

Heraklion, April 2021

Abstract

Software architecture plays a primary role in system specifications and design, acting as a bridge between requirements and implementation. As software systems evolve over time, conformance to the initial architecture may be broken, while unexpected or undesirable component dependencies may arise due to the way the source code evolves. Additionally, understanding the underlying architecture of large applications is required for effective maintenance and continuous improvement. However, the problem is that the architecture is not somehow reflected in the source code since it is not a programming language construct. As a consequence, methods and tools to extract architecture-related information from source code that can aid software developers in understanding system structure are required.

Reverse engineering is the process of analyzing a system in order to identify and represent the relationships between its components. Architecture mining is a subset of reverse engineering in which meaningful high-level abstractions that represent system components are detected. In this context, reverse engineering methods may be used to compute concrete architectures and compare it to the original conceptual architecture.

In this thesis, we propose an architecture mining tool with the aim of reconstructing the software architecture only from C++ source code. In order to accomplish this, we focus on class relationships, since they represent key elements at the source level. Specifically, our system parses C++ projects and statically analyzes the source code to extract class-related information. Then, it generates and visualizes a dependency graph that represents all the relationships between classes. On top of this graph, we apply clustering methods to identify high-level architectural entities. We use clustering algorithms like *Louvain*, *Infomap* and *Layered Label Propagation*, but also allow users to choose ad-hoc clustering via namespaces and folders. Finally, we have carried out a few case studies to assess and validate the utility of our architecture mining approach.

ΕΞΑΓΩΓΗ ΤΗΣ ΑΡΧΙΤΕΝΙΚΗΣ ΛΟΓΙΣΜΙΚΟΥ ΑΠΟ ΤΟΝ ΠΗΓΑΙΟ ΚΩΔΙΚΑ

Περίληψη

Η αρχιτεκτονική λογισμικού έχει βασικό ρόλο στον προσδιορισμό των προδιαγραφών και το σχεδιασμό της δομής ενός συστήματος, αποτελώντας τον συνδετικό κρίκο μεταξύ των απαιτήσεων και της υλοποίησης του. Παρόλα αυτά, κατά την διαδικασία ανάπτυξης λογισμικού, υπάρχει πιθανότητα να παραβιαστεί η αρχιτεκτονική του εκάστοτε συστήματος. Ως εκ τούτου, ενδέχεται να προκύψουν απροσδόκητες ή ανεπιθύμητες εξαρτήσεις μεταξύ των τμημάτων στον πηγαίο κώδικα. Επιπλέον, η κατανόηση της βασικής αρχιτεκτονικής σε συστήματα μεγάλης κλίμακας είναι απαραίτητη για την αποτελεσματική συντήρηση και συνεχή βελτίωση τους. Ωστόσο, είναι αδύνατο να αντικατοπτριστεί άμεσα στον πηγαίο κώδικα, καθώς δεν αποτελεί δομικό στοιχείο των γλωσσών προγραμματισμού. Επομένως, είναι αναγκαία η ανάπτυξης μεθόδων και εργαλείων που βοηθούν στην κατανόηση της δομής ενός συστήματος, εξάγοντας πληροφορίες σχετικά με την αρχιτεκτονική του απευθείας από τον πηγαίο κώδικα.

Η αντίστροφη μηχανική είναι η διαδικασία ανάλυσης ενός συστήματος με σκοπό τον εντοπισμό και την αναπαράσταση των σχέσεων μεταξύ των βασικών του στοιχείων. Η εξαγωγή της αρχιτεκτονικής αποτελεί υποσύνολο της αντίστροφης μηχανικής, η οποία στοχεύει στην ανίχνευση αφαιρέσεων που αντανακλούν τα βασικά στοιχεία του συστήματος σε υψηλό επίπεδο. Στο πλαίσιο αυτό, μέθοδοι αντίστροφης μηχανικής μπορούν να χρησιμοποιηθούν στην διαδικασία υπολογισμού της πραγματικής αρχιτεκτονική όπως αυτή αντικατοπτρίζεται, ώστε να συγκριθεί μετέπειτα με την αρχικά σχεδιασμένη αρχιτεκτονική.

Στην εργασία αυτή, προτείνουμε ένα εργαλείο το οποίο στοχεύει στην ανακατασκευή της αρχιτεκτονικής ενός συστήματος όπου ο πηγαίος κώδικας έχει αναπτυχθεί σε C++. Για να το πετύχουμε αυτό, εστιάζουμε στις σχέσεις μεταξύ των κλάσεων, καθώς

αποτελούν τα πιο βασικά στοιχεία σε επίπεδο πηγαίου κώδικα. Πιο συγκεκριμένα, το εργαλείο που έχουμε αναπτύξει πραγματοποιεί στατική ανάλυση σε C++ εφαρμογές, εξάγοντας δεδομένα σχετικά με τις κλάσεις και τις μεταξύ τους σχέσεις. Έπειτα, η προσέγγισή μας δημιουργεί και οπτικοποιεί ένα γράφο εξαρτήσεων που αντιπροσωπεύει όλες τις σχέσεις μεταξύ των κλάσεων. Στην συνέχεια, στον εξαγόμενο γράφο εφαρμόζουμε αλγορίθμους ομαδοποίησης, ώστε να δημιουργήσουμε μία αφαιρετική εικόνα του συστήματος εντοπίζοντας τα αρχιτεκτονικά του στοιχεία. Συγκεκριμένα χρησιμοποιήσαμε αλγορίθμους ομαδοποίησης, όπως τους *Louvain*, *Infomap* και *Layered Label Propagation*, ενώ δίνεται η δυνατότητα στους χρήστες να επιλέξουν ομαδοποίηση βάση συγκεκριμένων χαρακτηριστικών μέσω χώρων ονομάτων και φακέλων. Τέλος, έχουμε αναπτύξει αρκετά σενάρια χρήσης ώστε να ελέγξουμε την εγκυρότητα και την αποτελεσματικότητα της προσέγγισής μας.

Acknowledgments

First of all, I would like to thank my supervisor, professor of the University of Crete, Anthony Savidis, initially for trusting me and then for his continuous support and his valuable advice. I am also grateful to the professors Panos Trahanias and Polyvios Pratikakis for participating in the supervisory committee. I would also like to thank the Computer Science Department of Greece for offering a high level of academic education. In addition, I would like to thank each member of the PLATO laboratory individually for all of the talks, pleasant moments, and difficulties we faced together. Our frequent contact and collaboration made this experience much more enjoyable. I would also like to express my gratitude to my friends Michalis and Eleftheria (and all the others I will not mention one by one), as well as my sister, Ioanna, for their unwavering support during this period. Finally, most of all, I would like to thank my parents for never losing faith in me and always being by my side in any small or big step. Without them, I would not be the person I am today.

Στην οικογένειά μου

Contents

List of Figures

Chapter 1

Introduction

1.1 Software Architecture

Software architecture refers to the fundamental structures of a software system as well as the discipline of creating such structures and systems. Each structure represents the model of the software system on a high-level of abstraction. The architecture model plays a key role as a bridge between requirements and implementation, hiding all the implementation details, data representation and algorithms [1]. The software architecture comprises software components, relations among them, and properties of both components and relations [2].

According to Garlan, the role of software architecture is to provide an intellectually tractable guide to the overall system, enable designers to reason about the ability of a system to satisfy certain requirements, and propose a blueprint for system construction and composition. Particularly, software architecture plays a key role in at least six aspects of software development: (i) understanding, presenting a system at a high-level of abstraction that is easily understood, (ii) reuse, pointing out potential reuse components, frameworks, and patterns (iii) construction, indicating the major components and

dependencies between them, (iv) evolution, exposing the dimensions of a system's expected evolution, (v) analysis, including system consistency checking, conformance to quality attributes and dependence analysis, and (vi) management, since critical evaluation usually results in a better understanding of requirements, implementation strategies, and potential risks [1].

However, as software systems evolve over time, with new features being introduced and defects being repaired, it can lead to a degeneration of the architecture [3]. Thus, technologies that can be used to detect degeneration, repair degenerated systems, and provide further insight into the process of degeneration are significant.

1.2 Software Architecture Mining

Software architecture mining is a reverse engineering approach that aims to recreate viable architectural views of a software application from lower-level representations of the system, such as source code. The abstraction method for generating architectural elements frequently involves clustering source code entities into subsystems based on a collection of criteria that may be application dependent. In literature software architecture mining is used to refer with a number of alternative terms, including reverse architecting, architecture reconstruction, extraction, recovery, and discovery [4].

Based on software architecture roles mentioned by Garlan [1], software architecture mining can be used to meet a range of needs that can be summarized into the objectives mentioned below [4]. The primary goal is to re-establish software abstractions, helping reverse engineers document software applications and understand them. Secondly, it aims to provides reuse investigation since architectural views are useful to identify commonalities in the system. It has a key role in the conformance between the conceptual and the concrete architectures too. Furthermore, it can be used for architecture and implementation co-evolution to avoid architectural drifts [5]. Finally, it

aims to achieve architectural quality analyses as well as software evolution and maintenance.

According to Chikofsky and Cross [6], reverse engineering is defined as the process of analyzing a subject system to identify the system's components and their relationships and create representations of the system in another form or at a higher level of abstraction. In other words, reverse engineering is a technique to deduce design features from a system with little or no additional knowledge about its structure. Software architecture mining is a subset of reverse engineering which analyzes a subject system to identify meaningful higher-level abstractions that reflect its concrete architecture.

1.3 Problem Definition and Objectives

In a project's initial lifecycle phase, code can be fairly manageable and technologies are very familiar to the original authors. However, as technology improves, a need for legacy code maintenance emerges, which is usually handled by different developers than those who originally designed and developed it. Thus, developers have to evaluate the system in order to understand the existing, possibly poorly documented, source code for the maintenance and improvement of the software.

Moreover, on large scale systems, it is difficult for the entire system to be observed to ensure that an implementation adheres to design, which can lead to software bugs. This necessitates the inspection of the source code in order to have a high-level view of the system. Through software abstract view developers may compare the concrete with the idealized software architecture and recognize architectural drifts. Furthermore, graphical representations of code can provide a detailed view of the relationships among system components, allowing for the identification of undesirable dependencies as well as commonalities that can be abstracted and reused.

The intention of this thesis is to develop an architecture mining tool that aims to close the gap between conceptual and concrete architecture. Our tool analyzes source code statically, extracting relationships among systems elements and represent them as a dependency graph. We also use community detection algorithms to cluster the extracted graph's elements and infer the presence of high-level abstractions that correspond components in the designed architecture. This mined model can then be compared to an existing design to ensure the validity of the implementation, or it can be used as a blueprint for software that does not have any designed architecture.

The objectives of our tool are divided into three categories. First of all, it refers to the original software developers of a system assisting them in gaining more insights into how the source code evolved. In addition, it helps them to identify possible conflicts between the realized architecture and originally intended software architecture made during the design process. Secondly, it intends to be used for the reverse engineer of a not-owned disciplined system. It helps third-party developers to understand a system whose classes are known, at the surface, but they never participated in its development or inspected the code. This category primarily focuses on open-source software which provides a well-documented API but there is no clue on the architecture that source code reflects. Finally, it aims to evaluate a not-owed, totally unknown system too. It assists third-party software developers in gaining information for systems developed by others and no knowledge about them is available. It constructs a reverse engineering point of view on source code as well as its actually realized software architecture.

1.4 Thesis Structure

The rest of this work is organized as follows; In Chapter 2, we review some of the most popular related reverse engineering tools. Chapter 3 follows, where an overview of our architecture mining system is presented. The implementation details for the three main activities that we apply for software architecture reconstruction are stated in the

following three chapters. Particularly, Chapter 4 covers the extraction of class data from source code and their storage in a symbol table. In Chapter 5, a detailed description of the dependency graph composition based on the extracted data is provided. Chapter 6 presents class dependency graph analysis, configuration, and visualization. In Chapter 7 a description of the Case Studies, we have carried out in order to test the proposed architecture mining approach of our work is discussed. Chapter 8 concludes the work and identifies issues for further research work.

Chapter 2

Related Work

In this chapter, we review some related to our work tools. We focus on each tool which addresses (or not) issues relevant to maintenance which we solve in this thesis through our approach in order to reconstruct the actual software architecture of a given system. All of the tools mentioned below have reverse engineering as a primary or secondary feature. We begin by reviewing source code exploration and understand tool, then move on to software architecture mining tools, and finally, we review tools that specialize on software architecture design but also provide reverse engineering capabilities.

2.1 Source Code Analysis Tools

This section goes through some visualization tools that enable developers to see the source code at a high level. These tools are used to comprehend and navigate source code, as well as the relationships between its different components; however, they do not provide architecture mining capabilities.

Sourcetrail

Sourcetrail [7] is an open-source, cross-platform source explorer. It uses static analysis on C, C++, Java and Python source code with the view to help software engineers to understand the source code and the relationship between different components. It provides both overview and details within a user interface that combines interactive graph visualization, code view and symbol search. Specifically, Sourcetrail builds a dependency graph after indexing the source code files and builds a graphical overview of the source code. Finally, it is built in an extendable way, so it could be extended to support more programming languages.

Understand

Understand [8] is a static code analysis tool by Scientific Toolworks for visualizing source code through graphs, charts, and metrics. Specifically, it aims to achieve complete code navigation, control flow graph generation, metrics generation, code comparison, checking on the adherence of a code to some specific coding standards and code reengineering for a variety of programming languages like C, C++, Java, Jovial, Pascal, ADA, .NET, and more. Regarding the reverse engineered graphs, logic and dependency as well as class and function calls diagrams are generated. Finally, Understand tool has a command line interface which allows it to be integrated with any of the existing tool chain that the software companies normally use.

Source Insight

Source Insight [9] is an advanced code editor and browser from Source Dynamics. It bills itself not just as an editor but as a tool for understanding a large source code base. This tool comes with built-in analysis for C/C++, C#, and Java programs. Source Insight parses the source code and maintains its own database of symbolic information dynamically. As an analysis tool, it can view reference trees, class inheritance diagrams, and function call

trees. Auxiliary panel windows, such as relation, context, and symbol windows, are also included for fast navigation of source code and source information. Source Insight was designed for massive, demanding, real world programming projects. In fact, major technology companies are using it to develop commercial hardware and software products.

Class Visualizer

Class Visualizer [10] is an interactive class diagrams generator from Java and Kotlin bytecode. The generated class diagram shows all the relations of the selected class, including parent and child classes as well as outbound and inbound references (nested classes, associations, dependencies, annotations, thrown exceptions). In addition, class browsing is allowed, providing its UML and tree view for a detailed look into every class-related element. Class Visualizer does not support as many features as the aforementioned tools, but it provides a simple visual representation of program class interaction relationships.

2.2 Software Architecture Mining Tools

This section presents the detail discussion on tools that, apart from source understanding, can be used to reconstruct and recognize system software structure.

Imagix 4D

Imagix 4D [11] is a comprehensive program understanding tool for C, C++ and Java programs. It automates the review and browsing of code, enabling developers to reverse engineer software that is big, complicated, unfamiliar, or old. It provides views to rapidly check and systematically study software at any level ranging from high-level design to the details of its build, class, and function dependencies. Specifically, it offers subsystem

architecture views, which promote understanding of the fundamental aspects about how the software is structured, and structure views, imagine the interactions of files, classes and functions. Control flow as well as UML diagrams are also assisted. Software metrics are also available, as well as the ability to generate documents in ASCII, HTML or RTF. Finally, Imagix 4D has a long list of commercial customers, including Bosch, GE Healthcare, Hitachi, Nissan, and Samsung.

Rigi

The Rigi [12] is a research tool that provides functionality to reverse engineer software systems. Rigi helps to analyze, interactively explore, summarize, and document large systems. To succeed that, Rigi provides the extraction of static elements from software systems, a repository to represent and store these elements, and analyses and visualization of them. Specifically, Rigi includes parsers to extract information from C, C++, PL/AS, COBOL, and LaTeX source code. It offers an exchange format with a graph-based data model to store the extracted information, analyses to transform and abstract information as well as a scripting language and library to automate the process. Finally, Rigi includes a visualization engine (Rigiedit) that allows users to interactively explore and manipulate information in the form of typed, directed, hierarchical graphs.

ARMIN

The ARMIN [13] (Architecture Reconstruction and Mining) is an architecture mining tool developed by the Software Engineering Institute and Robert Bosch Corporation. Its architecture reconstruction approach is split into four major phases: Information Extraction, Database Construction, View Fusion, Architectural View Composition. The Information Extraction phase involves analyzing the system to construct its static view, extracting information from the source files, and its dynamic view, observing system execution. The set of extracted views are manipulated to create fused views. The final phase consists of two primary activity areas: the visualization and interaction area, which

provides a mechanism that allows the user to visualize and manipulate views interactively, and the command script definition and interpretation area, which allows the user to write scripts for elements aggregation.

Sonargraph-Architect

Sonargraph-Architect [14] is a general-purpose static analysis tool on C#, C/C++, Java and Python 3 code by hello2morrow. It provided metrics and dependency visualization as well as automated architecture checks. In addition, a Groovy based scripting engine is supported for user defined metrics or check for specific issues in the code. Furthermore, it provides a checker for duplicate code as well as a cycle break-up computer to identify the dependencies that are needed to cut to untangle cyclic dependencies. Finally, Sonargraph-Architect assists virtual refactoring, simulating the effect of a change without moodily the source code.

Sotograph

The Sotograph [15] product platform by hello2morrow consists of the products Sotograph, Sotoarc and Sotoreprt. It is a static code analysis on C/C++, C#, Typescript, Java, PHP and ABAP/ABAPObjects source code. Sotograph detects a wide range of potential problems in source code using predefined queries and metrics. Particularly, it provides the detection of all classes, files, packages and subsystems which are strongly coupled by cyclical relationships as well as the identification of large duplicated code blocks. Moreover, it detects methods, classes, files, packages and subsystems which are suspicious because of their size, coupling or complexity. In addition, Sotograph makes use of architecture models defined in Sotoarc. Sotoarc provides the code structure visualization as hierarchies (trees) of modules, packages and files. Besides the user can describe by graphical means the specified software architecture of a software system. By doing so the tool is immediately comparing this intended architecture with the implemented code structure and is highlighting all architecture violations. The key

difference between Sonargraph-Architecture and Sotograph is that the former has more metrics capabilities since it can apply user-defined metrics, while the latter performs architecture conformance check.

2.3 Software Architecture Design Tools

This segment addresses tools for software system architecture design that also have reverse engineering capabilities. Apart from the reverse engineering functionality offered by these tools, their architecture design feature is also investigated, as our tool approaches system architecture view from the opposite perspective.

Enterprise Architect

Enterprise Architect [16] is a visual modeling and design tool based on UML from Sparx System. Enterprise Architect supports the design and construction of software systems. It also supports modeling business processes and modeling industry-based domains. Enterprise Architect supports code generation templates in numerous languages like Action Script, C, C#, C++, Java etc. This tool, provides reverse engineering support of existing applications and round-trip engineering for a number of popular programming languages.

Visual Paradigm

Visual Paradigm [17] enables developer teams and individuals to model the development process in an effective manner. It provides code engineering features and supports current modeling languages and standards. Developers can use the template designer to create system documentation or to design class diagrams. In addition to modeling support, it provides report generation and code engineering capabilities including code

generation. Moreover, it can reverse engineer diagrams from code, and provide round-trip engineering for a variety of programming languages.

Altova UModel

Altova UModel [18] is a commercial UML modeling software tool from Altova. UModel can be integrated with Eclipse and Visual Studio as a plug-in. UModel supports UML 2 diagram types and adds a unique diagram for modeling XML Schemas in UML. *UModel* also supports *SysML* [19] for embedded system developers, and business process modeling (BPMN notation) [20] for enterprise analysts. UModel includes code engineering functionality including code generation in Java, C#, and Visual Basic programming language. *UModel* supports model interchange with other UML tools through the XMI standard, integrating with revision control systems. It also supports reverse engineering of existing applications, and round-trip engineering.

Chapter 3

System Overview

The main goal of our work is to provide a tool that is able to extrapolate and reconstruct the concrete software architecture of a given software system directly from source code. In order to succeed that, it collects the architectural components from the source code and analyzes the dependencies and relationships among them.

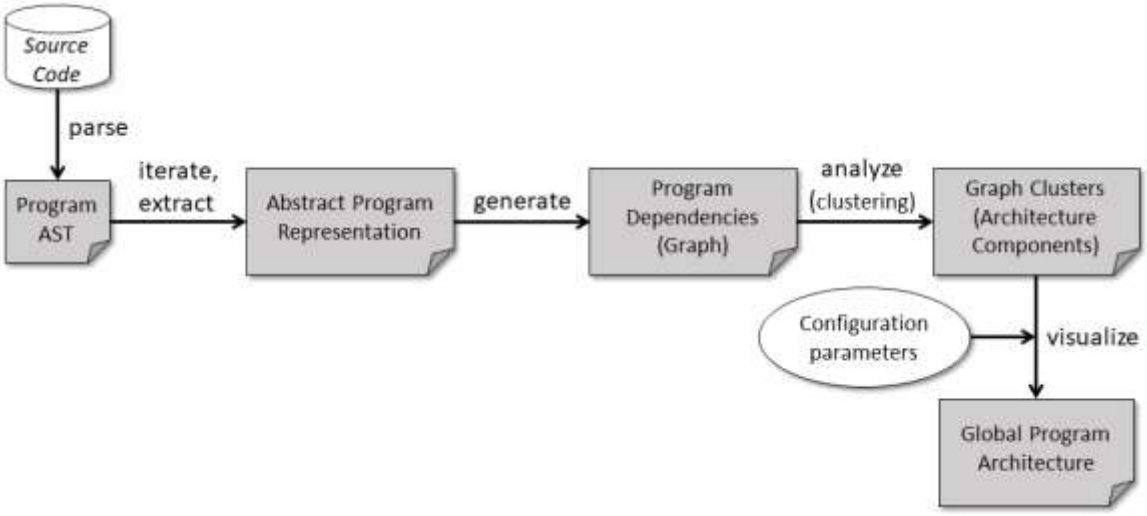

Figure 1 - System overview

Particularly, our system uses a compiler front-end in order to parse the given source code and build its abstract syntax tree (AST). Moreover, it extracts data regarding program classes since they constitute the main architectural components. The extracted data indirectly describes the dependencies between classes, reflecting the abstract relational representation of the program. Then, using this data, it generates a class dependency graph, which represents the relationships among classes as edges. Finally, it is visualized after it has been analyzed, reflecting the software architecture of the system. The system described above is depicted in Figure 1.

3.1 Dependencies

As already mentioned, program classes are the key architectural components of a software application. So, in order to reconstruct the architecture of a system, it is unavoidable to examine the relations among its program classes. Thus, we would like to study how often a class is used in another class body and what it serves. The use defines a relationship between these two classes; particularly, it represents a dependency from the examined class to the used class.

To succeed that we have to parse all the classes of the subject system and examine all the dependencies that will result from inheritance, member fields, friend declarations to other classes and methods, as well as outer class that probably includes a nested class. In addition, the objects that are deployed to support the class functionality should be studied. Thus, we analyze its method arguments and local variables as well as the member access expressions are applied to object operands in methods body.

Therefore, the dependencies among classes can be divided into two categories: the dependencies that are defined directly by the developer (inheritance, field definitions, friends) and these that arise from objects that are deployed in order to provide the class functionality (method variables, member access expression). However, in our

implementation we handle them equally, since all of them have the same impact in terms of software design extraction.

3.2 Architecture

In this section the architecture that we have designed for our system will be discussed.

3.2.1 Dependency Graph Generator

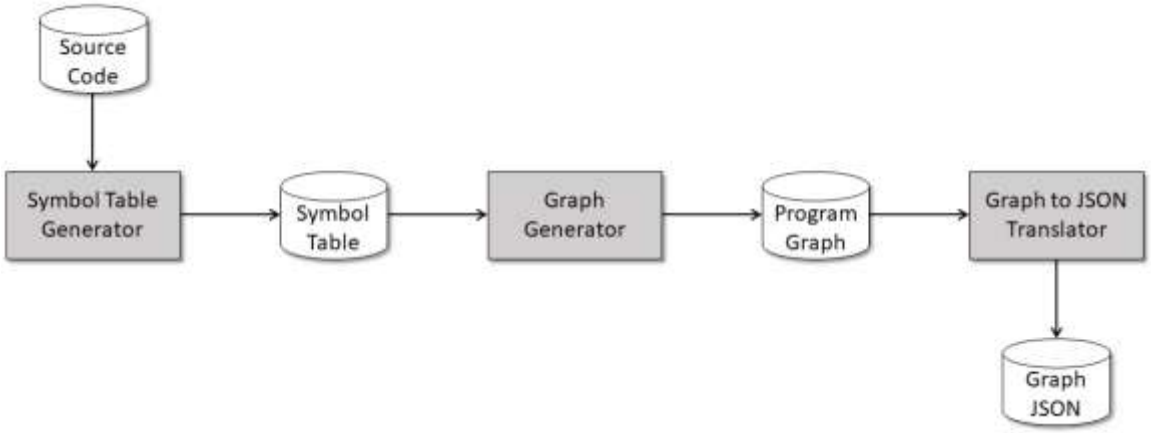

Figure 2 - Dependency graph generator architecture

The dependency graph generator is composed of three components: the symbol table generator, the dependency graph generator, and the graph to JSON translator, as shown in Figure 2. Regarding the first, it takes the program source code as input and constructs the program symbol table by extracting all class-related information directly from the source code. Following that, the class-dependency graph generator constructs the program's graph representation, by traversing the symbol table data. This graph represents all of the program class dependencies. Finally, the generated graph is translated and stored in JSON format, which will be used as input by the graph visualizer.

3.2.2 Dependency Graph Visualizer

The dependency graph visualizer consists of two components: the graph renderer and the interactive configurator. These two elements compose the graph visualizer user interface. Regarding the renderer, it gets as input the dependency graph that generated in previous section on JSON format. Based on these data, it builds the graphic model of the graph and render it. Regarding the configurator, it intercommunicates with the configuration data, which onchange notify the renderer in order to update the graph. This interconnection is achieved through the observer design pattern.

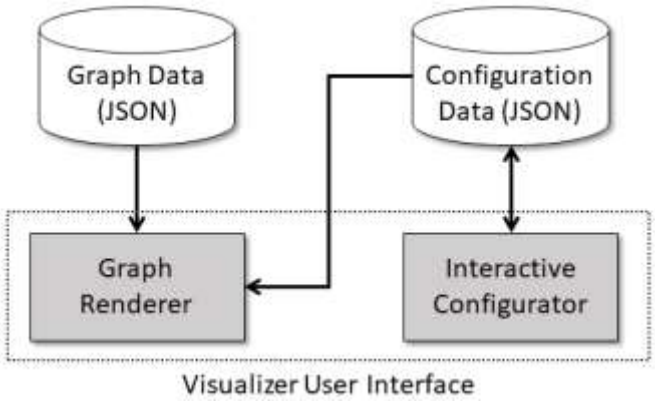

Figure 3 - Graph visualization and configuration architecture

3.3 System Input

Our system supports two different types of input, a directory that includes project sources or a compilation database. Regarding source files directory, we implement a loader that iterates over the elements of a directory as well as its subdirectories and loads all the C++ source files. However, if complete information on how to parse a translation unit are required, a JSON compilation database [1] is used. A compilation database is a JSON file that contains an array of “command objects”, each of which specifies how a translation unit is compiled in the project.

Specifically, our system takes five parameters. The first parameter specifies whether the project will be loaded from a directory (“--src”) or from a compilation database (“--cmp-db”). The second parameter specifies the path to the sources or the compilation database. The third and fourth parameters are optional, and they define the ignored file paths and namespaces, respectively. More details regarding ignored file paths and namespaces will be discussed in section 4.4.

Chapter 4

Class Data Extraction via the Compiler Front-End

4.1 Clang Compiler Front-End

In compiler design, the compilation is separated into three phases whose major components are the front-end, the optimizer and the back-end. The front-end is responsible for the analysis of the source code. In concrete terms, it parses source code, checking it for errors, and builds a language-specific Abstract Syntax Tree (AST) to represent the input code. Then, the AST is optionally converted to a new representation for optimization, and the optimizer and the back-end are responsible for the machine code generation.

Figure 4 - Three phases of compilation

The Clang [22] project provides a compiler front-end and tooling infrastructure for languages in the C language family for the LLVM project [23]. The latter provides the

required infrastructure in order to create tools that syntactic and semantic information about a program is needed.

For our system we will use Clang library to parse and analyze C++ programs.

4.1.1 Clang Abstract Syntax Tree

As mentioned above, Clang is a library that converts a C++ program into an AST and supports manipulations on it. Regarding its AST structure, it is built using three core node classes: declarations (*clang::Decl*), statements (*clang::Stmt*) and types (*clang::Type*). It is notable that AST nodes are modeled on a class hierarchy that does not have any common base class to inherit. As a result, there is no common interface for visiting all the nodes in the tree. However, each node dedicates some methods that allow tree navigation.

Apart from that, the *ASTContext* class is provided which keeps information about the AST of a translation unit that is not stored in its nodes, including the source manager and the identifier table. Moreover, it forms the entry point into the AST.

4.1.2 Abstract Syntax Tree Traversal

In order to traverse and manipulate the AST, Clang offers two abstractions: the *ASTMatchers* [24] and the *RecursiveASTVisitor* [25]. Both of them use LibTooling [26], a C++ interface aimed at writing standalone tools based on Clang.

The *ASTMatchers* provides a domain specific language (DSL) [27] to create predicates on Clang AST. This DSL is written in and can be used from C++. It matches AST nodes as well as allows access to the node C++ interface. When the predicate is matched the attached *MatchCallback* is executed. Moreover, *ASTMatchers* are classified in three categories:

node and narrowing matchers that match specific types and attributes respectively as well as traversal matchers that allow traversal between AST notes.

The *RecursiveASTVisitor* offers a traverse to the nodes of Clang AST in a depth-first manner. In order to succeed that, extending the class and implementing the desired *VisitNodeType(NodeType *)* method is needed.

4.2 Data Extraction Implementation

Given that we want to extract all the relative information about program classes the use of a compiler front-end is unavoidable. In order to parse the source code and build its AST we use the Clang front-end which is analyzed before. Furthermore, *ASTMatchers* as well as *RecursiveASTVisitor* are used to manipulate and traverse the produced AST.

Specifically, we define four node matchers (cxxRecordDecl, fieldDecl, cxxMethodDecl, varDecl) to match program classes, fields, methods, as well as method arguments and locals. Narrowing matchers are also used to frame the record declarations (isClass, isStuct). In addition, associated MatchFinder:: MatchCallbacks are defined in order to access AST nodes and extract all necessary information. Besides that, we implement a *RecursiveASTVisitor* to detect and process the member access expressions that are located in class methods.

4.2.1 Classes

As previously mentioned, classes play a key role in the architecture design of a system. Thus, their data extraction for further analysis is significant. For this reason, a *cxxRecordDecl* is defined, which matches all the struct, class and union representations in the program. Regarding the matched elements, information about their potential bases, friend classes and methods as well as their outer class for nested classes are extracted.

Additional information, such as the source location, namespace, and class qualified name, are also retrieved.

From the matched record declarations, we exclude a part of them since they are not needed in a program abstract representation. Firstly, we ignore all the matched unions and anonymous classes because their use is considered as a secondary type for local use without providing any architectural value. Additionally, we ignore all the class declarations that do not have any definition. About the latter, template definitions without a defined body are exempted since they may provide a common interface for further specializations.

```
1   template <class T, int size>
2   class StaticArray {
3       T m_array[size];
4       // ...
5   };
6
7   template <int size>
8   class StaticArray <double, size> { /* ... */ };
9
10  template<> class StaticArray<char, 12>  { /* ... */ };
11
12  int main() {
13      StaticArray<int, 6> intArray;
14  }
```

Figure 5 - Example code with template definition and specialization

Figures 6 and 7 depict the Clang AST of the example in Figure 5. Specifically, Figure 6 shows the template definition node of the *StaticArray* class as well as their parameter declarations. A few nodes later, we can see the template instantiation specialization that is generated automatically by the definition of the variable *intArray*. At the beginning of Figure 7, we see the partial specialization defined in line 7 of the example. This specialization takes as an argument the type of the array (*double*), but its size is still

Figure 6 - Template definition and instantiation specialization abstract syntax tree

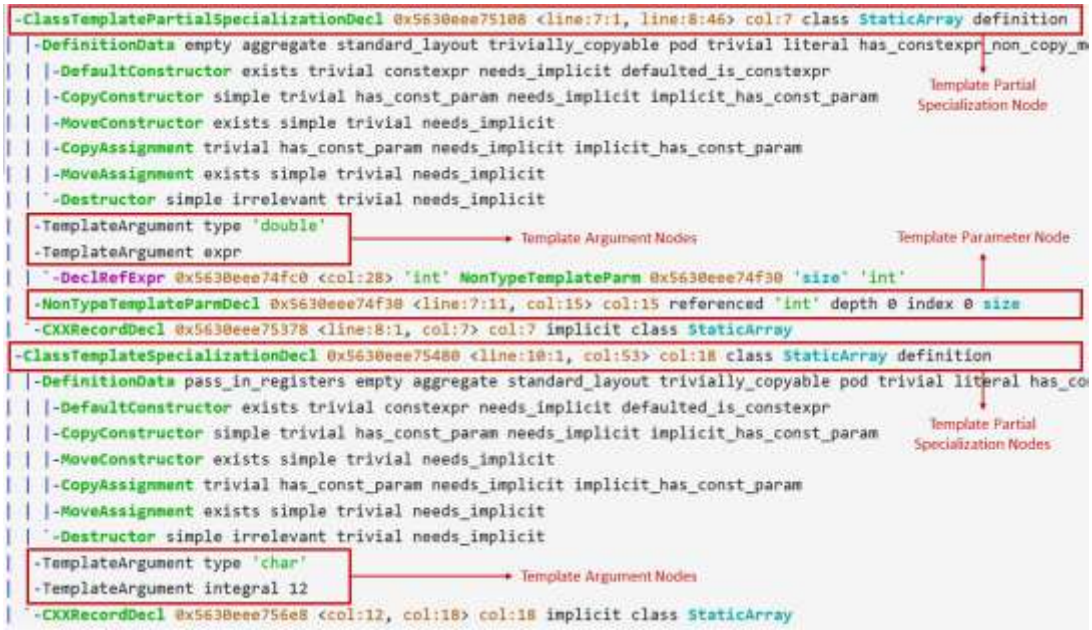

Figure 7 - Template partial and full explicit specialization abstract syntax tree

an undefined parameter. Finally, we can see that the full specialization defined in line 10 is represented in the AST as a *ClassTemplateSpecializationDecl* node. It is notable that in contrast to the explicit specializations defined by the user, the specialization produced by the compiler as a result of an instance of the template class is a child node of the initial template definition.

In view of the AST structure, we arrive at the following relationships between templates. Staring from a template definition, all of its specializations, regardless of type, define this template definition as their template parent. Apart from that, additional dependencies are born to the template argument types. It is worth mentioning that Clang provides a reference to their template definitions parent only for instantiation specialization nodes. For the explicit specializations, we construct the qualified name of the template definition based on the template specialization name, and map it to the corresponding element on the symbol table that returns all the required information. More details regarding symbol table use and structure will be discussed in section 4.3.

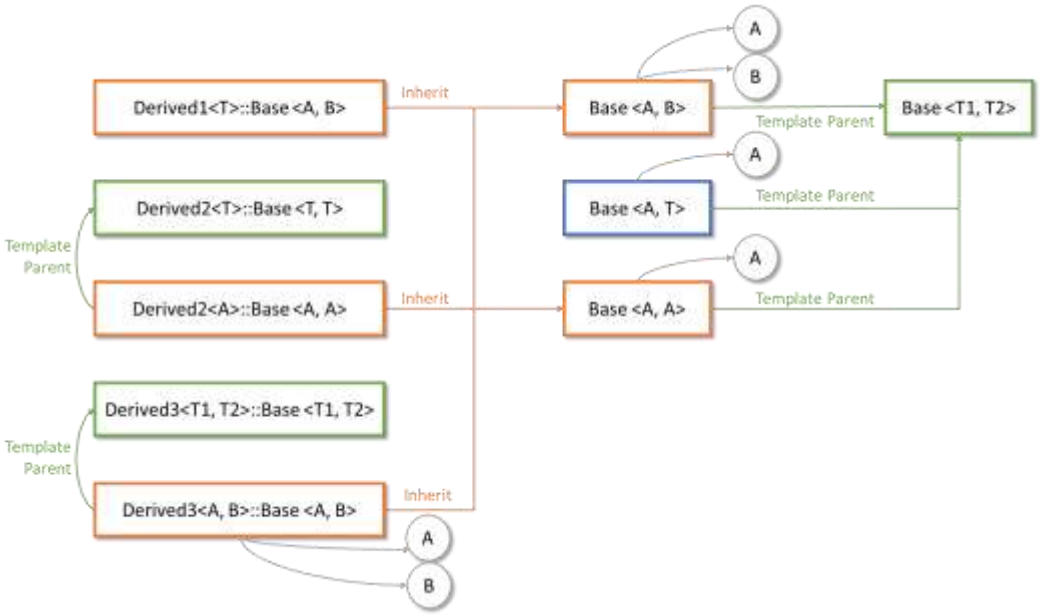

Figure 8 - Relationships between template definitions on specializations and inheritance
The template definitions are shown in green, the template partial specializations in blue, and the template full specializations or Instantiations in orange. The gray nodes can be of any class type.
Furthermore, the same result will be produced when Derived2 inherits from a partial specialization of class Base.

Besides that, information about class inheritance is extracted. Specifically, all the bases of a class are retrieved. About classes that derive from a template definition or partial specialization, Clang makes no reference to their bases. However, when the derived template class is instantiated, the inheritance relationship between the derived instance and the explicit full or instantiation specialization arises. As a consequence, no relationship is lost since it is born during class instantiation.

The relationships described before are illustrated in Figure 8.

4.2.2 Methods

Apart from classes, an AST matcher as well as the corresponding callback are defined in order to extract class methods relevant information. Specifically, a *cxxMethodDecl* matcher is defined, which matches all the method declarations in the program. Regarding these methods, information about their parent class, return type and member access expressions that are used in the method body are collected. Some additional information is also retrieved, such as the source location, namespace, and method qualified name. Finally, method arguments and locals are analyzed by another matcher.

From these matched elements, a number of them are excluded. Particularly, we ignore all the method declarations which do not provide their definition body because they are not needed for further analysis. Additionally, all the methods that their parent class is noted as ignored, are excluded too.

Concerning the matched methods, Clang categorizes them as trivial and non-trivial. In particular, trivial methods are those that are automatically generated by the compiler, including constructors, destructor as well as operator overloading methods. Non-trivial methods, on the other hand, are those that are defined explicitly by the user. This separation is very useful for our analysis because trivial methods have no needful

information about software architecture since the programmer is unaware of their existence, so they will be ignored in the next step of dependency graph composition. Furthermore, we manipulate template methods as if they were regular methods, ignoring dependencies between template definitions and specializations because they are irrelevant to program analysis. Their only difference from the other user defined methods is that when they are specialized, dependencies to template arguments that are not primary data types emerge.

As stated in section 3.1, it is critical to study the uses of other class types on methods through member access expressions. To achieve this, we use a *RecursiveASTVisitor* that, given the method body, detects member access expressions. Specifically, when a member of a class instance is accessed, a dependency on this class arises. In the case of member access from a method call a dependency to the return type turns out. For example, in line 16 of the program in Figure 9, the expression "*a->getB().c*" results a dependency on *class A* due to the subexpression "*a->getB()*" as well as a dependency on *class B* due to the expression "*getB().c*".

```
1   struct C { /* ... */ };
2
3   struct B {
4       C* c;
5       // ...
6   };
7
8   class A {
9       B b;
10  public:
11      B getB() { return b; }
12  };
13
14  struct D {
15      void method(A* a) {
16          C* c = a->getB().c;
17          // ...
18      }
19  }
```

Figure 9 - Member access expression example

4.2.3 Fields

Class fields play a vital role in the relationship among classes. Therefore, a *filedDecl* matcher is declared to match them. From these elements, we ignore all the field declarations whose data type is primitive, as well as those whose parent class is excluded. Additionally, on template instantiation, there are no extra dependencies to the fields that are declared on template definition, since they are insinuated on the class level dependency. Finally, for the fields whose data type is a pointer to a program class, the pointed type is retrieved.

4.2.4 Formal Arguments and Locals

In addition to member access expressions, dependencies can arise from the formal arguments and locals of class methods. Thus, a *varDecl* Mather is declared to match method formal arguments and locals. We ignore some of the matched variables since they are not needed for our analysis. Specifically, we ignore the elements that have a primary data type. The variables that their parent method or class is noted as ignored, are excluded too. Lastly, for the variables which data type is a pointer to a class, the pointed type is retrieved.

4.3 Symbol Table

In order to store these extracted data, a symbol table is provided. On this structure, each symbol, that we match as described in the previous section, is associated with its relative information. Specifically, a *Symbol Table* class is implemented (Figure 10). This class consists of a hash map with a unique identifier for the symbol as the key, and a *Symbol* instance as the value. As identifier, the symbol's qualified name is used. In order to install as well as lookup an element on the symbol table, the *Install()* and *Lookup()* methods are

provided. So, a generic *Symbol* class is implemented, which defines all the common characteristics that a symbol has. Its definition is shown in Figure 10. *Symbol* class has three derived classes: *Structure*, which represents all the classes that are met on the program, *Method*, which represents all the methods on the program, and *Definition*, which represents the fields and variables that are defined on a class body. The inner structure of these classes is depicted in Figure 11, and is described in the following paragraphs.

```
class SymbolTable {
    std::unordered_map<ID_T, Symbol *> ST;

    Symbol *Install(const ID_T &id, Symbol *symbol);
    Symbol *Lookup(const ID_T &id);

    void Accept(STVisitor* visitor) const;
}
class Symbol {
    ID_T id;
    std::string name;
    std::string nameSpace;
    SourceInfo srcInfo;
    ClassType classType;
}
```

Figure 10 - SymbolTable and generic Symbol classes code view

Regarding *Structure* class, six internal symbol tables are defined to store class methods, fields, bases classes, friends, nested classes, and template arguments (included in *Template* class). In the case of friends, a reference to the friend class or the parent class of the friend method is stored. Other kinds of friends, such as program functions, are ignored since they do not offer any needful information for the dependencies among classes. Additionally, for the case of template specialization a pointer to the template parent is provided. A pointer to the outer class for the nested classes is provided too. Lastly, it stores the structure type described in section 4.2.1.

Regarding *Method* class, three internal symbol tables are defined in order to store methods formal arguments, locals as well as template arguments (in *Template* class). A

pointer to the return type *Structure* is provided too. Additionally, there are stored all the member access expressions that are met on each method body. Specifically, for each member access expression, a table with its members as well as their type is stored. Moreover, it stores the method type that is described in section 4.2.2.

Finally, *Definition* class stores a pointer to the type of the variable.

```
class Structure : public Symbol {
    StructureType structureType;
    Structure *nestedParent;
    Template<Structure> templateInfo;
    SymbolTable methods;
    SymbolTable fields;
    SymbolTable bases;
    SymbolTable friends;
    SymbolTable contains;
}
class Method : public Symbol {
    class MemberExpr {
        SymbolTable members;
     };

    MethodType methodType;
    Structure *returnType;
    SymbolTable arguments;
    SymbolTable definitions;
    Template<Method> templateInfo;      // tempalte parent is ignored
    std::map<std::string, MemberExpr> memberExprs;
}
class Definition : public Symbol {
    Structure *type;
}

template<typename Parent_T> class Template {
        Parent_T* parent = nullptr;
        SymbolTable arguments;
}
```

Figure 11 - Structure, Method and Definition classes code view

As previously stated, classes are the primary elements used in program architecture reconstruction. As a result, the main symbol table is made up of *Structure* elements that represent these program classes. These entries, have internal symbol tables for the *Symbols* defined in their bodies as well. This structure creates a complete symbol table containing all of the information extracted from the AST via the compiler front-end. Additionally, through the references among class entries, the relational program representation is formed.

```
1   class Point { int x, y; };
2
3   class Shape {
4       Point position;
5   public:
6       Point& getPosition() { /* ... */ }
7   };
8
9   class Circle : public Shape {
10      unsigned radius;
11  };
```

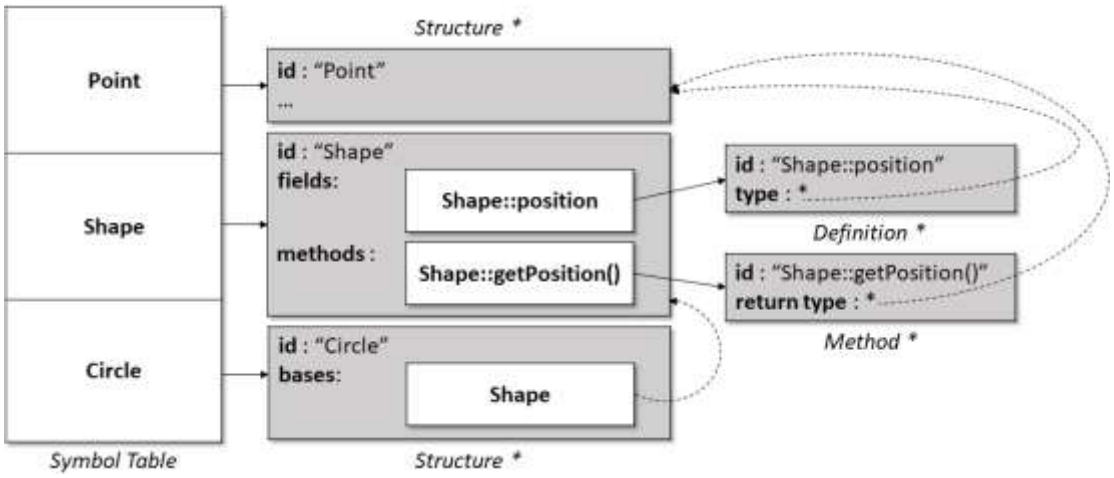

Figure 12 - Symbol table view
With white color the main as well as the inner symbol tables are represented. With gray color symbols related data at each case are represented.

The use of a generic base class *Symbol* as well as a symbol table such as main storage structure as inner data representation in classes is anything but random. On the contrary, it facilitates the traverse on all its levels with a common interface with great ease. Based

on this infrastructure, a symbol table visitor is defined in order to make the traversal over its elements possible. In Figure 13 symbol table visitor definition is illustrated.

```
1   class STVisitor {
2   public:
3       virtual void VisitStructure(Structure *s) = 0;
4       virtual void VisitMethod(Method *m) = 0;
5       virtual void VisitDefinition(Definition *s) = 0;
6   };
```

Figure 13 - Symbol table visitor definition

In Figure 12 the generated symbol table for the code example at the same figure is illustrated.

4.4 Ignored Namespaces and File Paths

So far, we have ignored certain elements that we believe are not needed further analysis because they have zero architectural meaning. Besides them, a mechanism is supported in order to ignore a set of elements regardless of their meaning. Specifically, we support the exclusion of groups of elements that are defined at a specific namespace. The exception of an entire folder is supported too. Specifically, during the source code analysis we examine if the matched element belongs to a namespace or file path that are nodes as excluded. It is notable that we cannot ignore them earlier since they consist parts of program's AST.

We have already defined a number of file paths as well as namespaces that we think that have to be ignored for every project. Particularly, we exclude some standard file paths that have to be ignored on Microsoft Windows including C++ standard libraries and Visual Studio files. The *std* namespace is excluded too. Furthermore, we offer the ability to our system user to define the namespaces and file paths that they may like to exclude from the analysis. Specifically, a file with the excluded absolute paths for directories with

sources as well as a file with the ignored namespaces can be passed optionally as command arguments.

Chapter 5

Generic Class-Dependency Graph Composition

In the previous section, the procedure of class data extraction using the Clang front-end is represented. In this way, the subject system is described with the help of a symbol table that allows us to store all the class relative information that generates interrelationships among them. But it is vital the representation of these data in a way that will facilitate further analysis. For this reason, we decide to compose a class-dependency graph. Specifically, we create a directed graph where classes are represented as nodes and relationships as edges between two nodes. To succeed that we implement a generic graph structure that represents a directed graph with weighted edges among its nodes.

5.1 Generic Graph

```
1   class Node;
2   class Edge;
3
4   class Graph {
5       std::set<Node *> nodes;
6   public:
7       void Accept(GraphVisitor* visitor);
8       // ...
9   };
10
11  class Node {
12      untyped::Object data;
13      std::list<Edge *> outEdges;
14  public:
15      void AddOutEdge(Node* to, const Edge::DependencyType& depType, Edge::Cardinality card);
16      // ...
17  };
18
19  class Edge {
20  public:
21      using Cardinality = unsigned;
22      using DependencyType = std::string;
23  private:
24      std::map<DependencyType, Cardinality> dependencies;
25      Node *to;
26  public:
27      void AddDependency(const DependencyType& depType, Edge::Cardinality card);
28      // ...
29  };
```

Figure 14 - Generic graph definition

In order to construct the class-dependency graph model of the subject system, a generic graph data structure consisting of nodes and directed weighted edges is implemented. In short, graph nodes are stored as records and each one of them stores a list of outgoing edges. The storage of additional data on the nodes is allowed too. Additional data regarding the capacity are also stored in edges. Specifically, a *Graph* class is defined which stores a set with graph nodes. These nodes are instances of *Node* class, that stores node related data as well as a list with outgoing edges of this node. The storage of node-related data is achieved through the use of the *Object* class, which represents an untyped dynamic object. The use of this object, allows us the storage of data with the key-value pair model regardless of their number and data type. Furthermore, *Edge* class is defined

to represent these outgoing edges. This class consisting of a reference to the pointed node as well as a structure that stores pairs of the cardinality of the edge and its potential dependency type. The definitions of the classes described above are shown in Figure 14.

Besides that, a *Visitor* over this generic graph is implemented in order to provide an interface for traversal over graph elements. Its definition is illustrated in Figure 15.

```
1   class GraphVisitor {
2   public:
3       virtual void VisitNode(Node* n) = 0;
4       virtual void VisitEdge(Edge* e) = 0;
5   };
```

Figure 15 - Graph visitor definition

5.2 Dependency Graph Composition

In order to create the class-dependency graph of the subject system, a traverse of the generated symbol table is required. To succeed that, we use a class that implements the symbol table visitor which is shown in Figure 13. Specifically, it implements all the abstract visit methods in order to traverse over the symbols of the table. It also contains a graph instance that holds the generated graph as well as some secondary definitions that aid in graph composition. Its definition is depicted in Figure 16.

The key idea is that by visiting all of the symbols in the symbol table recursively, we convert their related data to untyped object format and compose the graph, generating a node for each class. Particularly, when we visit a symbol with class type, if it is the first time, a *Node* class instance is created and added in the graph. Then we parse all of its related data and store it as key-value pairs in the node data object. For each symbol on the symbol tables that stores bases, friends, template arguments as well as template and nested parent, we call recursively the VisitStructure() method for the pointed class type. Similarly, the *VisitMethod()* and *VisitDefinition()* are called for class methods and fields

respectively. When we visit a method, its information is copied in an object instance as well. For each related to method *Symbol,* the appropriate visit method is called too. When the method analysis is completed, the composed object is returned to the caller class to merge it into its node object. In like manner the *Definition* visits are manipulated.

```
1   class GraphGenerationSTVisitor : public STVisitor {
2       Graph graph;
3       Node* currNode;
4       untyped::Object innerObj;
5       Edge::DependencyType currDepType;
6   public:
7       virtual void VisitStructure(Structure* s);
8       virtual void VisitMethod(Method* m);
9       virtual void VisitDefinition(Definition* s);
10      Graph& GetGraph();
11  };
```

Figure 16 - Symbol table visitor implementation for graph generation

Concerning dependencies, we categorize them based on the relationship they convey between classes. Specifically, eleven types of dependencies are defined: *Inherit, Friend, Nested Class*, *Class Field*, *Class Template Parent*, *Class Template Argument*, *Method Return*, *Method Argument*, *Method Definition*, *Member Expression,* and *Method Template* Argument. The visit caller for each *Structure* specifies the type of dependence as well as the node that is dependent on the structure being visited. On the other side, when a *Structure* is visited define an edge with the given dependency from the dependent code to itself. With this technique, dependency edges among class nodes arise. Finally, self-dependencies as well as trivial method analysis are ignored.

The pseudocode in Figures 17, 18, 19 sums all the steps analyzed before.

```
void VisitStructure(Structure s, DependencyType outerDep, Node outerNode)
   if s does not exist in Graph then
```

```
      create node and add it in graph
      copy symbol data from s to node (id, name, namespace, source info, …)
      if hasTemplateParent(s) then
         VisitStructure(s.templateParent, classTemplateParent_dep_t, node)
      end
      if hasNestedParent(s) then
         VisitStructure(s.nestedParent, nestedParent_dep_t, node)
      end

      foreach ST in s.data do
          foreach [id, symbol] in ST do
             DependencyType dep_t = dependency that the ST expresses
             if symbol.typeof(Structure) then
                VisitStructure(symbol, dep_t, node)
             else if symbol.typeof(Definition) then
                node.data.add(VisitDefinition(symbol, dep_t, node))
             else
                node.data.add(VisitMethod(symbol, node))
             end
         end
      end
   end

   if outerNode && isNotSelfDependency(s, outerNode) then
      add edge from outerNode to this structure node with dependency type
      outerDep
    end
end
```

Figure 17 - Visit Structure symbol pseudocode

```
Object VisitMethod(Method s, Node outerNode)
   Object data
   copy symbol data from s to data object (id, name, namespace, …)

   if hasReturnType(s) then
      VisitStructure(s.returnType, MethodReturn_dep_t, outerNode)
   end

   foreach ST in s.data do
      foreach [id, symbol] in ST do
         DependencyType dep_t = dependency that the ST expresses
         data.add(VisitDefinition(symbol, dep_t, outerNode))
      end
   end

   data.add(VisitMemberExpr(s.memberExpr, MemberExpr_dep_t, outerNode))
   return data
end
```

Figure 18 - Visit Method symbol pseudocode

```
Object VisitDefinition(Definition s, DependencyType outerDep, Node outerNode)
```

```
    Object data
    copy symbol data from s to data object (id, name, namespace, …)
    VisitStructure(s.type, outerDep, outerNode)
    return data
end
```

Figure 19 - Visit Definition symbol pseudocode

The result is the representation of the subject system as a graph where all class related data is stored in nodes with the form of an untyped dynamic object. Aside from that, dependencies between class types that are represented in the symbol table as pointers are converted to edges between them. Furthermore, all information regarding the type of dependencies and their capacity is included in these edges.

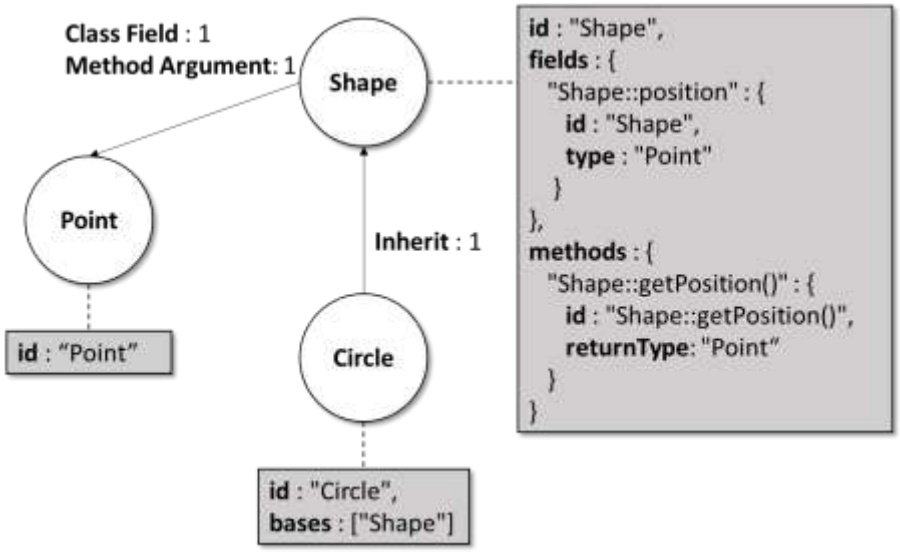

Figure 20 - Graph representation of the example in Figure 12

In Figure 20 the graph representation of the example in Figure 12 is depict. Particularly, in the gray boxes the related data of each node are presented. Moreover, the kind of the dependency as well as its cardinality that an edge declares is noted.

5.3 Dependency Graph Conversion to JSON

The next step is the representation of the subject system graph into JSON format. Specifically, all the information that extracted via Clang and represented as class-dependency graph are stored in a JSON file. To construct it, we traverse over the graph using the graph visitor represented in Figure 15. Additionally, in order to construct and manipulate the JSON value we use the JsonCpp [28] library. The resulting JSON has the following format:

```
{
    "edges" :
    [
        {
            "dependencies" :
            {
                DependencyType : Cardinality,
                ...
            },
            "from" : StartingNodeID,
            "to" : EndingNodeID
        }
    ],
    "nodes" :
    {
        NodeID :
        {
            RelatedData
        },
        ...
    }
}
```

Figure 21 - JSON format for graph representation

As shown in Figure 21, the graph is represented as an object that contains two subelements, an array which stores graph edges and an object that contains graph nodes. Specifically, the array of graph edges stores the beginning and ending nodes, as well as dependency information, for each edge. The nodes object stores graph nodes as key-value pairs, where key is the unique identifier of each node and value is all of its associated data.

Chapter 6

Graph Visualization and Configuration

Our goal is to represent the actual software architecture of a given software system. In the two previous chapters class dependencies extraction and representation as a graph structure are discussed. In this chapter we introduce the implementation of a graph visualizer that enables the class-dependency graph rendering and configuration.

6.1 User Interface

In Figure 22 the user interface of the graph visualizer is depicted. On the left side it is illustrated the graph renderer. On the right side the configuration user interface is shown. In the following sections its implementation details as well as the full range of its functionality are presented.

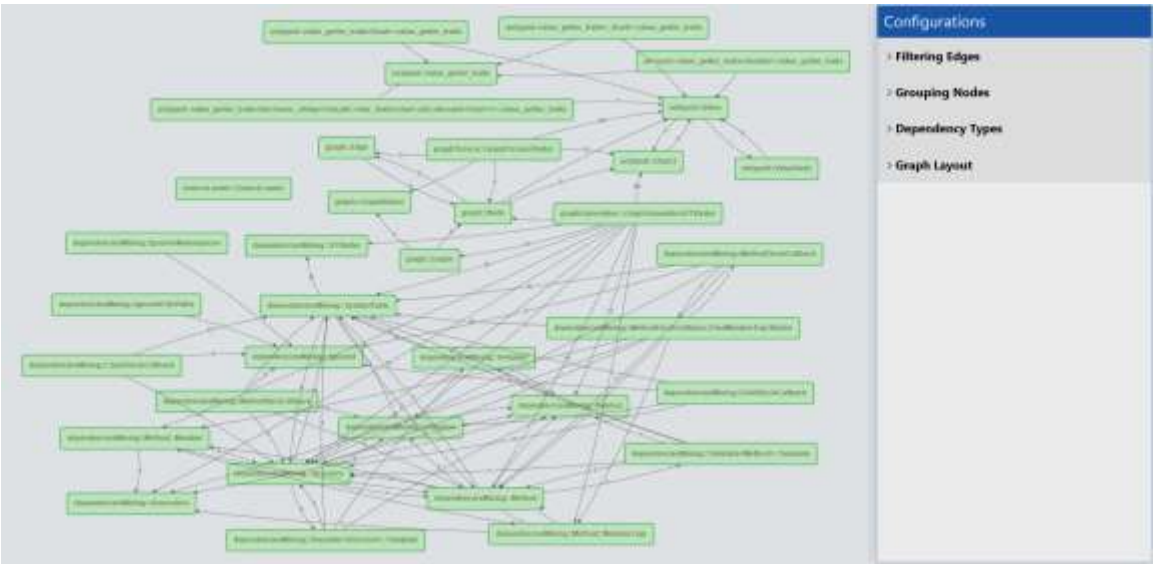

Figure 22 - Graph visualizer user interface

6.2 Implementation

In contrast to the class-data extraction and the representation as a graph structure that developed in C++, the visualization and configuration of the generated graph is implemented in JavaScript, HTML and CSS. In addition, the GoJS [29] library is used for graph visualization, as is the Vue.js [30] framework for creating the configuration panel. Finally, a number of clustering algorithms are used for grouping over graph nodes.

6.2.1 Renderer

As previously mentioned, GoJS library is used for graph rendering. Briefly, GoJS is a JavaScript library to create interactive diagrams and graphs in modern web browsers. Specifically, we use GoJS to construct a directed graph comprised of nodes connected by links. In addition, grouping over them is supported. Finally, force directed and packed graph layouts are used.

The dependency graph generated in the previous chapter and saved in JSON format is passed as an input to the renderer. In order to visualize this graph, its nodes and edges are converted to GoJS format. Additionally, the weight of each edge is calculated and stored. The weight in this initial state is equal to the sum of the edge cardinalities, regardless of dependency type. This could vary because configurations over the computation can be applied to give different importance to each dependency type.

Moreover, a wrapper over GoJS for graph view interaction and configuration is introduced in order to communicate with the graph while hiding all GoJS-related details. For example, addition and removal of graph elements, as well as color and layout configurations, are supported. It is notable that groups are manipulated likewise graph nodes, so we can customize them as well.

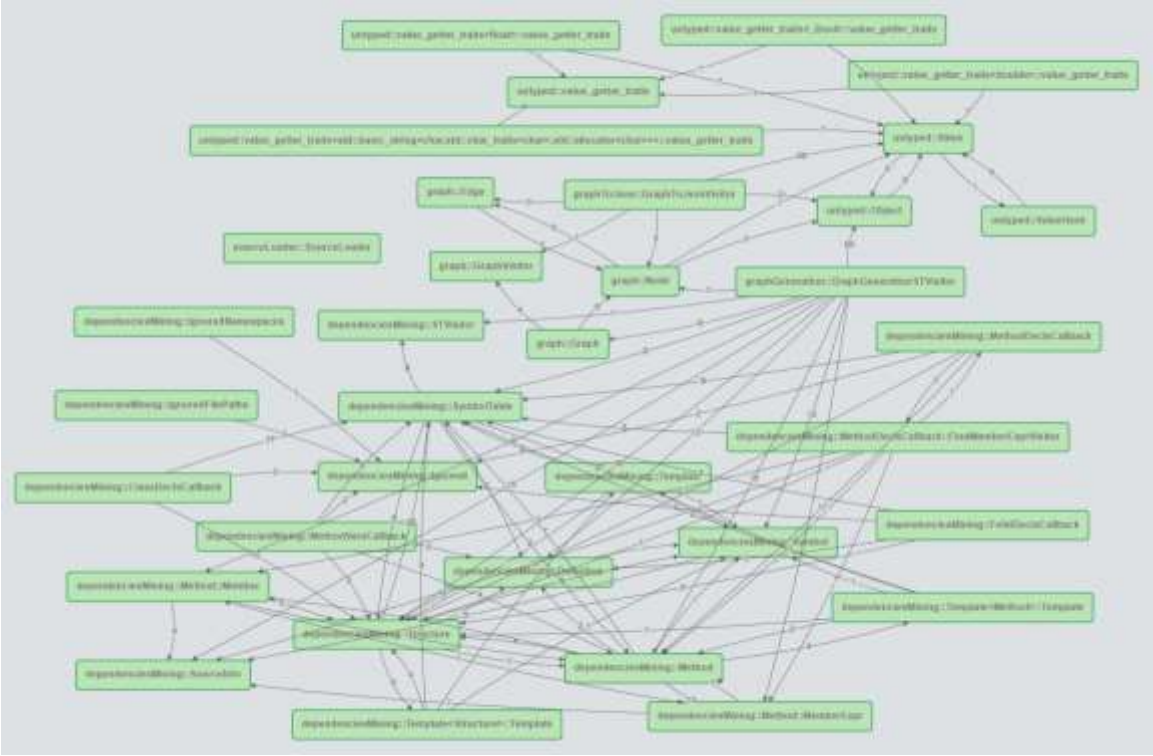

Figure 23 - The graph that is generated by the input JSON file

In Figure 23, an example of the built directed graph is shown. Specifically, nodes are drawn as rectangles using their unique qualified name as a label. Regarding the

connecting edges, they represented as directed links with a label that notes the corresponding weight.

Furthermore, renderer implements all the event handlers that apply the configurations on the graph. Additionally, it installs them on the observer that is responsible to fire them when the corresponding change take place. More details regarding the configuration observer we will see in section 6.2.3. Finally, an API is offered for installing additional configuration.

6.2.2 Configurator

To explore and manipulate the generated graph, we introduce a configuration panel that allows interactive changes. This panel may include a variety of configuration elements such as sliders, checkboxes, and selectors. Additionally, collections of associated elements can be clustered into multi-level groups.

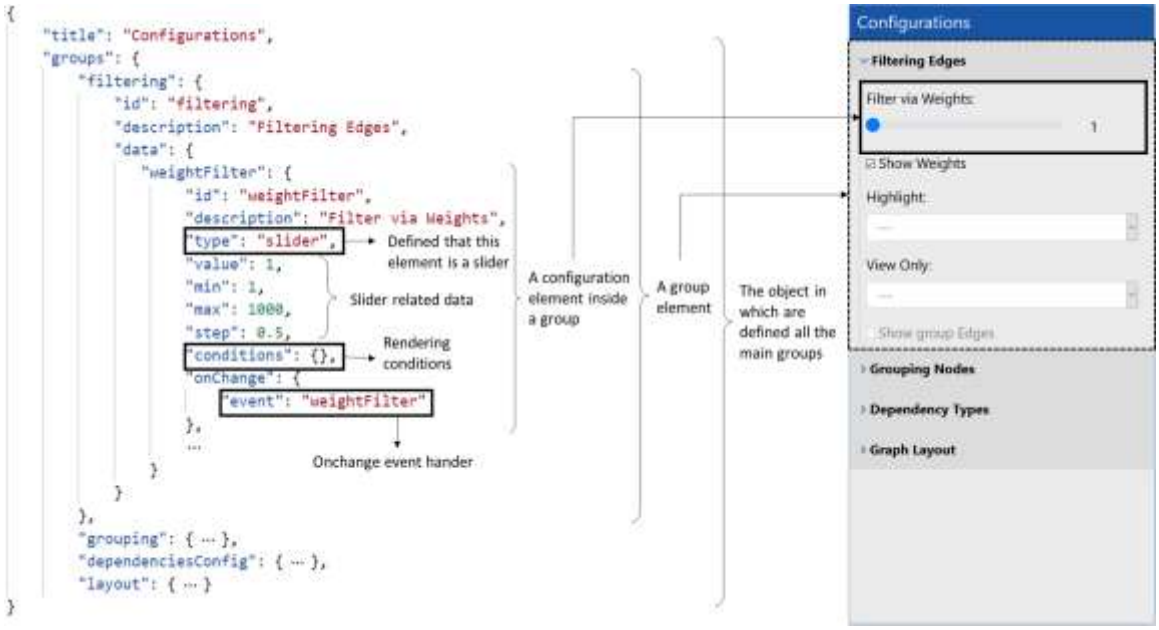

Figure 24 - Example configuration JSON file

In order to construct the panel, a JSON file that describes it in a standard format is required. Based on the structure specified by the JSON file, we construct the panel using Vue.js. In Figure 24 a subset of the configuration JSON file is shown. Specifically, an object that includes a collection with first level groups is defined. Each group contains its configuration elements, which will in turn contain their component type as well as all type related data. Additionally, we can link the generated element with the event handler (onchange and/or onclick) that we implement in order to manipulate this configuration. Finally, we can include some rendering conditions that affect the configuration element's disabled and displayed attributes. It is worth noting that dependencies between components can be defined using these conditions in order to interact as their values change. There are two ways to associate two components. Firstly, using the full path of the element that the component depends on starting from the root object of the JSON (with the keyword *this*). The second way is using the *$currGroup* variable that refers to the group that the dependent component belongs to. An example that represents the dependency definition among components through conditions is illustrated in Figure 25.

```
{
    "groups": {
        "group1": {
            "id": "group1",
            "description": "Group 1",
            "data": {
                "element1": {
                    "id": "element1",
                    "description": "Element 1",
                    "value": false,
                    ...
                },
                "element2": {
                    "id": "element2",
                    "description": "Element 2",
                    ...
                    "conditions": {
                        "disabled" : "this.groups.group1.data.element1.value === true"
                                     or "$currGroup.element1.value === true"
                    }
                }
            }
        }
    }
}
```

Figure 25 - Configure elements interactivity through conditions
In this example, when element1 is set to true, element2 will become disabled until element1's value is changed.

To create this panel, we load the previously analyzed configuration file. JSON Schema [31] is used to validate its contents. The configuration is then passed to the Vue instance as a parsed JSON object in order for the panel to be constructed. In order to represent the configuration elements, we implement a number of Vue template components. Additionally, we use conditions and loops that Vue provides in order to traverse the configuration object fields and create the user interface. Finally, Vue provides two-way data binding on the generated elements so it automatically updates the elements based on the input.

6.2.3 Observer

For the communication of these two components the observer design pattern is used. Through this pattern, objects install an event and get notified when the event occurs. In our system, we implement an observer in which graph renderer installs the event handlers that implements to manipulate all the supported configurations. So, when a configuration occurs, the configurator notifies the observer which fires the corresponding event handler. This communication is depicted in Figure 26.

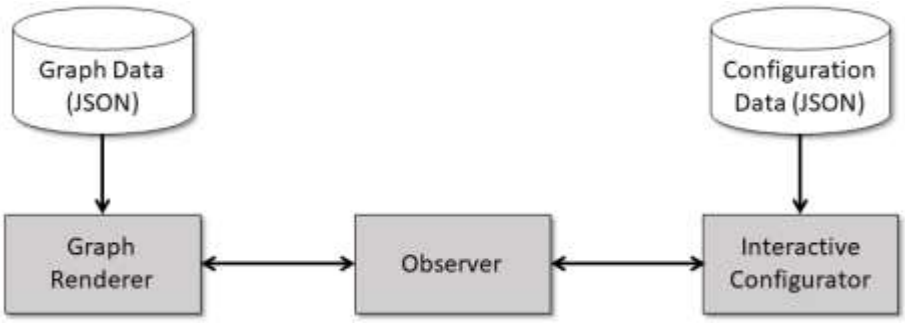

Figure 26 - Communication between graph renderer and configurator through an observer

6.3 Interactive Configuration

Interactivity with the generated graph is significant for assisting the user in a responsive and incremental exploration of it. To accomplish this, we provide an interactive configuration panel as well as direct interaction with graph components in the graph. It is interesting to note that applying a configuration does not cancel the one that is already in existence. All of the available configurations are extensively discussed in the following subsections.

6.3.1 Filtering Edges

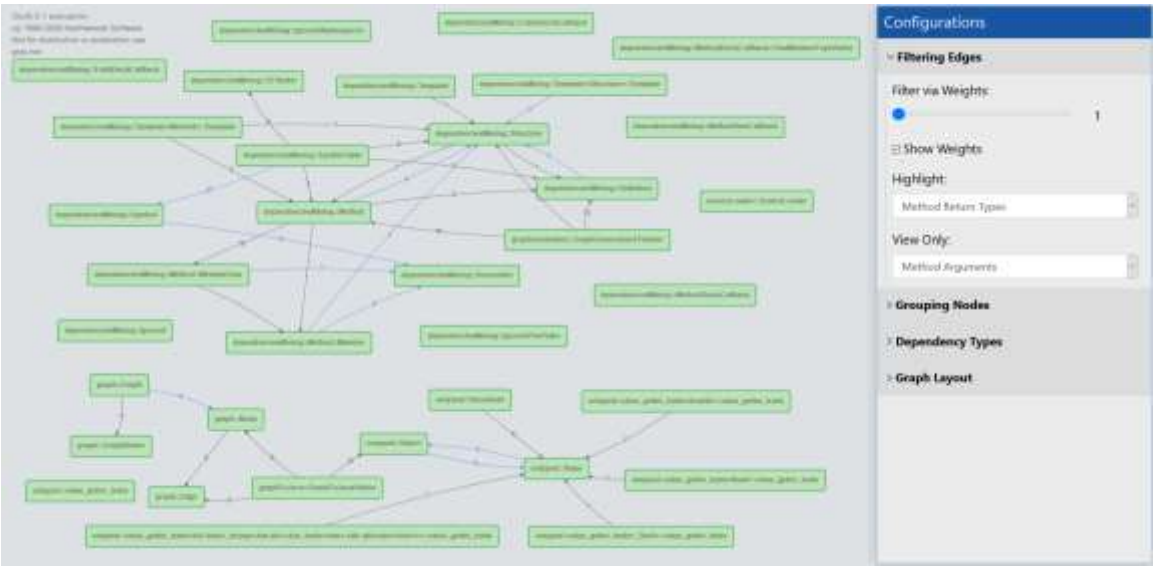

Figure 27 - Apply "View Only" and "Highlight" Filter on graph
In the configuration panel all the filters that describer before is shown.

In this group, configurations concerning graph edges are included. Specifically, we implement a weight filter that hides all edges on the graph with weights less than the selected value. Furthermore, a view only filter is given to display only the edges with a particular dependency type. Similarly, the highlight filter marks the edged with a specific dependency type. It is worth mentioning that we can express the intersection of two dependency types using a compound of view only and a highlight filter. For example,

in Figure 27 the edges that express method argument dependencies and are also method return types are highlighted. Finally, weight label concealment is provided.

6.3.2 Grouping Nodes

The most critical assisted configuration is grouping over graph nodes, which decomposes the system into subsystems. Our approach creates hierarchical graphs in a variety of ways. We may divide them into two groups: those built on the system developer's specified modules and those built on the dependencies between system classes.

Firstly, grouping by the namespaces is provided. Through this grouping we can represent the modules that system develops design, since namespaces are used to provide a means of grouping logically related elements into corresponding modules. Grouping by the source files is also supported. Like namespaces, source files use to contain logically related elements and can thus provide information for developers' built structure. Figure 28 depicts a grouping based on namespaces.

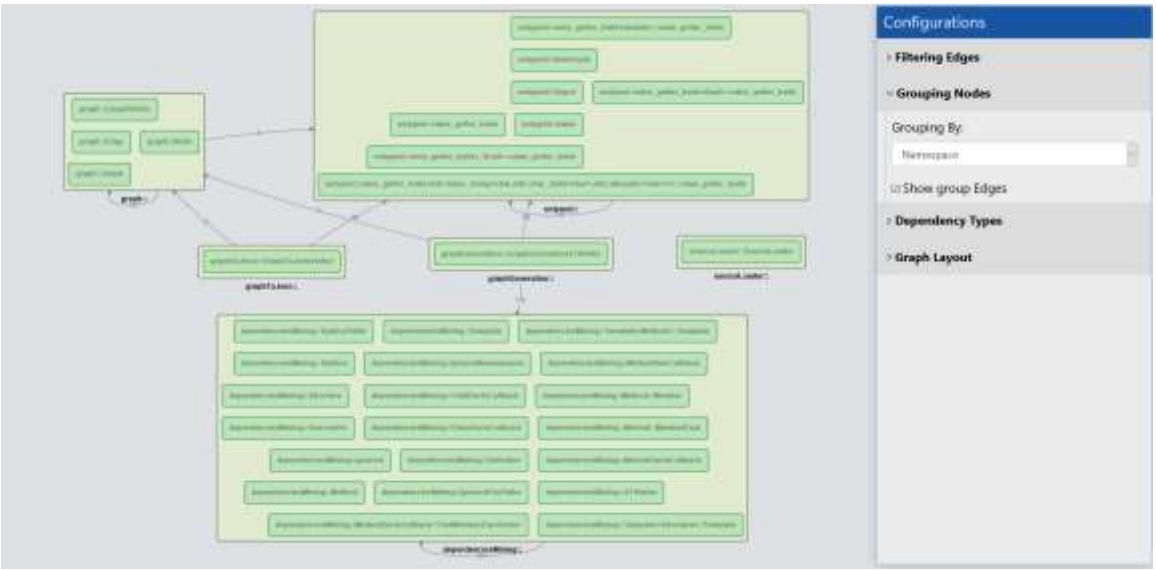

Figure 28 - Grouping by namespace

Furthermore, clustering algorithms are used to group nodes in order to represent system structure based on node dependencies. Section 6.4 goes into greater detail about clustering algorithms. In addition, the most common filename among the nodes in the group is used as the group label.

Finally, we provide group edges between the generated groups. A group edge summarizes the dependencies from the group's owned elements to another group in one edge. Its weight is the sum of the weights of the inner edges. Additionally, self-group edges are generated because they express the dependencies between elements in the same group. Figure 28 shows an example of a dependency representation using group edges.

6.3.3 Dependency Types

```
Number totalWeight(edgeDependencies, configData)
   total = 0
   foreach [type, cardinality] in edgeDependencies do
      if !configData.include(type) then
         total += cardinality
      else if configData.countOnce(type) then
         if configData.isCustom(type) then
            total += configData. getCustomWeight(type)
         else
            total += 1
         end
      else
         total += configData.getCustomWeight(type) * cardinality
      end
   end
   return total
end
```

Figure 29 - Pseudocode for computing edge total weight

Not all class dependencies are equally important. Furthermore, there is no objective way to measure the significance of a dependence type. As a consequence, we give the option of configuring graph edge weights based on dependency type. Specifically, the ability to count the cardinality of a particular dependency only once is supported, as is the ability to determine the value of a dependency by increasing or decreasing the corresponding

weight. The pseudocode in Figure 29 illustrates the steps that we follow for an edge total weight computation applying the potential configurations.

Figure 30 depicts the “count once” and “customize weight” configuration of dependencies that express class field relationships between classes. Because changes are not immediately reflected on the graph, it is necessary to press the “Apply” button to set them. If no customization is applied, the dependency's default value is one. In addition, when the “count once” configuration is used, the customized weight value is applied. It is worth noting that by setting the weight of a dependency type to zero, it is possible to completely ignore it from the graph. Finally, it is important to mention that the configuration that will be applied on edge weights will have an effect on node clustering computation.

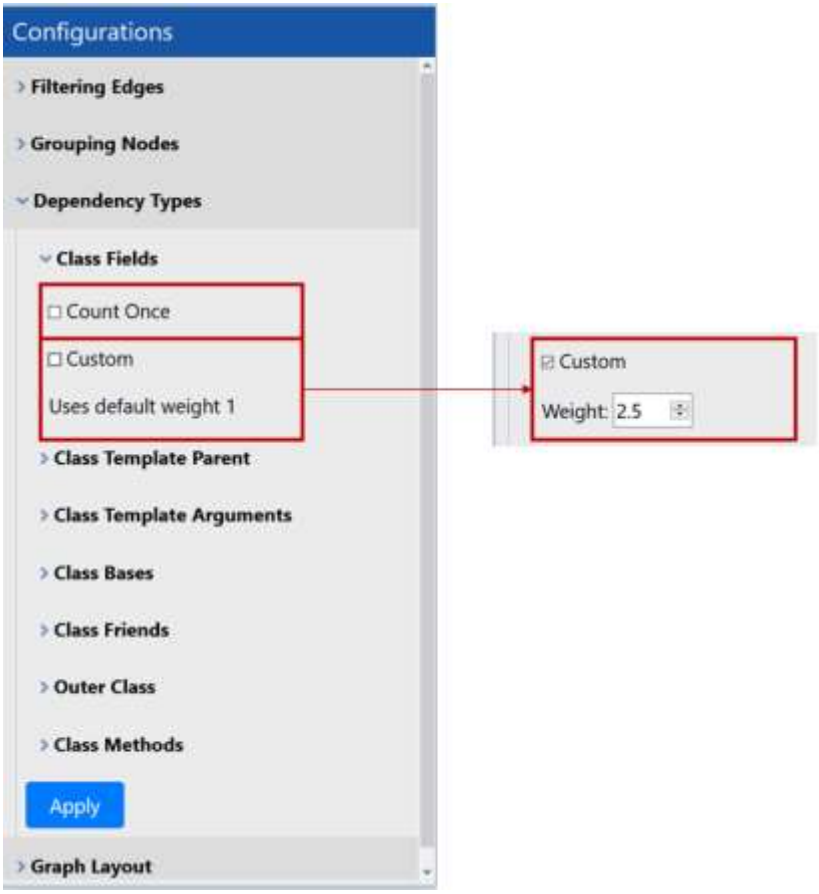

Figure 30 - Count once and custom weight configurations
On the right the user interface for weight customization is presented.

6.3.4 Graph Layout

Two layouts are used for node positioning inside and outside of graph groups: force directed and packed. In short, force directed layout treats the graph as if it were a system of physical bodies with forces acting on them and between them. This layout creates a well-balanced graph, where nodes positioning depends on their interrelationships. On the other hand, packed layout attempts to pack nodes as close together as possible without overlap. This layout is used for a simpler rendering, but it provides no details about node dependencies.

By default, we define the outer graph layout to force directed and inside groups to packed. The ability to change the outer layout to packed is provided.

6.3.5 Direct Configurations

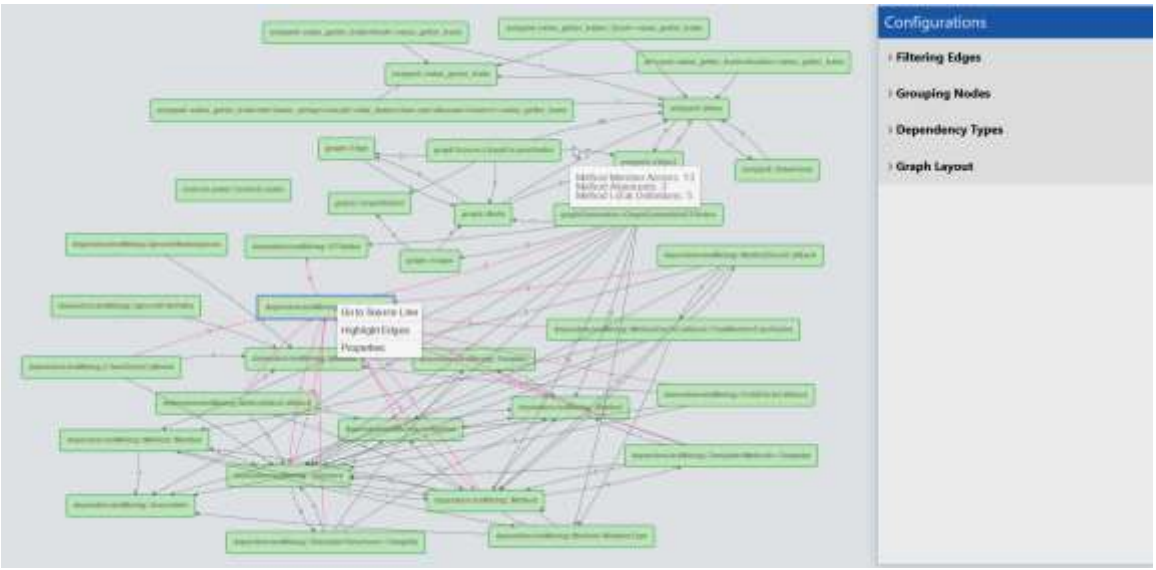

Figure 31 - Direct graph interaction

Finally, we can interact with graph components directly in the graph. Firstly, we provide a context menu for a specific note exploration. Specifically, we let the direct transition to

the class definition in the source code as well as the highlighting of incoming and outcoming edges of a node. Additionally, when the mouse hovers on a graph element, related information for this element is presented. Some of these functionalities are illustrated in Figure 31.

6.4 Clustering Algorithms

Initially, the generated graph is flat containing only atomic nodes. Grouping nodes into super-nodes allows us to impose structure on a flat graph. As a result, we choose three network clustering algorithms in order to build clusters with tightly interconnected nodes into modules and clusters with the optimal number of nested modules (where supported). The ability to manage weighted graphs is a requirement for the algorithm, while it is desirable to support directed graphs and multi-level clusters. Finally, there is the option to incorporate new clustering algorithms.

6.4.1 Louvain

The Louvain method [32] for community detection is a method to extract the community structure of large networks. To succeed that it maximizes the modularity score for each community, where the modularity measures the relative density of edges inside communities with respect to edges outside communities. This algorithm is divided into two phases: modularity optimization and community aggregation. First, it looks for "small" communities by optimizing modularity in a local way. Second, it aggregates nodes of the same community and builds a new network whose nodes are the communities. These steps are repeated iteratively until a maximum of modularity is attained.

Louvain is a hierarchical clustering algorithm and supports weighted as well as unweighted graphs. Additionally, since it recursively merges communities into a single

node and executes the modularity clustering on the condensed graphs, it provides nesting clustering. Finally, no predefined information about the communities is needed.

6.4.2 Infomap

Infomap optimizes the map equation [33], which exploits the information-theoretic duality between finding community structure in networks and minimizing the description length of a random walker's movements on a network. Its methodology is similar to that of the Louvain algorithm, but they differ in the optimizing function.

It is a hierarchical algorithm that can handle unweighted and weighted edges, as well as undirected and directed edges. It also offers simple clusters with interconnected nodes on a single level, as well as multi-level clustering solutions. Finally, no prior knowledge of the communities is required.

6.4.3 Layered Label Propagation

The Label Propagation algorithm [34] is a fast algorithm for finding communities in a graph. It detects these communities using network structure alone as its guide. Particularly, it works by propagating labels throughout the network and forming communities based on this process of label propagation. We used the Layered Label Propagation [35], an iterative algorithm that produces a sequence of node orderings. At each iteration, the Layered Propagation algorithm is run with a suitable value of parameter γ, and the resulting labeling is then turned into an ordering of the graph that keeps nodes with the same label close to one another. Values of γ close to 0 highlights a coarse structure with few, big and sparse clusters, while, as γ grows, the clusters are small and dense, unveiling a fine-grained structure.

This algorithm works for both unweighted and weighted graphs, as well as undirected and directed graphs. However, its main drawback is that it provides an aggregate of many solutions rather than a deterministic result.

Chapter 7

Case Studies

To test the proposed architecture mining approach of our work and assess its expressive power, we applied several case studies. In this chapter, we describe them by separating them into three different categories. Each of these categories achieves one of our tool objectives introduced in section 1.3. In the first section, we examine and discuss the outcomes of the system itself. In the second section, we outline the case for evaluating a complete programming language for which some structure information is provided. Finally, in the last section, we describe a Super Mario game implementation with an undefined architecture.

7.1 Architecture Miner

Our first case study is the system itself, in order to visualize and judge its structure as a software system as well as its effectiveness as an architecture mining tool. In addition, we would like to assess potential surprises that may occur to the structure of the system. Specifically, we execute it over the dependency graph generator, the part of our project which is implemented in C++. In order to isolate the output to our own system, we ignore

all the elements that pertain to Clang project as well as those that belong to JsonCpp library which is used for graph translation to JSON format.

7.1.1 Expected Output

We group our system elements based on their logical relation in six namespaces as they are illustrated in Figure 32. Briefly, the *dependencyMining* namespace contains all of the functionality that is responsible for data extraction from sources and symbol table composition. In addition, the *graph* namespace defines the generic graph. The *graphGeneration* and g*raphToJson* namespaces determine the symbol table to graph and graph to JSON translators respectively. Finally, the *untyped* namespace for the untyped object representation, and the *sourceLoader* namespace for sources loading, are defined. Thus, these namespace definitions describe the structure that we assume our system has. Apart from namespace grouping, our system is divided into two logical components: class data extraction, which generates the program symbol table, and graph composition based on these extracted data, both of which have been extensively discussed in chapters 4 and 5. These components are not directly defined in source code but we expect that they will arise from class interrelationships.

7.1.2 Results and Discussion

From the evaluation of our system, interesting results arise from the one-level as well as multi-level Infomap clustering. Figure 32 illustrates the one-level clustering result. We observe that the output is really close to the structure that we define through the namespaces during the implementation. Specifically, the only difference is that the *graphToJson* group is merged with the *untyped* namespace group. This merge is reasonable, since the *graphToJson* group contains a node that has many inner dependencies to the *untyped::Value* and *untyped::Object* nodes. It should be noted that no configuration over dependency type weights is used.

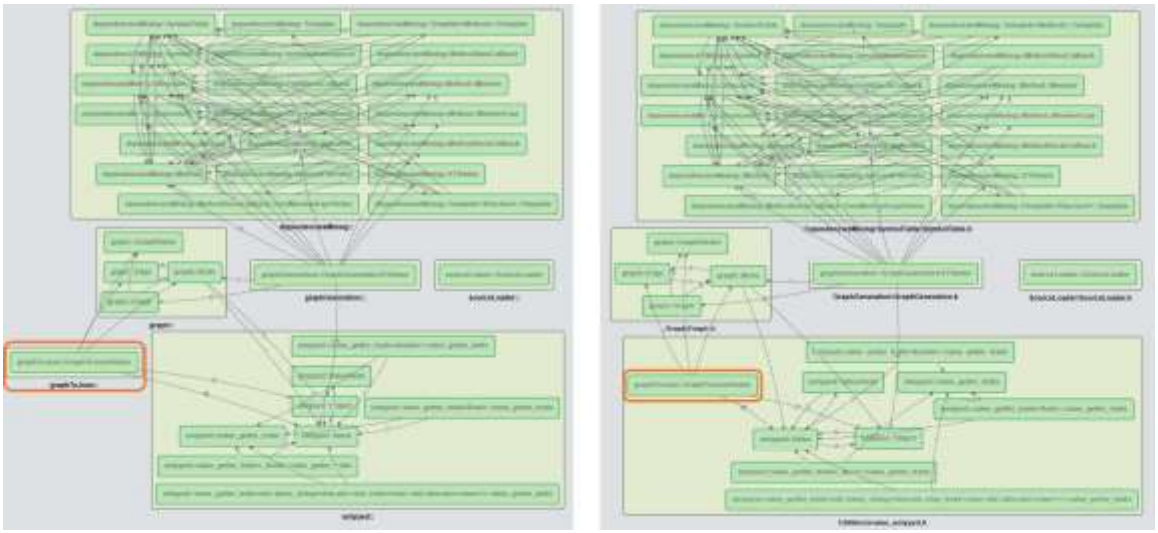

Figure 32 - Namespaces grouping (left) and one-level Infomap clustering (right) of Architecture Miner system

In Figure 33 the outcome of the Infomap algorithm for nested clustering results is depicted. As we can see, the graph is divided into three basic groups. Since *sourceLoader* has no interrelationship with the other nodes it will be ignored for the rest of the discussion. Studying the other two groups, we observe that they represent the class data extraction (left) and dependency graph composition (right) logical components. We can observe that these two components are independent, with no connecting edges between their inner elements. The link between them is the *graphGeneration::graphGenerationSTVisitor* node (1). This result is valid since that class is also the logical interconnection between them because the extracted data is used to construct the dependency graph. We reasonably expected this node to be part of the graph generation cluster, but it seems that data traversing dependencies have attracted it to the data extraction group. Furthermore, the inner group structure is relative to assumption because it adheres to the above-mentioned namespace categorization. It should be noted that no configuration over dependency type weights is used.

It is remarkable that the Infomap algorithm divides nodes with only inner interconnections and incoming edges (2, 3, 4, 5) and those with only outer dependencies

(1, 6) into separate clusters, treating them as independent components. As a result, it maintains the highest relative density of edges within the clusters.

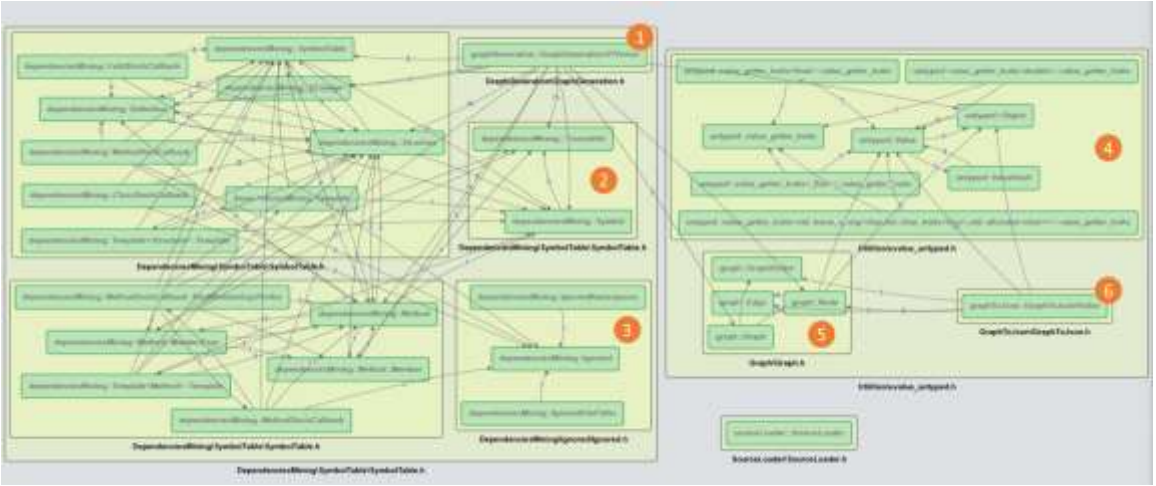

Figure 33 - Multi-level Infomap clustering output of Architecture Miner system

Figure 34 illustrates the outcome of the Louvain algorithm for nested clustering results. At first glance, the graph appears to be divided into four major classes. When we examine them more closely, we can see that the cluster on the right represents the dependency graph composition component as it arises in the Infomap output, with minor differences in the structure of its inner groups. We also note that the logical component of the class data extraction is divided into two groups, with the methods related nodes being placed in a separate group.

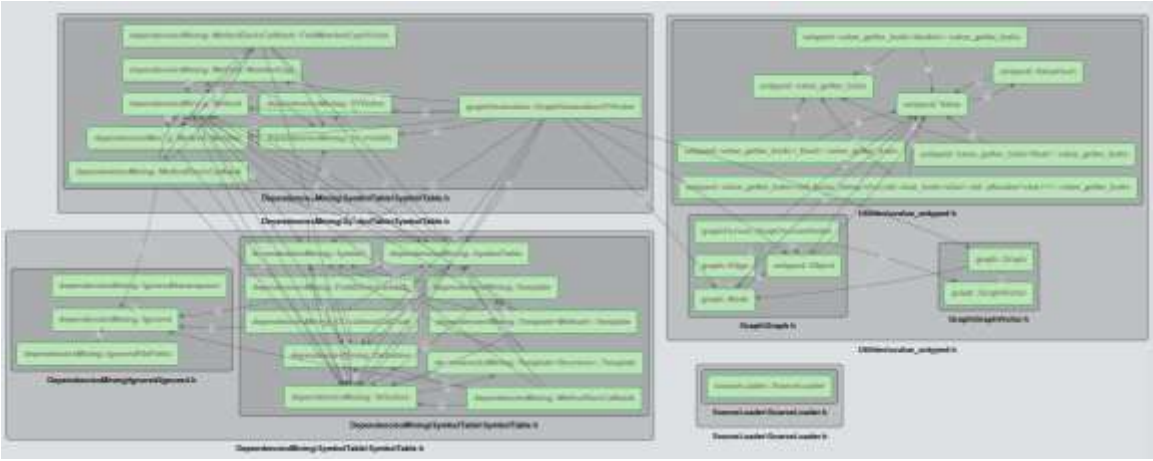

Figure 34 - Multi-level Louvain clustering output of Architecture Miner system

Finally, the Label Propagation algorithm is applied. However, since there is no nested clustering and the result is non-deterministic, it is difficult to handle.

7.2 Delta Language

In this section a complex language system will be studied. Specifically, we reverse engineer the Delta language [36], an untyped object-based, dynamically-typed language that runs by a virtual machine. Additionally, some documentation is provided.

7.2.1 Expected Output

Regarding the Delta language is implementation in C++. Overall, the implementation includes: compiler, virtual machine, debugger back-end and front-end and standard runtime library. Furthermore, extra libraries, like JSON and XML, as well as an IDE are implemented, but they are excluded from this analysis. Thus, we are waiting to deduce these components from the graph representation of system.

7.2.2 Results and Discussion

At first glance, in Figure 35 we observe that the graph is divided into a main, strongly connected subgraph that contains the majority of system nodes, and many independent smaller subgraphs or individual nodes that form their own communities. So, we consider the main subgraph to be the core implementation of the language, while the other components are library elements that make up the standard library of the language.

Clustering the graph in multi-levels using the Infomap algorithm, the core of the language is divided into three main groups. Looking closely, we observe that two of them includes compiler related nodes while the other includes nodes that implements the runtime

environment. Regarding the compiler elements, they are separated based on the compiler phase they serve. Particularly, in the first cluster lexical and syntax analysis related nodes are included. In the second group, the symbol table and code optimizer are contained. Finally, in the third cluster the virtual machine as well as the debugger front-end and back-end are located.

Lastly, both the graph representation and the clustering results provide useful information about this language, mainly to the system own developer, who can use it to quickly inspect its implementation. For a third-party developer, the outcome of such a large and complicated system may be difficult to read. Nevertheless, they can acquire some basic knowledge regarding the system structure.

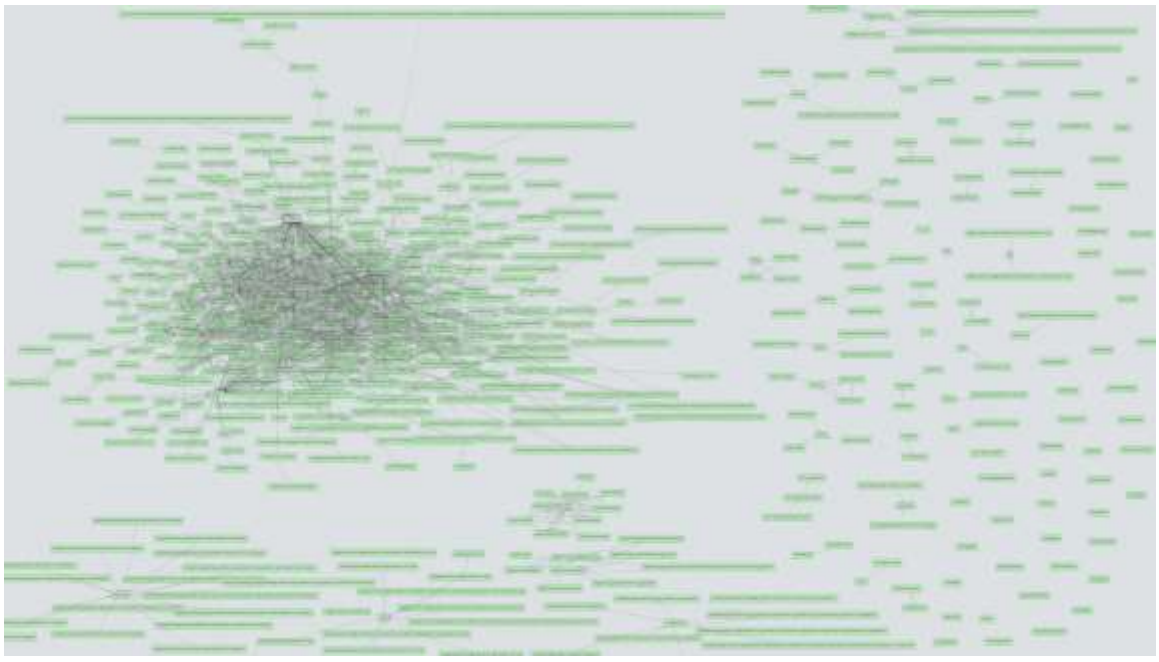

Figure 35 - Delta language dependencies graph

7.3 Super Mario

Our last case study is a Super Mario game implementation based on the homonym video game. This system is a 2D, tile-based video game but no further information regarding its

development or architecture is available. Our goal is to reverse engineer the source code in order to extract knowledge for the system structure.

7.3.1 Expected Output

Since it is a game, even if no knowledge about the system structure is given, we expect to recognize some basic components. Specifically, we are waiting to identify the game engine as well as the game implementation using this engine.

7.3.2 Results and Discussion

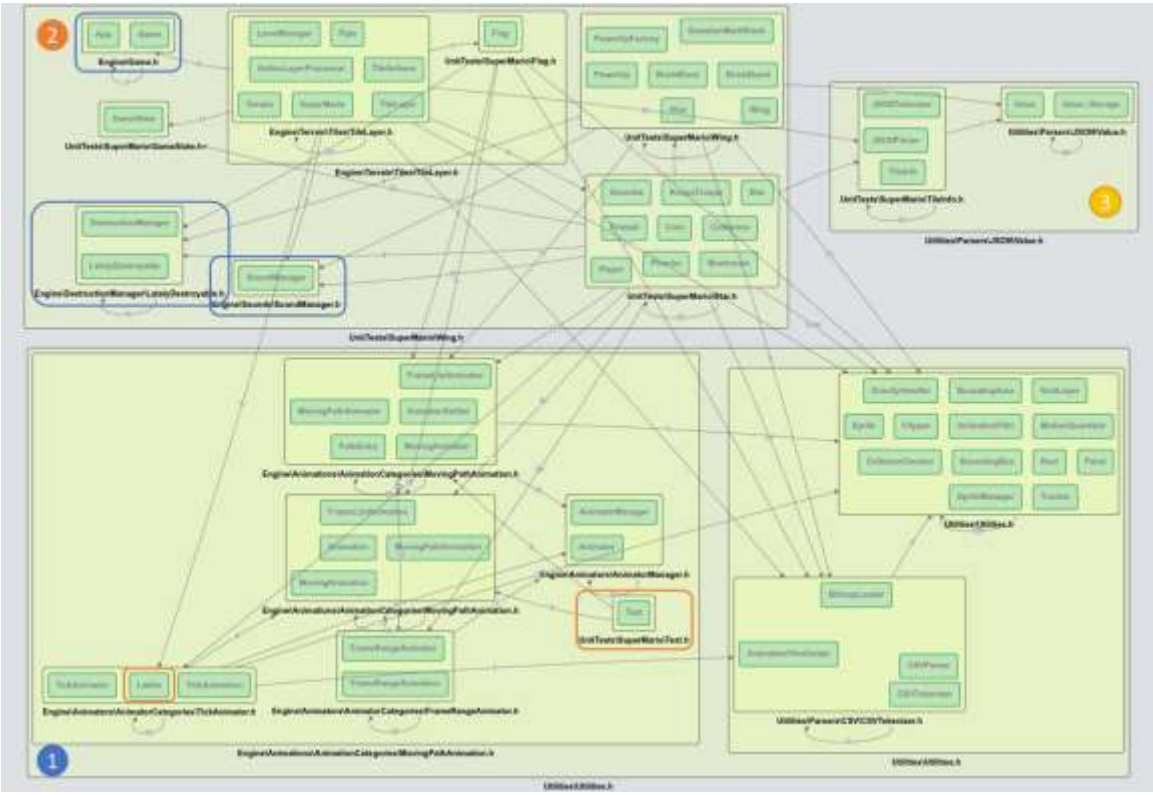

Figure 36 - Reverse engineering of Super Mario game using Infomap algorithm
The nodes that will logically constitute game engine (group 1) elements are annotated in blue. The nodes that will logically be included in the game implementation basic group (group 2) are annotated in orange.

In Figure 36 the resulted clustering of the Infomap algorithm is illustrated. At first glance, the source code is divided into three basic groups. Looking at them observantly, we can recognize the two main parts of a game system, the game engine (group 2) and the Super Mario game (group 1) which is implemented using this engine. In addition, an extra basic group (group 3) is presented, including some utilities related to game implementation. Regarding the game engine, it is separated into two subgroups. Studying the included nodes, we observe that animation and animator elements appertain to the left group (1-left) while the right group (1-right) consists of screen related elements as well as utilities for animation loading. Concerning the game implementation group, it includes the elements that compose the Super Mario game. In this group, most of the game objects as well as the game state holder are located.

While the basic groups seem to have great expressiveness, incorrect node assignments are observed. Specifically, the *SoundManager* and the *DestructionManager* are included in game group, even though constitute game engine components. This behavior is reasonable since these elements have no dependency with the other game engine components, thus, they attracted by the nodes that depend on them. Moreover, some game objects are wrongly included in game engine cluster. Both of them are acceptable errors because of the dependency-based clustering.

Another piece of information that we acquire from the graph, is the use of the JSON and CSV parsers. We can see, that the *JSONParser* is grouped with the *TileInfo* node, and that gives us the hint that the programmers used a JSON parser in order to parse information about the tiles. Also, the *CSVParser* node is grouped with *BitmapLoader* and *AnimationFilmHolder* which means that the Comma Separated Values format is used for animations and bitmap loading.

Figure 37 illustrates the outcome of the Louvain algorithm for one-level clustering results. As we can see, cluster 2 includes the animation and animator classes. Additionally, most of the game objects are grouped in cluster 1. However, the other three clusters do not

appear to represent any specific system unit. Furthermore, we observe that many game objects are strewn around in all of the clusters. As compared to Infomap, we can infer that Infomap produces better results.

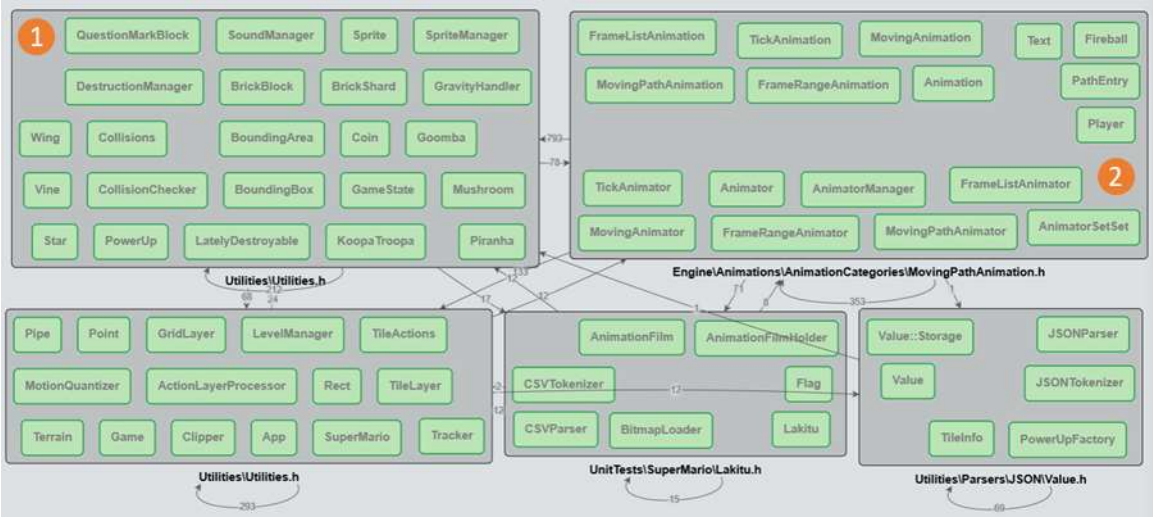

Figure 37 - One-level clustering of Super Mario game using Louvain algorithm

The Label Propagation algorithm is used too. Even so, it is difficult to manage the result since it is non-deterministic.

7.4 Clustering Algorithms Discussion

By the three clustering algorithms that we use, Infomap seems to have better clustering results compared to the other two. Specifically, the application of Louvain algorithms on the cases analyzed above has sufficiently results, but the fact that it does not value the direction of edges has effects on the output, since it does not recognize the difference between the "uses" and "is used" relationship of nodes. On the other hand, the lack of nested clustering as well as the non-deterministic output of Label Propagation algorithm, makes its outcome difficult to be managed. Thus, since Infomap provides both nesting clustering and values the direction of the edges, it is considered being more appropriate. This can be observed in the examples that studied in this chapter.

Chapter 8

Conclusion and Future Work

In large-scale systems, software architecture is fundamental for their maintenance and understanding. However, as software systems evolve over time, choices may be made that deviate from the originally intended architecture. Furthermore, the lack of documentation in many systems makes developer work more difficult. Several tools, such as source code structure visualizers, architecture miners, and quality checkers, have been proposed to aid developers in getting a better understanding of the system, some with excellent results and others with less so.

In this thesis we introduce an architecture mining tool with the aim of reconstructing a given system's architecture. In particular, we statically analyze the given source code using the Clang compiler front-end. The result of this analysis is a symbol table that includes data regarding program classes that extracted directly from the source code. Then, based on this data, it generates a dependency graph that represent all the relationships between program classes. Moreover, we implement a graph visualizer that enables dependency graph rendering and interactive configuration. In order to deduce abstractions that correspond to system components, we also use Louvain, Infomap and Layered Label Propagation community detection algorithms for graph clustering.

We have also carried out some case studies to test and validate the architecture mining approach of our work. Each of them looks at a different aspect of our tool. Firstly, we evaluate and discuss the results of our system in order to visualize and judge its structure as a software system as well as its effectiveness as an architecture mining tool. In the other two cases, we assess non-owned systems with little or no additional knowledge attempting to understand their core components. In comparison to the other two, Infomap appears to have better clustering results from the applied clustering algorithms, providing more meaningful abstractions.

In conclusion, it will be very interesting to expand our system to identify both good practices, such as design patterns, and bad practices, such as circular dependencies. It will also be useful to make suggestions for improvements to the detected bad practices. Furthermore, it would be beneficial to provide metrics for evaluating the quality of software. For instance, the average number of methods and fields can be estimated. Finally, the visualization of the model has to be improved to provide a clearer overview, with the option of hiding details from modules at multiple levels of abstraction, possibly with the creation of a custom visualization library.

Bibliography

[1] Garlan, D. (2000, May). Software architecture: a roadmap. In Proceedings of the Conference on the Future of Software Engineering (pp. 91-101).

[2] P. Clements, D. Garlan, R. Little, R. Nord, and J. Stafford, “Documenting software architectures: viewsand beyond,” in 25th International Conference on Software Engineering, 2003. Proceedings. IEEE, 2003, pp. 740–741.

[3] Hochstein, L., & Lindvall, M. (2005). Combating architectural degeneration: a survey. Information and Software Technology, *47*(10), 643-656.

[4] Ducasse, S., & Pollet, D. (2009). Software architecture reconstruction: A process-oriented taxonomy. IEEE Transactions on Software Engineering, 35(4), 573-591.

[5] Perry, D. E., & Wolf, A. L. (1992). Foundations for the study of software architecture. ACM SIGSOFT Software engineering notes, 17(4), 40-52.

[6] Chikofsky, E. J., & Cross, J. H. (1990). Reverse engineering and design recovery: A taxonomy. IEEE software, 7(1), 13-17.

[7] Sourcetrail – The open-source cross-platform source explorer. Version 20.2.43. Released on 06/2020. Development Team: The Sourcetrail Development Team. Official Website: https://www.sourcetrail.com/ Accessed online: 02/2021.

[8] Understand – Take Control of Your Code. Version 5.1. Released on 02/2019. Development Team: Scientific Toolworks, Inc. Official Website: https://www.scitools.com/ Accessed online: 02/2021.

[9] Source Insight Programming Editor and Code Browser. Version 4. Released on 11/2020. Developers Team: Source Dynamics. Official Wepsite: https://www.sourceinsight.com/ Accessed online: 02/2021.

[10] Class Visualizer – interactive class diagrams generator from Java bytecode and Kotlin bytecode. Version 11.0.0. Released on 02/2019. Authored by: Jonatan Kaźmierczak. Official Website: https://class-visualizer.net/ Accessed online: 02/2021.

[11] Imagix 4D – Reverse Engineer and Analyze Your Source Code. Version 10.1.0. Released on 10/2020. Development team: Imagix Corporation. Official website: https://www.imagix.com/ Accessed online: 02/2021.

[12] Kienle, H. M., & Müller, H. A. (2010). Rigi—An environment for software reverse engineering, exploration, visualization, and redocumentation. Science of Computer Programming, 75(4), 247-263.

[13] Kazman, R., O'Brien, L., & Verhoef, C. (2003). Architecture reconstruction guidelines third edition. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

[14] Sonargraph Product Family – Sonargraph-Architect. Version 6.0. Released on 11/2020. Development team: hello2morrow. Official website: https://www.hello2morrow.com/products/sonargraph/architect9 Accessed online: 02/2021.

[15] Sotograph Product Family. Version 5.0. Released on 12/2018. Development team: hello2morrow. Official website: http://www.hello2morrow.com/products/sotograph Accessed online: 02/2021.

[16] Enterprise Architect – Visual Modeling Platform. Version 15.2. Released on 08/2000. Development Team: Sparx Systems. Official website: http://www.sparxsystems.com/products/ea/index.html Accessed online: 02/2021.

[17] Visual Paradigm – Suite of design, analysis and management tools that drive your IT project development and digital transformation. Version 16.1. Released on

12/2019. Development Team: Visual Paradigm International Ltd. Official website: https://www.visual-paradigm.com/ Accessed online: 02/2021.

[18] UModel 2021 – UML tool for software modeling and application development. Released on 10/2020. Development Team: ALtova. Official website: http://www.altova.com/umodel.html Accessed online: 02/2021.

[19] Systems Modeling Language SysML by OMG. Official website: http://www.omgsysml.org/ Accessed online: 02/2021.

[20] Business Process Model and Notation BPMN by OMG. Official website: http://www.bpmn.org/ Accessed online: 02/2021.

[21] JSON Compilation Database Format Specification – Clang 12 documentation. Development Team: LLVM Developer Group. Official Website: https://clang.llvm.org/docs/JSONCompilationDatabase.html Accessed online: 02/2021.

[22] Clang – A C Language Family Frontend for LLVM. Released on 09/2007. Development Team: LLVM Developer Group. Official Website: https://clang.llvm.org/ Accessed online: 02/2021.

[23] The LLVM Compiler Infrastructure. Released on 2003. Development Team: LLVM Developer Group. Official Website: https://llvm.org/ Accessed online: 02/2021.

[24] Matching the Clang AST – Clang 12 documentation. Development Team: LLVM Developer Group. Official Website: https://clang.llvm.org/docs/LibASTMatchers.html Accessed online: 02/2021.

[25] How to write RecursiveASTVisitor – Clang 12 documentation. Development Team: LLVM Developer Group. Official Website: https://clang.llvm.org/docs/RAVFrontendAction.html Accessed online: 02/2021.

[26] LibTooling – Clang 12 documentation. Development Team: LLVM Developer Group. Official Website: https://clang.llvm.org/docs/LibTooling.html Accessed online: 02/2021.

[27] AST Matchers Reference. Development Team: LLVM Developer Group. Official Website: https://clang.llvm.org/docs/LibASTMatchersReference.html Accessed online: 02/2021.

[28] JsonCpp – A C++ library for interacting with JSON. Version 1.9.4. Released on 09/2020. Github Repository: https://github.com/open-source-parsers/jsoncpp Accessed online: 02/2021.

[29] GoJS – Interactive JavaScript Diagrams for the Web. Version 2.1. Released on 11/2019. Development Team: Northwoods Software. Official Website: https://gojs.net/latest/index.html Accessed online: 02/2021.

[30] Vue.js – A progressive JavaScript framework for building user interfaces. Released on 02/2014. Development Team: Vue.js Developer Team. Official Website: https://vuejs.org/ Accessed online: 02/2021.

[31] JSON Schema – A vocabulary that allows you to annotate and validate JSON documents. Released on 12/2009. Development Team: json-schema-org. Official Website: https://json-schema.org/ Accessed online: 02/2021.

[32] Blondel, V. D., Guillaume, J.-L., Lambiotte, R. & Lefebvre, E. (2008). Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment, 2008, P10008.

[33] Rosvall, M., Axelsson, D. & Bergstrom, C. T. (2009). The map equation (cite arxiv:0906.1405Comment: 9 pages and 3 figures, corrected typos. For associated Flash application, see http://www.tp.umu.se/~rosvall/livemod/mapequation/).

[34] Raghavan, U., Albert, R., & Kumara, S. (2007). Near linear time algorithm to detect community structures in large-scale networks. Physical review. E, Statistical, nonlinear, and soft matter physics, 76 3 Pt 2, 036106.

[35] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In Proceedings of the 20th international conference on World wide web (WWW '11). Association for Computing Machinery, New York, NY, USA, 587–596. DOI:https://doi.org/10.1145/1963405.196348.

[36] The Delta programming language – Dynamic embeddable language for extending applications. Released on 2004. Authored by: Anthony Savidis. Official Website: https://projects.ics.forth.gr/hci/files/plang/Delta/Delta.html Accessed online: 02/2021.