Facilitating comprehension of Swift programs
Abstract

Program comprehension is the process of gaining knowledge about software system by extracting it from its source code or observing its behavior at runtime. Often, when documentation is unavailable or missing, this is the only reliable source of knowledge about the system, and the fact that up to 50% of total maintenance effort is spent understanding the system makes it even more important. The source code of large software systems contains thousands, sometimes millions of lines of code, motivating the need for automation, which can be achieved with the help of program comprehension tools. This makes comprehension tools an essential factor in the adoption of new programming languages. This work proposes a way to fill this gap in the ecosystem of Swift, a new, innovative programming language aiming to cover a wide range of applications while being safe, expressive, and performant.

The proposed solution is to bridge the gap between Swift and VizzAnalyzer, a program analysis framework featuring a range of analyses and visualizations, as well as modular architecture which makes adding new analyses and visualizations easier. The idea is to define a formal model for representing Swift programs and mapping it to the common program model used by VizzAnalyzer as the basis for analyses and visualizations. In addition to that, this paper discusses the differences between Swift and programming languages which are already supported by VizzAnalyzer, as well as practical aspects of extracting the models of Swift programs from their source code.

**Keywords:** software comprehension, program analysis, program model, VizzAnalyzer, Swift programming language
## Contents

List of Figures .......................................................................................................................... 5

List of Abbreviations .................................................................................................................. 6

1 Introduction ............................................................................................................................... 7
   1.1 Problem Background .......................................................................................................... 7
   1.2 Problem Motivation .......................................................................................................... 8
   1.3 Problem Formulation ....................................................................................................... 9
   1.4 Contributions .................................................................................................................. 10
   1.5 Target Groups ................................................................................................................. 10
   1.6 Report Structure ............................................................................................................. 10

2 Background ............................................................................................................................ 11
   2.1 Related Work .................................................................................................................. 11
   2.2 Theoretical Background ................................................................................................. 12
      2.2.1 Software Maintenance .............................................................................................. 12
      2.2.2 Program Comprehension ......................................................................................... 13
      2.2.3 Program Comprehension Tools .............................................................................. 14
      2.2.4 Architecture of Program Comprehension Tools ..................................................... 15
      2.2.5 Program Metamodels .............................................................................................. 16
   2.3 VizzAnalyzer Architecture .............................................................................................. 17
      2.3.1 Low-level Analysis Engine (LLA) ............................................................................ 18
      2.3.2 High-level Analysis Engine (HLA) .......................................................................... 19
      2.3.3 Visualization Engine (VE) ....................................................................................... 19
      2.3.4 Definition of Common Meta Model .......................................................................... 20

3 Overview of Swift Features .................................................................................................... 22
   3.1 Type System .................................................................................................................... 22
   3.2 Optional Types ................................................................................................................ 25
   3.3 Control Flow ................................................................................................................... 25
   3.4 Code Organization and Access Control .......................................................................... 26
   3.5 Code in Global Scope ..................................................................................................... 26

4 Approach .................................................................................................................................. 27
   4.1 Extending VizzAnalyzer ................................................................................................. 27
   4.2 Analysis of Compatibility of Swift and CMM ................................................................. 27
      4.2.1 Type System ............................................................................................................. 27
      4.2.2 Control flow ............................................................................................................. 28
      4.2.3 Code organization and access control .................................................................... 28
   4.3 Mapping of Swift FSMM to CMM ................................................................................... 29
      4.3.1 FSMM for Swift ....................................................................................................... 29
      4.3.2 Mapping of Swift FSMM to CMM ......................................................................... 30
4.4 Overview of the Proposed Solution..................................................31

5 Implementation ..................................................................................32
  5.1 Extracting Swift FSMM Instances from Source Code ....................32
    5.1.1 Parsing the AST Dump Produced by Swift Compiler ..............32
    5.1.2 Parsing Source Code Using ANTLR Parser ..........................33
    5.1.3 Using SourceKit ....................................................................33
    5.1.4 Creating Swift FSMM from the Output of the Extraction Utility ..38
  5.2 Conversion of Swift FSMM to CMM ..............................................38
  5.3 Verification of Swift Frontend for VizzAnalyzer ............................39

6 Results and Evaluation ......................................................................45
  6.1 Results .........................................................................................45
  6.2 Evaluation ....................................................................................45

7 Conclusion and Future Work ............................................................47
  7.1 Conclusion ...................................................................................47
  7.2 Future Work .................................................................................47

References ............................................................................................48

Appendix 1. The Algorithm of Conversion of Swift FSMM to CMM .........51
Appendix 2. Projects Used for Verification of the Swift Frontend ............53
Appendix 3. Results of Verification of the Swift Frontend .......................55
List of Figures

Figure 1 Composition of program comprehension tools..............................15
Figure 2 Overview of VizzAnalyzer architecture.......................................18
Figure 3 Architecture of low-level analysis engine (LLA)..............................18
Figure 4 Swift program demonstrating the use of classes and protocols ..........22
Figure 5 Swift program demonstrating how extensions can add protocol conformance to a type..........................................................23
Figure 6 Swift program demonstrating default implementation of a method in a protocol extension.......................................................23
Figure 7 Swift program demonstrating the use of Swift enumerations ............24
Figure 8 Swift program demonstrating value type immutability.....................24
Figure 9 Swift program demonstrating the use of tuples............................25
Figure 10 Swift program demonstrating the use of type aliases....................25
Figure 11 Swift program demonstrating the use of optional types..................25
Figure 12 Swift program demonstrating the use of guard statement..............25
Figure 13 Swift program demonstrating the use of for and forEach statements ...26
Figure 14 Swift program with declarations and statements in the global scope ...26
Figure 15 The proposed architecture of Swift program comprehension tool......31
Figure 16 A trivial Swift program serving as input for AST dump generation....32
Figure 17 AST dump for the trivial Swift program......................................32
Figure 18 A Swift program serving as input for structure and index requests.....34
Figure 19 Result of index request for the program in Figure 18.....................34
Figure 20 An example of structure request result......................................35
Figure 21 Algorithm of extraction of program model from source code..........36
Figure 22 An example of combined result of index and structure requests for the program in Figure 18 produced with algorithm in Figure 21 ..........37
Figure 23 Algorithm of conversion of program model JSON into Swift FSMM .38
Figure 24 A test Swift program featuring a simple class............................40
Figure 25 CMM graph for the test program in Figure 24............................41
Figure 26 A test Swift program featuring inheritance and extensions ..........41
Figure 27 CMM graph for the test program in Figure 26............................42
Figure 28 A test Swift program demonstrating advanced type system features..42
Figure 29 CMM graph for the test program in Figure 28............................43
Figure 30 A test Swift program featuring control flow structures...............43
Figure 31 CMM graph for the test program in Figure 30............................44
List of Abbreviations

AST – Abstract Syntax Tree
CMM – Common Metamodel
FSMM – Frontend-specific Metamodel
LLA – Low-level analysis engine
HLA – High-level analysis engine
CFE – Compiler frontend
FE – Filtering Engine
AE – Annotation Engine
VE – Visualization Engine
ANTLR – Another Tool for Language Recognition
JSON – JavaScript Object Notation
IDE – Integrated Development Environment
USR – Unified Symbol Resolution
XML – Extensible Markup Language
API – Application Programming Interface
URL – Uniform Resource Locator
LOC – Lines of Code
HTTP – Hypertext Transfer Protocol
1 Introduction

1.1 Problem Background

Software systems deal with objects of the real world, relations, and interactions between them. According to the Greek philosopher Heraclitus, "change is the only constant," which, of course, is true for software as well. The way software is used changes, requirements for it change, the environment it runs in changes; therefore software has to change as well to stay relevant.

Another known quality of the real world is its complexity, and, apparently, any useful software is inevitably complicated as well. Human cognition is limited in its capability of grasping the full complexity of the world, which means that software designed by humans contains flaws.

Maintenance is the software development process which deals with adapting the software to the changing world and correcting the flaws it contains. According to [1], it is one of the core processes in software development lifecycle, along with requirement collection, design, implementation, verification, and installation.

Software systems are often used for a time span much longer than their development time. One of the extreme examples is Unix, which has been in service since the 1970s. Over this time, it has grown substantially, now comprising millions of lines of code. Besides that, over the same time processors became 100 times faster, the memory capacity of a typical computer grew by orders of magnitude. Computers changed from huge mainframes occupying a building to become small devices that fit in a pocket or on the wrist. Typical users evolved from highly educated engineers to include almost every possible group of people. The world around Unix has changed dramatically, and it had to adjust accordingly.

The long lifecycle of software means significant changes in its environment and requirements. Multiplying this amount of time by the number of different types of maintenance activities leads to the conclusion that maintenance of a software system causes the significant part of its cost. [2] estimates this part to be around 50-75%.

The process of software evolution has been studied for a long time by many researchers who identified laws which govern this process. For instance, one of the software evolution laws discovered by [3] states that the need for adaptation requires the software system to be modified; otherwise, it becomes less useful. Another law says that these modifications inevitably increase its complexity unless deliberate effort is directed against this. As [4] points out, software becomes more fragile and resistant to change in the process of evolution. This means that any subsequent changes are likely to cause the system to fail, making maintenance more difficult and expensive. [5] identifies two leading causes for this: architectural erosion and architectural drift. The former means that architecture is violated, and the latter that developers become less aware of the architecture, which makes its application less consistent across the system. This exacerbates the growing complexity and increases disorder in the system.

To summarize, the software has to change constantly to stay useful. The changes required for that are applied in the process of software maintenance. In the process of evolution software systems become larger, more complex, less orderly and thus more fragile, resistant to changes and difficult to understand.

Understanding the system is a crucial factor for the success of its maintenance. Before attempting any modifications to the system, maintainers have to gain a solid understanding of it by collecting as much information as possible.
According to [6], maintainers need to know what the system does, how it interacts with the environment. They also need to know what parts it consists of and how they communicate during execution. Knowing the composition of the program helps understand which of its elements need to be changed to achieve the desired effect and how the changes are going to affect the rest of the program. [7] estimates the effort spent understanding the system to be around 50% of the total maintenance effort. [8] estimates the cost of time developers at Hewlett-Packard spend reading the code (an essential part of the understanding process) at $200 million a year.

One of the ways to obtain the information about the system is to contact the original developers. However, this is not always possible, because when the system is developed and maintained over an extended period, development teams may change. Even if it is possible to contact the developers who worked on the system before, it may happen that their knowledge of the system has eroded. People tend to forget things because human memory is imperfect. For this reason, it is important to have documents which contain information about the functionality and structure of the system, design decisions made in the process of development and rationale behind them.

Reading program documentation is another way. The documentation contains the concentrated knowledge about the system and some artifacts (such as the rationale behind the design decisions) which may be unavailable in other knowledge sources. Unfortunately, as pointed out by [9], architecture documentation is often either outdated or missing because of lack of budget, time or discipline, or the need to reuse undocumented legacy software.

In a situation when relying on documentation is not possible, the primary means of getting knowledge about the software system are reading its source code and observing its behavior at runtime. Unfortunately, getting information this way is less efficient because these sources produce a lot of information which is not useful, so knowledge extraction becomes harder or even completely impractical.

1.2 Problem Motivation

A way to make the extraction of information from the source code more efficient is to use comprehension support tools. [10] defines the aim of these tools as support of human cognition, assistance, and improvement of the process of thinking. Tools supporting program comprehension need information about the structure of the analyzed program. The source of this information may be abstract syntax tree (AST), a directed rooted tree generated by the compiler as one of the intermediate representations of the program in the process of compilation.

The fact that software comprehension tools rely on information produced by the compiler means that they can be developed by the same people who work on the compiler. However, in case the programming language is new, there might be more important goals, such as completing the implementation of language itself, so the development of comprehension tools is deprioritized. If the specification of the compiler is open, the community can develop tools supporting comprehension tasks. On early evolution stages of the language this community is small, and there might not be enough developers with sufficient expertise, so the development of comprehension support tools may take longer.

One of the examples of such new programming languages is Swift. It was released by Apple in 2014 after four years of development. The idea behind Swift was to combine the best features of other programming languages to be able to cover a variety of low-level (where performance is crucial) and high-level use cases
(where it is essential to be able to model complex application domains while keeping the code clear and easy to understand). Another goal was to maintain seamless compatibility with C and Objective-C to enable gradual migration of existing codebases with minimal effort required from developers.

The bar was set high, so development process took a lot of consideration and effort. The Swift team also chose to involve the community in the process by accepting improvement proposals. So far almost 200 community proposals were reviewed, extensively discussed and implemented (see [11]). As of early 2018, Swift language is still not complete. Essential features like binary stability, complete generic types, memory ownership model are still missing; according to [12] the plan is to implement them in Swift 5.0, the next major release.

In such situation, it is unsurprising that tooling infrastructure has been lagging behind. Xcode, the standard IDE for Swift development, introduced basic refactoring capabilities such as symbol renaming and method extraction only in version 9 released in September 2017 (see [13]), more than three years after the initial release of Swift. Such pace is unsatisfactory, so help from the community is obviously needed.

1.3 Problem Formulation

Even though Swift has many innovations, the people who designed it wanted it to feel familiar to programmers using languages from the C family (C++, C#, Objective-C, Java, and others), so many concepts are similar to these languages. For example, many Swift entities (control flow constructs – conditional, loop statements, functions, variables – local, class and static, classes, interfaces, enumerations) can be found in Java as well. Same is valid for relations between the concepts (inheritance – implements, extends, overrides; containment; calls; references). Although some features are language-specific (such as abstract classes, packages in Java; free functions, class extensions, enumeration members with associated values in Swift), the shared portion is significant enough to say that there is a common model that is suitable for representing programs in these languages. This means that there is a set of analyses and refactorings which can operate on this model without the need to consider specific implementation for every language.

VizzAnalyzer is a program analysis framework that features several analyses and visualization techniques in a way that makes it possible to combine them to address new use cases, which makes it a suitable basis for the development of program comprehension tools. However, Swift has several features which might not be expressible with CMM (such as class extensions, free functions, associated values for enumeration members, calculated properties). This means it is not clear whether VizzAnalyzer in its current state is suitable for Swift. It is also unclear what kinds of information required to generate an instance of CMM can be extracted from a Swift program using the existing tools, which might be another obstacle to using VizzAnalyzer as the basis.

This work explores the possibility of creating comprehension tools for Swift programs by answering the following research questions:

- To what extent can Swift programs be expressed using the Common Meta Model (CMM), which is the basis for the VizzAnalyzer framework?
- How can the information required for creating CMM models be extracted from the Swift programs?

Answering these questions will lay the groundwork for building comprehension tools for Swift, which will reduce the effort of developing and
maintaining Swift programs. An attempt will be made to build a prototype of Swift frontend for VizzAnalyzer as a proof of concept. The prototype will be verified on a set of open-source Swift projects.

1.4 Contributions
This work contributes to the field of program comprehension by analyzing the differences and similarities between Swift and other existing, widely-adopted programming languages such as Java and C#. Besides that, the formal model of Swift is defined, as well as its mapping to CMM, a mature and well-researched programming language metamodel used as the basis for a variety of program analyses and visualizations in the VizzAnalyzer framework.

In practice, this means that VizzAnalyzer becomes a tool which can assist comprehension of Swift programs. This makes it possible to study Swift in-depth and compare it with other programming languages using the same set of analyses. In addition to that, developers who use Swift in their projects can benefit from increased productivity and develop larger software projects with less effort.

Bridging the gap between Swift and the VizzAnalyzer framework also provides infrastructure for developing new analyses and visualizations, which enables further improvement and extension of comprehension tools for Swift.

1.5 Target Groups
This work might be of interest to the following groups:

- Researchers in the field of program analysis who might be interested in studying Swift.
- Developers and maintainers of large-scale Swift projects, who need help understanding large, unfamiliar programs or monitoring the quality of project continuously during the development process.
- Developers of Swift compiler, who might be interested in analyzing the impact of individual language features on programs.
- Developers of program analysis tools, who might want to reduce the development effort by reusing an existing platform.

1.6 Report Structure
The remainder of this report is organized as follows:

- Chapter 2 contains an overview of related work, as well as the key concepts in the area of software maintenance, program comprehension, program comprehension tools, program metamodels and describes the architecture of VizzAnalyzer.
- Chapter 3 includes an overview of the most distinguishing and interesting features of Swift.
- Chapter 4 describes the approach taken to answer the research questions.
- Chapter 5 describes the realization of this approach.
- Chapter 6 summarizes the achieved results and discusses them.
- Chapter 7 contains the conclusion and suggests directions for future work.
2 Background

This chapter starts by giving an overview of related work in section 2.1 and proceeds by introducing the fundamental concepts and terminology concerning software maintenance, program comprehension, program comprehension tools and program metamodels in section 2.2. A brief description of VizzAnalyzer architecture can be found in section 2.3.

2.1 Related Work

Program comprehension is described by [6] as the process of reconstructing the knowledge about the program. According to this work, maintainers form a mental model of the program as a result of this process. They use this model later as a working model of the system to be able to reason about it. The process of program comprehension involves reading the documentation, source code, observing the behavior of the program at runtime.

As [10] points out, a way to make the extraction of information from the source code more efficient is to use comprehension support tools. The aim of these tools is support of human cognition, assistance, and improvement of the process of thinking. The same author draws an analogy with the use of pencil and paper to help humans divide large numbers, pointing out that augmenting human memory in this manner increases its capacity. He also describes three different theories which explain how human cognition process can be supported. One of them is redistribution (reducing the load on internal resources by redistributing partial results to the external resources). Others include perceptual substitution (make a task easier by transforming it to rely on the operations which human brain can perform faster), and ends-means reification (simplifying problem solving by materializing parts of the problem space). This means that human cognition can be supported in a variety of different ways all of which can be automated using software tools.

[14] researches the difference between sentential (i.e., a series of sentences) and diagrammatic representations of the same information. The same paper also contains a conclusion that diagrammatic representations are superior because they reduce the amount of search needed to solve a problem by grouping related elements together and automatically support perceptual inferences which are very easy for humans to make. This means that software comprehension tools can be useful not only when documentation is missing or incomplete, but also when it is sentential in case they can represent the software system diagrammatically.

[15] in agreement with [10] and [14] points out that the process of understanding a program by humans can be made more efficient by reducing the load on their short-term memory and representing the system in a way which is less cognitively taxing. In alignment with the model of software comprehension proposed by other researchers, the author identifies some information needs by fulfilling which the tools can assist this process. These needs include, for example, the structure of project and organization of components in it, distribution of functionality between the modules, inheritance, function call and control flow graphs, usage information for types and objects.

[16] argues that program comprehension tools function by applying analyses and visualizations to a model of the analyzed program. The authors propose a language-independent metamodel called Common Meta Model (CMM). Instances of CMM (models of concrete programs) are intended to serve as a data repository for analysis and visualization of programs.
The same work also proposes a clean, decoupled architecture for analysis tools featuring clear separation of concerns by defining language-specific, analysis-specific and intermediate (common, language-independent) models. This architecture separates the common components performing the filtering, analysis, and visualization based on the language-independent model from the language-specific frontends generating this model by processing the source code of a program. This architectural vision is implemented in VizzAnalyzer [17], a framework combining program analysis and visualization techniques with the goal of simplifying program comprehension.

Clear separation of language-specific frontends, analysis and visualization components promotes their reuse and makes it easier to extend VizzAnalyzer with new languages, analyses, and visualizations, as well as combining the predefined ones to suit the specific task. [18] explains how this can be leveraged for rapid development of program comprehension tools. As a result, VizzAnalyzer supports analysis of programs in C#, Java, Objective-C, as well as several less widely used languages. [19] defines a software quality model and relies on VizzAnalyzer for extracting it from programs.

Program comprehension, the tools which support it, and formal models of programs have received plenty of attention from the academic community. The same, however, cannot be said about the Swift programming language. A search for scientific papers dedicated to Swift performed when the work on this project started (in March 2017) did not yield any results. A possible explanation for this could be that Swift was still relatively new and had not yet become popular enough to gain the attention of the academic community.

2.2 Theoretical Background

2.2.1 Software Maintenance

The primary goal of software development is to create a software system which satisfies user requirements [20]. These requirements can be functional and non-functional. Functional requirements describe the desired behavior of the system; the non-functional ones concern aspects such as performance, efficiency, and effectiveness, security, safety, scalability, reliability, modifiability, usability, compatibility and interoperability, regulatory compliance (see [21]).

User requirements change: new requirements are added, existing ones can change. Discrepancies between desired and actual behavior can be discovered while using the software. Software functions in an environment which includes other software, hardware and communication networks. This is the operating environment of software. When user requirements or operating environment change, software needs be modified to accommodate the changes to stay relevant and useful. This process is referred to as software maintenance.

Software maintenance includes the same activities as software development: analysis, design, implementation, documentation, testing, integration. Additionally, there are several activities specific to the maintenance process:

- **Program comprehension** (understanding what parts a software system consists of and how they interact to provide certain functionality).
- **Impact analysis** (assessing which parts of the system will be affected by a change and how).
- **Reengineering** (modifying the system without changing its behavior to improve its design and maintainability).
The goal of software maintenance activities is to fulfill the tasks which according to [22] can be classified as:

- **Corrective** (fixing the faults discovered during its use).
- **Perfective** (implementing feature enhancements in response to changing user requirements).
- **Adaptive** (modifying the system to support changes in operating environment).
- **Preventive** (modifications aimed at improving the maintainability).

Non-corrective tasks dominate the maintenance effort, constituting about 80% of all maintenance costs (according to [22]).

The organization or persons performing software maintenance are referred to as maintainers, as opposed to developers, who are involved during the development phase. This distinction underlines the fact that maintainers often have to work with software which they did not develop and thus are not familiar with.

### 2.2.2 Program Comprehension

Before maintainers attempt any modifications of the maintained system, they need to have the following information:

- **Requirements** (how should the system behave? what are the expected inputs and outputs?)
- **Structure of the system**, usually referred to as its architecture (what are the components of the system? what are their responsibilities? how do they interact with each other?).
- **Design of the system** and rationale behind the design decisions (why is the system designed this way? what algorithms and data structures were used? what were the alternatives when certain design decisions were made and what were the tradeoffs?)

This information is required to be able to efficiently identify the part of the system which needs to be modified, the other parts which interact with it and therefore might be affected by the modification, the proper way to change the system to avoid violating the existing design constraints. The process of gaining the knowledge about the system is software comprehension. Maintainers can obtain this knowledge from developers or software artifacts such as documentation and source code.

As a result of program comprehension developers construct a representation of the program in their mind (a mental model) following a particular cognitive strategy. Researchers have proposed several classifications of program comprehension strategies. One classification concerns the direction between higher and lower level of abstraction. Top-down strategies identified by [23] imply that programmers start on the higher level of abstraction by generating a hypothesis about the purpose of a program and then trying to confirm it. Bottom-up strategies identified by [24] put the starting point of the comprehension process on the lower levels of abstraction. Programmers start by reading the code and gradually constructing the mental model of the program on higher levels of abstraction by identifying and grouping the specific concepts on the lower level.

Both when using top-down and bottom-up approaches programmers try to take advantage of beacons, i.e., familiar features in code which signify the presence of specific structures. Recognizing these structures makes comprehension easier and faster. For example, the presence of classes with particular names may indicate
that specific object-oriented design pattern was used, which helps direct the process of comprehension.

[25] argues that programmers do not use only one type of strategies, switching between top-down and bottom-up models as needed instead (a combined strategy). The choice of particular strategy depends on how familiar the domain is for the programmer: top-down strategies are preferred for familiar domains, bottom-up ones are chosen when the domain is less familiar.

Another classification concerns breadth. [26] identifies two types of strategies: systematic and as-needed (opportunistic). The former means that programmer tries to gain a complete understanding of the program before attempting any modifications, the latter means that programmer takes the more pragmatic approach by focusing on the minimum necessary part of the program. This may result in lack of complete understanding, making modifications more error-prone. While complete understanding is preferable, [27] argues that it is not practical for large-scale programs.

The choice of program comprehension strategy depends on the several factors. One of them is how well the program is designed and documented. Another one is the programming language and paradigm used (procedural, object-oriented). Individual characteristics of a programmer (experience, ability, creativity), the task at hand (fixing an error, refactoring, redocumentation, reusing code, analyzing the impact of a proposed change), and its scope should also be considered.

2.2.3 Program Comprehension Tools

Regardless of the cognitive strategies employed, program comprehension can be made more efficient and productive with support from tools. The need for comprehension supporting tools is motivated by the scale of software and limited mental capacity of programmers; both factors make unassisted program comprehension time-consuming and impractical.

According to [28], the features provided by comprehension tools can be classified into the following categories:

- **Browsing and navigation**: with top-down and bottom-up strategies programmers navigate between different parts of a program on different levels of abstraction (modules, classes, methods, program statements) to find beacons. Better navigation and browsing can make the process of finding beacons more efficient.
- **Search**: tools can help finding code artifacts by their full name or pattern.
- **Querying**: tool support is beneficial for answering queries about the role of the given code artifact (e.g., variable, function).
- **History**: tools may store the history of navigation (recent locations in code), searches or other queries so that it is possible to return to them quickly.
- **Different views**: tools may represent the program in different ways to support various comprehension tasks. These views include a textual representation of program code, graph of classes and relations between them, function call graph.
- **Code analysis**: tools may analyze the code to identify clusters of code artifacts or calculate specific metrics.
- **Cognitive support**: tools may reduce cognitive load by giving quick access to the essential information about the program depending on the task and context. For example, information about the signature of the called
function (description, return type, parameter names and types) may be shown on request.

2.2.4 Architecture of Program Comprehension Tools

Each of the wide variety of program comprehension tasks described in the previous section requires different information about the analyzed program at a different level of abstraction. [18] argues that such variation makes it impossible to predefine a set of combinations of analyses, filtering, and visualizations that would cover all possible use cases. Instead, the authors propose an architecture which separates these concerns, making it easier to combine them as needed later. [18] identifies two major concerns for program comprehension: program models and views on these models. Program models are defined as different ways of representing the program. There is a base model extracted directly from the source code, as well as more abstract models produced from the base one by filtering it and applying a set of transformations. Filtering removes the information not relevant to the current comprehension task, transformations add or delete information by using a combination of program analysis, abstraction or merging steps. Filtering and transformations are referred to as mappings between models. The mappings are designed to be configurable. For example, filtering can be configured by specifying the kinds of information about the program that needs to be removed.

In addition to program analysis, which is defined by mapping between models, the results of program analysis need to be represented in a way which is understandable for humans. This is achieved with visualization. In [18] visualizations are defined by views on the program models, while the steps required to produce view models from program models are referred to as view mappings.

An architecture of program comprehension tools which incorporates the ideas mentioned above is displayed schematically in Figure 1. [18] argues that such architecture simplifies the development of comprehension tools. Instead of programming a tool from scratch for each comprehension task, it should be sufficient to use a combination of existing configurable model and view mappings.

![Figure 1 Composition of program comprehension tools](image)

As for the program and view models, [18] points out that software can be accurately described by the entities it contains and relations between them. Therefore, a graph is proposed to be the suitable data structure for the models. Nodes of the graph represent the program entities; edges describe the relations between them. Properties of the program entities can be expressed as attributes of the nodes. Mappings can be applied by removing nodes and edges, adding new ones deriving new semantic information and adding it as attributes to the nodes and edges.
The base model is extracted from the source code. The extracting tool parses the source code and creates an abstract syntax tree, storing the intrinsic properties of the program entities as attributes. This tool is similar to a compiler, although there are differences in the kinds of information that may be captured or discarded. For example, code comments are stripped by the compiler, but since they may contain information useful for comprehension, they should be preserved by the extractor.

Another key idea from [18] is that the base model can be reused for programs written in different languages, provided the kinds of entities and relations between them are the same or have a similar meaning. This enables application of the same set of analyses to programs written in different programming languages, which means that comprehension tools for a new programming language can be developed simply by defining a way to extract instances of the base model from programs in this language.

2.2.5 Program Metamodels

The architecture of program comprehension tools discussed in the previous section requires a clear distinction between model and metamodel. The former represents a concrete program, and the latter defines all possible models of programs, i.e., the model of models (thus the name). Mappings between frontend-specific, common and view models are defined by metamodels. Besides that, there is also a meta-metamodel, a common model which defines frontend-specific, common and view metamodels.

Each frontend-specific metamodel (FSMM) can capture programs in a concrete programming language. The entities and relations between them are part of the language specification and cannot be changed, but they are known, therefore can be described by a metamodel. An FSMM can be denoted as follows:

$$M_F = (G_F, R_F),$$

where $G_F$ is a tree grammar describing the language entities and the rules of their containment, $R_F$ is a set of possible semantic relations between the entities. $G_F$ in its turn can be formally described as:

$$G_F = (T_F, P_F, prog_F),$$

where $T_F$ denotes a set of entity types, $P_F$ is a set of productions describing the containment relations between entities and $prog_F$ is the entity type of structural tree root, $prog_F \in T_F$. Productions $P_F$ can be defined as:

$$P_F = \{ P_{F1}, P_{F2}, ... P_{Fn} \},$$

$$P_{Fi} : t ::= expr, t \in T_F$$

where each production $P_{Fi}$ is written in Backus-Naur form. $expr$ is either a sequence ($t_1 t_2 ... t_k$, an expression containing all of the terms $t_1$, $t_2$, ... $t_k$ in the given order), an iteration ($t^*$, an expression in which term $t$ is repeated 0 or more times) or a selection ($t_1 | t_2 | ... | t_k$, an expression expressing selection of exactly one of the terms $t_1$, $t_2$, ... $t_k$). $R_F$ can be denoted as follows:

$$R_F = \{ R_{F1}, R_{F2}, ... R_{Fn} \},$$

$$R_{Fi} = T_i \times T_2 \times ... \times T_n, T_i \in T_F, 1 \leq i \leq n, 1 \leq j \leq k.$$  

Common metamodel (CMM) describes all possible FSMMs, so it is logical to describe it in the same terms. It can be written using the same notation as FSMMs while omitting the frontend-specific index, e.g.:
\[ M = (G, R). \]

Mappings have to be defined to enable translation of an FSMM into CMM:

\[ \alpha_{FT} : T_F \rightarrow T, \]
\[ \alpha_{FR} : R_F \rightarrow R, \]

where \( T_F \rightarrow T \) and \( R_F \rightarrow R \) describe how frontend-specific entities and relations are translated to the common ones. As for \( \text{prog}_F \rightarrow \text{prog} \), in practice, there is only one possible translation.

A frontend mapping \( \alpha_F \) does not have to include translations for all frontend-specific entities and relations; some of them may be omitted without affecting any analyses based on the common metamodel. Moreover, this omission is the basis of abstraction the frontend mapping provides; including all entities and relations would mean the inclusion of unnecessary information in the common metamodel. When an entity is excluded, its children are added to the set of children of its direct ancestor.

At the same time, some entities or relations defined in common metamodel may not be provided by specific frontends, which means that the analyses requiring these entities or relations are not applicable (other analyses, however, may still be applied).

Analyses use the information contained in the common model of the program to perform the required computations. However, the common model includes information not needed by the analysis, so it makes sense to exclude this information. Views on the common model define the information required by analysis, serving precisely this purpose.

A \textit{view} specifies an abstraction over the common model, containing a subset of its entities and relations. It is, therefore, logical to define the view in the same terms:

\[ V_A = (G_A, R_A), G_A \subseteq G, R_A \subseteq R, \]

where \( G_A \) is a tree grammar defining view entities and their containment, \( R_A \) is a set of relations between them. \( G_A \) and \( R_A \) are defined using the same notation as tree grammar and relation set for common metamodel (\( G \) and \( R \) respectively).

Mapping \( \alpha_A \) between common metamodel \( M \) and view \( V_A \) is defined the same way as the mapping between frontend-specific model \( M_F \) and common metamodel \( M \). It also follows the same rules: some entity types are excluded, which means the respective nodes are removed. Children of the deleted nodes are attached to their respective direct ancestors. \( \alpha_A \), however, is not frontend-specific, but rather specific for a set of analyses \( A \).

2.3 \textit{VizzAnalyzer} Architecture

Section 2.2.4 describes a program comprehension tool architecture which defines several layers; each of these layers produces a model of the comprehended program at a higher level of abstraction. \textit{VizzAnalyzer} \cite{17} is a program analysis framework designed to conform to this architecture. \textit{VizzAnalyzer} consists of the following components:

- Low-level analysis engine (LLA).
- High-level analysis and metrics engine (HLA).
- Visualization engine (VE).

LLA produces the common model of a program from its source code in the form of graphs according to the configuration which specifies the types of nodes.
required for the comprehension task. HLA processes these graphs by adding new nodes, relations and annotations calculated using specialized algorithms and invokes VE which lays out the nodes and edges in a way most suitable for consumption by humans, outputs the images and provides a user interface for navigating, zooming and aggregation. Figure 2 gives an overview of the architecture of VizzAnalyzer.

![Figure 2 Overview of VizzAnalyzer architecture](image)

2.3.1 Low-level Analysis Engine (LLA)

LLA consists of the following parts:
- Compiler Frontend Engine (CFE).
- Filter engine (FE).
- Annotation engine (AE).

The CFE is similar to the actual compiler in that it produces a structural graph of the program from its source code. However, the compiler is much more complex because it needs to perform several checks of the program to make sure it is correct (for example, that the source code is compliant to the language syntax, all types are correct, all referenced declarations are available). At the same time, CFE may assume that the program is correct.

The FE filters the structural graph produced by CFE by removing the information not needed for the current comprehension task. For example, when the goal is to understand the inheritance structure of a program, information about method calls and field accesses is irrelevant. It is important to note though that the level of abstraction remains the same.

Filtering is performed according to the configuration. An assumption is made that in general more node and edge types need to be removed, so the configuration specifies the types of nodes and edges that need to remain in the graph.

Structural graph typically includes nodes corresponding to the program entities such as classes, methods, variables, control flow structures, and edges expressing the containment relations between them. However, some important information such as inheritance relations (e.g., "extends", "implements", "overrides"), method calls, field accesses exists only as attributes of nodes. It is the goal of AE to process this information and add corresponding nodes and edges to the structural graph to simplify higher-level analyses. Figure 3 visualizes the architecture of LLA.

![Figure 3 Architecture of low-level analysis engine (LLA)](image)

The structural graph produced by LLA contains nodes annotated with additional information (e.g., type, access level), each corresponding to an AST
node, as well as edges between them expressing containment, inheritance, and access relationships.

2.3.2 High-level Analysis Engine (HLA)

The structural graph output by LLA represents the program on a low level of abstraction. A program may contain thousands of classes, each having several members (fields, methods); each method may include several variables, control flow structures, field accesses, method calls. This means that the graph contains a lot of information (even when filtering is applied on the LLA level), and this amount grows as the size of the comprehended program increases, which means it is difficult or even impossible to understand for humans. This motivates the need to reduce the amount of information, which can be achieved by producing more abstract models of the program based on the graph, which is precisely the goal of HLA. Of course, reducing the amount of information means loss of accuracy, but this tradeoff is acceptable because achieving comprehension is more important.

HLA contains analyses of two kinds:
  • Graph analyses (GA).
  • Metric analyses (MA).

GA reduce the amount of information by filtering nodes and edges of certain predefined types and aggregating specific groups of nodes by replacing each of them with a single representative node based on a predefined aggregation function. MA calculate different software metrics (e.g., class, control flow complexity, software quality metrics) and attach the values to the nodes. HLA may apply series of GA and MA to achieve the model of the program at the level of abstraction appropriate for the comprehension task.

2.3.3 Visualization Engine (VE)

Abstract models of the program produced by HLA are graphs. Graphs are more readily understood by humans if they are visualized, i.e., represented as an image. Generating an image of the graph requires choosing appropriate metaphor, which includes images for its nodes and edges serving as their visual representation and their layout, i.e., their position in space.

Different metaphors exist depending on the kind of space and the image chosen for the nodes and edges. The most straightforward and most widely used metaphor comprises simple geometric shapes (rectangles, circles) representing nodes laid out in two-dimensional space. The shapes are connected with lines representing edges. Different node and edge attributes can be expressed with text labels, different colors, shape kinds, line width. Similar metaphor exists which relies on geometric bodies in 3D space. According to [29], some metaphors try to assist the process of understanding even more by mapping graph nodes to real-world objects like building. The whole graph then becomes a city, and node attributes are expressed using different textures and sizes, relations between the nodes can be represented by the proximity of the nodes (e.g., if several nodes are located close to each other, they belong to the same group). This approach leverages the natural understanding of surrounding physical world humans have.

VE is the component which generates the visual representation of the structural graph of the program produced by HLA. It also provides the user interface for navigating the resulting image interactively (zooming out or in, following the edges, displaying the neighbor nodes).
2.3.4 Definition of Common Meta Model

This section contains the concrete definition of Common Meta Model (CMM) provided by VizzAnalyzer. Consistent with notation used in section 2.2.5, CMM can is defined as:

\[ M = (G, R) \]

\[ G = (T, P, prog) \]

\[ T = \{ \]

Project, Library, Directory, File, Scope,
Type, ClassType, PrimitiveType, Interface, Enumeration, Class, PartialClass,
AnonymousClass, GenericType, GenericClassType, GenericMethodType,
Member, Initializer, Constructor, Field, Method, FormalParameter, Operator,
EnumerationValue, Property, Variable, Indexer, Element, Delegate,
ComplexStatement, Condition, Do, ElseBlock, EnhancedFor, IfBlock, If,
For, Loop, Statement, SwitchBlock, Switch, While,
Reference, FileRef, TypeRef, GenericInstanceRef, DependsRef, InheritanceRef,
ExtendsRef, ImplementsRef, AccessRef, CallRef, IndexerRef, MemberRef,
ConstructorRef, DefaultConstructorRef, ExplicitConstructorRef, FieldRef,
EnumerationValueRef, PropertyRef, MethodRef, ReturnTypeDef, OperatorRef,
FireEventRef, AddEventMethodRef;
Event, OutOfScopeEntity, NotYetDefinedEntity, UndefinedEntity, UnknownEntity

\[ \}

\[ P = \{ \]

Scope ::= Library* Project*,
Library ::= Directory* File*,
Project ::= Library* Directory* File*,
Directory ::= File*,
OutOfScope ::= Library* Type*,
File ::= Type* FileRef* Member*,
Type ::= Property* Constructor* Field* Method* Member*,
Constructor ::= Variable* Statement* FormalParameter* ConstructorRef*,
DefaultConstructorRef* FieldRef* MethodRef*,
Initializer ::= Variable* Statement* FormalParameter* ConstructorRef*,
DefaultConstructorRef* FieldRef* MethodRef*,
Field ::= TypeRef*,
Method ::= OverridesMethodRef* FormalParameter* Variable* Statement*,
ReturnTypeRef* ConstructorRef* DefaultConstructorRef* FieldRef* MethodRef*,
If ::= Condition IfBlock ElseBlock?,
Switch ::= SwitchBlock+,
For ::= Condition Statement*,
EnhancedFor ::= Condition Statement*,
While ::= Condition Statement*,
Do ::= Statement* Condition,
IfBlock ::= Statement*,
ElseBlock ::= Statement*,
SwitchBlock ::= Statement*

\[ \}

\[ prog = Scope \]

\[ R = \{ \]

FileRefEdge : FileRef × File,
ExtendsRefEdge : ExtendsRef × ClassType,
ImplementsRefEdge : ImplementsRef × Interface,
OverridesMethodRefEdge : OverridesMethodRef × Method,
GenericInstanceRefEdge : GenericInstanceRef × GenericType,
TypeDefEdge : TypeRef × Type,
\text{CallRefEdge} : \text{CallRef} \times \text{Member}
\text{AccessRefEdge} : \text{AccessRef} \times \text{Member}
\text{DependsRefEdge} : \text{DependsRef} \times \text{Library}
\text{MemberRefEdge} : \text{MemberRef} \times \text{Member}
\text{ConstructorRefEdge} : \text{ConstructorRef} \times \text{Constructor}
\text{DefaultConstructorRefEdge} : \text{DefaultConstructorRef} \times \text{Type}
\text{ExplicitConstructorRefEdge} : \text{ExplicitConstructorRef} \times \text{Constructor}
\text{EnumerationValueRefEdge} : \text{EnumerationValueRef} \times \text{Enumeration}
\text{FieldRefEdge} : \text{FieldRef} \times \text{Field}
\text{PropertyRefEdge} : \text{PropertyRef} \times \text{Property}
\text{MethodRefEdge} : \text{MethodRef} \times \text{Method}
\text{ReturnTypeRefEdge} : \text{ReturnTypeRef} \times \text{Type}
\text{IndexerRefEdge} : \text{IndexerRef} \times \text{Indexer}
\text{OperatorRefEdge} : \text{OperatorRef} \times \text{Operator}
\text{FireEventRefEdge} : \text{FireEventRef} \times \text{FireEvent}
\\
P^* = \{
\text{Element} ::= \text{Type} \mid \text{Statement} \mid \text{Variable} \mid \text{Reference} \mid \text{Member} \mid \text{Unknown} \mid \text{Undefined} \mid \text{NotYetDefined}
\text{Type} ::= \text{PrimitiveType} \mid \text{ClassType} \mid \text{Delegate} \mid \text{GenericType},
\text{GenericType} ::= \text{GenericMethodType} \mid \text{GenericClassType},
\text{ClassType} ::= \text{Class} \mid \text{AnonymousClass} \mid \text{Interface} \mid \text{Enumeration} \mid \text{PartialClass},
\text{Member} ::= \text{Initializer} \mid \text{Constructor} \mid \text{Event} \mid \text{Property} \mid \text{Operator} \mid \text{Method} \mid \text{EnumerationValue} \mid \text{Indexer} \mid \text{Field},
\text{Variable} ::= \text{FormalParameter} \mid \text{Field},
\text{Statement} ::= \text{ComplexStatement} \mid \text{IfBlock} \mid \text{SwitchBlock} \mid \text{ElseBlock},
\text{ComplexStatement} ::= \text{Condition} \mid \text{Loop},
\text{Condition} ::= \text{If} \mid \text{Switch},
\text{Loop} ::= \text{While} \mid \text{For} \mid \text{EnhancedFor} \mid \text{Do},
\text{Reference} ::= \text{MemberRef} \mid \text{TypeRef} \mid \text{DependsRef} \mid \text{FileRef} \mid \text{OverridesMethodRef} \mid \text{ConstructorRef},
\text{MemberRef} ::= \text{CallRef},
\text{CallRef} ::= \text{PropertyRef} \mid \text{OperatorRef} \mid \text{IndexerRef} \mid \text{MethodRef} \mid \text{FireEventRef},
\text{ExplicitConstructorRef},
\text{TypeRef} ::= \text{DefaultConstructorRef} \mid \text{GenericInstanceRef} \mid \text{ReturnTypeRef},
\text{InheritanceRef},
\text{MethodRef} ::= \text{AddEventMethodRef} \mid \text{AccessRef},
\text{AccessRef} ::= \text{EnumerationValueRef} \mid \text{FieldRef},
\text{InheritanceRef} ::= \text{ImplementsRef} \mid \text{ExtendsRef},
\text{ConstructorRef} ::= \text{DefaultConstructorRef} \mid \text{ExplicitConstructorRef},
\}

\text{P^*} is a special set of rules defining inheritance relations between the entities. Each rule \( T ::= T_1 \mid T_2 \) means that the production rules specified for \( T \) apply to the entities \( T_1 \) and \( T_2 \), i.e. \( T_1 \) and \( T_2 \) inherit the production rules for \( T \). This is done to avoid duplicating the production rules for similar entities. For example, \text{Class}, \text{PartialClass} and \text{AnonymousClass} are different kinds of classes and can contain the same kinds of members, so they all extend \text{Type} (via \text{ClassType}) for which the relevant production rules are defined. It is still possible to get the production rules for any entity by following its inheritance chain, therefore this way of defining the common metamodel is consistent with that used in section 2.2.5.
3 Overview of Swift Features

Swift is a multiparadigm programming language which combines the most interesting concepts from multiple other programming languages in an innovative way. This chapter gives an overview of Swift features and aspects. The list of covered features and language aspects is not exhaustive, and many details are omitted for brevity.

3.1 Type System

Swift supports object-oriented programming; classes with protocols (corresponding to interfaces) constitute the basis of Swift’s type system.

Swift types may have initializers (constructors), properties (fields), methods, operator functions (which provide implementations for operators) and subscript functions (providing implementations for the subscript operator, []). Initializers may throw errors and return nil (so-called failable initializers). Properties may be stored and computed. Stored properties are backed by a chunk of memory. Computed properties are not stored in memory. Instead, they must specify a getter (get) function which calculates and returns their value on each invocation and a setter (set) function which accepts a new value and processes it (possibly storing the result in another property). Properties may also define special observer functions, willSet and didSet, the former being called when the respective property is about to be set, the latter immediately after it was set. Properties and methods may be static, class and instance. Class properties and methods are similar to the static ones, but also participate in inheritance (e.g., can be overridden in a subclass). Swift program in Figure 4 gives an example of how classes, protocols, and their members can be used.

```swift
protocol SomeProtocol {
    var aStoredProperty: Int { get }
    func someMethod()
}

class SimpleClass: SomeProtocol {
    static var staticProperty: Int = 0
    var aStoredProperty: Int
    var aComputedProperty: Int { return aStoredProperty * 2 + 25 }
    var anotherComputedProperty: Int {
        get { return aStoredProperty * 100 }
        set { aStoredProperty = newValue – 100 }
    }
    var propertyWithObserver: Int? {
        didSet {
            print("propertyWithObserver changed: \(propertyWithObserver)")
        }
    }
    init() { aStoredProperty = 20 }
    func someMethod() { print("Just printing...") }
}
```

*Figure 4 Swift program demonstrating the use of classes and protocols*

Swift classes may define deinitializers. Deinitializer is a block of code run when an instance of a class is about to be deallocated. Deinitializers may be useful for freeing up resources used by the class. Although C# features finalizers, which are a similar concept, CMM does not define a special entity type for them. That said, deinitializers are essentially instance methods with additional semantics (they are called on deallocation of the instance) and can be mapped to CMM as such.

Type system in Swift also includes the concept of inheritance with basic rules similar to Java. A class may inherit from a single parent class and conform to (implement) multiple protocols. Hiding of superclass members is not supported, a
subclass member must override the superclass member with the same signature. Protocols may inherit from various other protocols. Besides specifying requirements for instance methods, protocols may also specify requirements for properties and initializers.

A significant feature of Swift type system is extensions. An extension may add new methods and computed properties to a type (including the types declared in the standard library and other libraries). Extensions are often used to add conformance to protocols or add more functionality to a type (see Figure 5). Members declared in extensions of types can be accessed the same way as those declared in the types itself.

```swift
protocol AnotherProtocol {
    func methodInAnotherProtocol()
}
extension SimpleClass: AnotherProtocol {
    func methodInAnotherProtocol() {}
    func someUnrelatedExtensionMethod() {}
}
let thing = SimpleClass()
thing.someMethod()
thing.methodInAnotherProtocol()
```

*Figure 5 Swift program demonstrating how extensions can add protocol conformance to a type*

Protocols can be extended as well. Protocol extensions cannot add requirements, but can provide methods and properties that satisfy the existing ones, in which case all types conforming to the protocol automatically acquire this implementation. Extension of AnotherProtocol in Figure 6 adds default implementation for methodInAnotherProtocol(). Just as in the other example, SimpleClass (as well as EvenSimplerClass) now conforms to AnotherProtocol, relying on the default implementation of methodInAnotherProtocol() in the protocol extension to satisfy its requirements.

```swift
extension AnotherProtocol {
    func methodInAnotherProtocol() {}
}
extension SimpleClass: AnotherProtocol {}
class EvenSimplerClass: AnotherProtocol {}
```

*Figure 6 Swift program demonstrating default implementation of a method in a protocol extension*

Possibility to provide methods and properties satisfying requirements of a protocol together with the possibility of a type conforming to multiple protocols resembles trait inheritance. Trait inheritance has been proposed by [30] as a compromise between single and multiple inheritance models. The compromise is needed because neither model of inheritance is satisfactory.

Single inheritance allows a class to inherit from at most one parent class, but, as [30] point out, this might not be enough to separate common features of several classes in complex class hierarchies; hence it may force duplication of code.

On the other hand, multiple inheritance allows a class to inherit from several parent classes, which gives more expressive power but introduces certain problems. The most prominent of these problems is so-called “diamond problem” which occurs when the parent classes contain conflicting field and method definitions. Various programming languages solve this problem in different ways, description of which can be found in [30].
Trait inheritance model is one of such solutions. It allows only single inheritance of classes, extending this model by enabling multiple inheritance of traits. Traits are collections of behavior which is contained in method definitions. Traits cannot be instantiated, nor can they have own state (i.e., field definitions are not allowed). In case a class inherits multiple traits containing conflicting method definitions, it must resolve the conflicts by providing own overriding definitions which may optionally reference the overridden ones. Protocol extensions in combination with multiple protocol inheritance in Swift can be considered the implementation of trait inheritance.

Besides classes, protocols, and extensions, Swift type system has another dimension: reference and value types. When an instance of value type is assigned to a variable or passed to a function, it is copied, while instances of reference types are not copied (instead of that, new references to the same existing instance are created).

Reference types in Swift are represented by classes; value types are represented by enumerations and structures. Just like classes, structures may have properties, methods, initializers, conform to protocols or be extended with additional functionality. Unlike classes, structures do not allow inheritance. Enumerations provide a common type for a group of related values, just like in other programming languages. Enumerations may include initializers, properties, methods, subscripts, operators (just like types of different kinds). An additional feature of Swift enumerations is associated values. This feature makes it possible to store (associate) additional values with any enumeration case (see Figure 7).

```swift
enum Drink {
    case beer(style: String, abv: Float, ibu: Int)
    case soda(flavor: String)
    case water
}
let fanta: Drink = .soda(flavor: "orange")
```

**Figure 7 Swift program demonstrating the use of Swift enumerations**

One of the most important features of value types in Swift is their immutability, which means that instances of these types cannot be modified by the instance methods. In multithreaded context, this guarantees that different threads can use instances of value types without the need for synchronization or risk of race conditions. If an instance method needs to modify the instance, it has to be explicitly marked as mutating. The requirement for explicit mutability annotations allows the compiler to reason about mutability, which helps detect situations where mutation might lead to incorrect or unexpected behavior. The example in Figure 8 demonstrates the use of value type immutability. c1 in this example is a constant, which means that it cannot be modified. c2 is variable, so its modification is allowed.

```swift
struct Counter {
    var value: Int = 0
    // modifies an instance property, has to be annotated as mutating
    mutating func increment() { value += 1 }
}
let c1 = Counter()
c1.increment() // error: cannot use mutating member on immutable value
var c2 = Counter()
c2.increment() // no error
```

**Figure 8 Swift program demonstrating value type immutability**

Another interesting feature of Swift type system is tuple types. Tuples are compound values consisting of other values of arbitrary types (see Figure 9).
let person = (name: "John Doe", age: 45)
print("Hello, my name is \(person.name), I am \(person.age).")

Figure 9 Swift program demonstrating the use of tuples

Tuples are intended to be used as temporary groups of related values. One situation where this is particularly useful is returning value from a function. It is possible to group multiple values into a tuple in place instead of defining an explicit special type just for this purpose.

Swift also allows creating aliases for existing types. The example in Figure 10 demonstrates how to use type aliases by creating an alias Person for a tuple type.

typealias Person = (name: String, age: Int)
let person: Person = ("John Doe", 45)
print("Hello, my name is \(person.name), I am \(person.age).")

Figure 10 Swift program demonstrating the use of type aliases

3.2 Optional Types

One of the goals of Swift is safety, in particular, null-safety. This goal is achieved by proving that null pointer dereferencing errors are not possible at runtime. Swift has a special feature that allows a programmer to mark variables that can hold null values (or rather nil, Swift’s keyword for null values). Swift program in Figure 11 demonstrates this.

let a: Int? = nil // optional integer, can hold null and non-null values
let b: Int = 4 // non-optional integer, can never be null
let c = b.hashValue // safe, will never crash
let d = a.hashValue // error: value of optional type 'Int?' not unwrapped

Figure 11 Swift program demonstrating the use of optional types

In the example from Figure 11, variable b can never be nil, so it is safe to dereference. Variable a, however, has to be checked before it is safe to dereference; otherwise the compiler produces a compilation error. Optional types are denoted by appending ? to the type name. The compiler turns optional types into instances of a special enumeration Optional<T>.

3.3 Control Flow

Swift provides all control flow structures more traditional C-like programming languages supported by VizzAnalyzer provide: for...in, while, do...while loops, if and switch statements. One exception is the plain for loops (which were originally included in the language but removed later).

There are also guard statements, a special kind of if statements with a condition and else block. They are different from the normal if statements in that they have an else block instead of if, essentially negating the condition and enforce the exit from the enclosing scope by requiring a return, break, continue or throw statement at the end of the else block (see example in Figure 12).

guard someCondition else {
    // someCondition is false in this block
    return // or break, continue, throw
}

Figure 12 Swift program demonstrating the use of guard statement

Swift provides alternative ways to iterate through collections. The built-in Sequence type (which is the common supertype of all collections, such as arrays,
dictionaries, sets) defines several functions (forEach, filter, map, flatMap, reduce), which accept a function executed for every item of the sequence. The example in Figure 13 demonstrates the use of for and forEach statements.

```swift
let things = ["foo", "bar", "baz", "qux"]
for thing in things { print("\(thing) is one of things") }

things.forEach { thing in print("\(thing) is one of things") }
```

Figure 13 Swift program demonstrating the use of for and forEach statements

3.4 Code Organization and Access Control

Files are the basic unit of code distribution in Swift. Most commonly a file contains a declaration of a single type, although it is not unusual that files contain several related type declarations, free functions or variables. Modules are the next, higher-level code distribution unit. A module is a collection of source files built together. A module typically corresponds to application bundle or framework. A module can reuse code in other modules by importing them using the import keyword.

Access control model is designed around the concepts of source files and modules. There are five different levels of access: private, file private, internal, public and open. Private declarations can be accessed only within the current type, its extensions or the current source file. File private declarations, as follows from the name, can only be accessed within the same source file. Internal declarations are accessible from all files within the same module. Public and open declarations are accessible in their module and other modules, the difference between them is that only open declarations can be extended (classes) or overridden (class members).

3.5 Code in Global Scope

Swift allows declaration of functions and variables in the global scope. Such functions and variables are called free. Besides declarations of types, functions, and variables, the global scope may also contain statements (such as function calls, control flow statements). Figure 14 contains an example of a program which contains declarations and statements in the global scope.

```swift
#!/usr/bin/env swift
// main.swift
import Foundation

guard let input = CommandLine.arguments.first else {
    print("Not enough arguments")
    exit(1)
}

func calculateResult(for input: String) -> Int {
    print("got \(input), doing work...")
    return 42
}

let result = doWork(for: input)
print("Done! Result for \(input) is \(result).")
exit(0)
```

Figure 14 Swift program with declarations and statements in the global scope

The purpose of allowing code in global scope is to make the language more flexible by not requiring all code to be contained in types. Scripting is the primary use case for this (the above example is a simple script), but surely not the only one (for example, global variables declared in the scope of a file may be used as part of an implementation of the Singleton design pattern).
4 Approach

This chapter explains how knowledge from the previous chapters is applied to solve the problem of missing program comprehension tools for Swift. It starts by describing how VizzAnalyzer framework can be extended. After that, an analysis of the compatibility of Swift features with CMM is given. Then the definition of Swift FSMM and its mapping to CMM are given. Overview of the proposed solution concludes this chapter.

4.1 Extending VizzAnalyzer

The architecture of VizzAnalyzer framework described in section 3.1 is modular, features clear separation of concerns, which makes extending the framework easier. LLA, HLA, and VE provide points of extension for adding new language frontends, analyses, and visualizations. Building a comprehension tool for Swift programs requires extending the LLA with a new language frontend.

The added language frontend needs to be able to create an instance of frontend-specific metamodel $M_S$ from Swift code and transform it into an instance of common metamodel according to mapping $\alpha_S$ between $M_S$ and $M$. The CMM instance can then be used as data store by analyses and visualizations in the VizzAnalyzer framework.

4.2 Analysis of Compatibility of Swift and CMM

This section walks through the aspects of Swift which are different from other programming languages supported by VizzAnalyzer (and therefore expressible with CMM) and discusses possible ways to express these aspects with CMM.

4.2.1 Type System

CMM cannot express the concept of optionality. However, since none of the existing analyses in VizzAnalyzer takes null safety into account, optional types can be treated as their non-optional counterparts.

Class inheritance in Swift is similar to Java. Therefore, it can be modeled with CMM. Type members in Swift are mostly analogous to type members in other languages, except for several distinctive features such as failable initializers, computed properties, property observers, and particular operator and subscript functions. Failable initializers are only different from the non-failable ones in that they may return nil values, but neither CMM nor any analyses based on it account for null safety. Therefore, failable initializers can be treated like the non-failable ones. As for the computed properties and property observers, they may be considered special methods without parameters (get) or with implicit parameters (oldValue, newValue for set, willSet, and didSet) using special syntax. Static members are analogous to their counterparts in other languages, class members can be treated as static, but if they override a superclass member, the corresponding relationship needs to be reflected in the metamodel. It is unclear though whether the analyses can correctly handle the "overrides" relationship between static members.

Swift extensions are similar to partial classes in C# which also allow splitting a type declaration into multiple parts. This means that extensions can be directly modeled with CMM. One significant difference is that all parts of a partial C# type have to be declared in the same module, while Swift allows extending types
declared in other modules and standard library. In such cases declaration of the extended type is unavailable, but can be replaced by an empty declaration.

VizzAnalyzer does not support the concept of mutability, so value types are mapped to the existing type entities (structures to classes, enumerations to enumerations).

Swift tuples can be considered the same as anonymous classes in C#, so the CMM entity for anonymous classes is suitable for mapping the tuples as well.

A type alias is nothing more than an alternative name for the aliased type. Therefore references to the alias can be treated as references to that type.

4.2.2 Control flow

Switch statements in Swift are similar to the traditional switch statements, with a few exceptions. Firstly, the compiler makes sure that the cases cover all possible values of condition either specifically or by providing a default branch. Secondly, branches do not "fall through," i.e., execution exits the switch statement after the last statement in a branch is executed (unless a special `fallthrough` statement is used), which is different from switch statements in C-like languages, where the following branches are executed until the break statement is encountered. This, however, does not affect how `switch` statements are mapped to CMM.

The `forEach`, `filter`, `map`, `flatMap`, `reduce` functions can be mapped to loop statements since using them is essentially equivalent to using a loop statement.

With all the above considered, CMM is sufficient to express Swift control flow statements.

4.2.3 Code organization and access control

Organization of source code and access levels in Swift are considerably different from C# and Java. Java organizes classes into packages, C# relies on namespaces for this purpose. A project may contain multiple nested packages and namespaces, in C# a namespace may also span multiple assemblies (projects). Besides that, Swift allows declaration of multiple public types per file, while Java enforces the policy that each file may only contain a single public type.

CMM allows declarations in the global scope, but statements are not allowed, so they have to be ignored.

As for the access levels, Swift and C# have similarities: both have internal access level; types and members marked internal are accessible by any code within the same module/assembly. The differences include the protected access level (featured in both C# and Java) which makes corresponding members accessible by subclasses, something Swift does not have. At the same time, Swift makes the distinction between public and open access levels, while both Java and C# only have the public access level.

Differences between Swift, Java, and C# discussed above make mapping Swift to CMM less straightforward. A module can be expressed as Project (an entity containing all code of the currently analyzed program). As for the source files, the intuitive approach would be to map each of them directly to File. However, the fact that Swift source files may contain multiple public declarations and similarity of the file/private access level in Swift and package-local in Java makes Swift source file conceptually closer to Java package (expressed as Directory in CMM).

CMM stores information about the access level of each type and member as node attribute. CMM does not provide formal definition of entity attributes, but
inspection of VizzAnalyzer code reveals that analyses expect to access levels corresponding to those in Java (private, protected, public and package-local). Considering this, open and public access levels in Swift can be mapped to public, private to private, fileprivate to package-local. There is no equivalent for the internal level, but it can be mapped to public. The rationale behind this decision is that internal members and types in Swift are accessible by any code in the same module, just as public types in Java can be accessed anywhere in the same project. The purpose of public access levels in Swift is to define the public interface for a module, but because VizzAnalyzer cannot analyze interaction between modules or encapsulation on the module (project) level, this decision does not result in loss of information (as far as VizzAnalyzer is concerned).

4.3 Mapping of Swift FSMM to CMM

This section contains the definition of FSMM for Swift and its mapping to CMM based on the description of Swift features given in the previous section.

4.3.1 FSMM for Swift

Consistently with the notation used for specifying a generic FSMM proposed in the section 2.2.5, based on the compatibility analysis performed in section 3.1, FSMM for Swift can be defined as follows:

\[ M_S = (G_S, R_S) \]
\[ G_S = (T_S, P_S, prog_S) \]
\[ T_S = \{ \]
  - Scope, Module, File,
  - Type, Class, Protocol, Struct, Enum, Extension, TypeAlias,
  - Member, Initializer, Deinitializer, Parameter, Statement, Property, Getter, Setter,
  - Observer, EnumerationValue, Method, Subscript, Operator,
  - If, Condition, IfBlock, ElseBlock, Guard, Switch, SwitchBlock, Loop, ComplexStatement,
  - While, RepeatWhile, For, ForEach, Filter, Map, FlatMap, Reduce,
  - ModuleRef, TypeRef, OverridesRef, MemberRef, InitializerRef, PropertyRef,
  - EnumerationValueRef, OperatorRef, SubscriptRef, MethodRef, ExtendsRef, ConformsRef,
\]
\[ P_S = \{ \]
  - Scope ::= Module*,
  - Module ::= File*,
  - File ::= ModuleRef* Type* Member*,
  - Type ::= TypeRef* Member*,
  - Initializer ::= OverridesRef* Parameter* Statement* TypeRef* MemberRef*,
  - Deinitializer ::= Statement* TypeRef* MemberRef*,
  - Property ::= OverridesRef* TypeRef* Getter Setter? Observer*,
  - EnumerationValue ::= TypeRef*,
  - Method ::= OverridesRef* Parameter* Statement* TypeRef* MemberRef*,
  - If ::= Condition IfBlock ElseBlock?,
  - Guard ::= Condition ElseBlock
  - Switch ::= SwitchBlock*,
  - Loop ::= Condition Statement*,
  - IfBlock ::= Statement*,
  - ElseBlock ::= Statement*,
  - SwitchBlock ::= Statement*
\]
\[ prog_S = Scope \]
$R_s = \{ $

ModuleRefEdge : ModuleRef × Module,  
TypeRefEdge : TypeRef × Type,  
ConformsRefEdge : ConformsRef × Protocol,  
ExtendsRefEdge : ExtendsRef × Class,  
OverridesRefEdge : OverridesRef × Member,  
MemberRefEdge : MemberRef × Member,  
InitializerRefEdge : InitializerRef × Initializer,  
EnumValueRefEdge : EnumValueRef × Enum,  
PropertyRefEdge : PropertyRef × Property,  
MethodRefEdge : MethodRef × Method,  
SubscriptRefEdge : SubscriptRef × Subscript,  
OperatorRefEdge : OperatorRef × Operator

$\}$

$P_s^* = \{ $

Type ::= Class | Protocol | Struct | Enum | Extension | TypeAlias,
Member ::= Initializer | Deinitializer | Property | Method | Operator | Subscript | EnumValue,
Statement ::= ComplexStatement | IfBlock | SwitchBlock | ElseBlock,
ComplexStatement ::= Condition | Loop,
Condition ::= If | Guard | Switch,
Loop ::= While | RepeatWhile | For | ForEach | Filter | Map | FlatMap | Reduce,
MemberRef ::= InitializerRef | PropertyRef | EnumValueRef | OperatorRef | MethodRef

SubscriptRef | MethodRef

$\}$

### 4.3.2 Mapping of Swift FSMM to CMM

Each FSMM needs to be translated into CMM. This is achieved by mapping entities and relations in FSMM to entities and relations in CMM. As discussed in section 2.2.2, direct mapping from Swift frontend-specific model to CMM is not possible, so this mapping is not trivial. Using notation from section 2.2.5, mapping from Swift FSMM to CMM can be described as follows:

$\text{ast} : \{ $

Scope → Scope, Module → Project, File → Directory,
Type → Type, Class → ClassType, Protocol → Interface, Struct → ClassType,
Enum → Enumeration, Extension → PartialClass, TypeAlias → Type,
Tuple → ClassType,
Member → Member, Initializer → Constructor, Deinitializer → Method,
Parameter → FormalParameter, Statement → Statement, Property → Field,
Getter → Method, Setter → Method, Observer → Method,
EnumValue → EnumerationValue,
Method → Method, Subscript → Method, Operator → Method,
If → If, Condition → Condition, IfBlock → IfBlock, ElseBlock → ElseBlock,
Guard → If,
Switch → Switch, SwitchBlock → SwitchBlock, Loop → Loop,
ComplexStatement → ComplexStatement,
While → While, RepeatWhile → Do, For → EnhancedFor,
ModuleRef → DependsRef, TypeRef → TypeRef, OverridesRef → OverridesRef,
MemberRef → MemberRef, InitializerRef → ConstructorRef, PropertyRef → FieldRef,
EnumValueRef → EnumerationValueRef, OperatorRef → MethodRef,
SubscriptRef → MethodRef, MethodRef → MethodRef, ExtendsRef → ExtendsRef,
ConformsRef → ImplementsRef

$\}$. 
The proposed solution to the problem of lack of program comprehension tools in Swift ecosystem is to take advantage of analyses and visualizations provided by VizzAnalyzer framework. This is achieved by extending the low-level analysis engine (LLA) of VizzAnalyzer with a Swift language frontend. This frontend generates an instance of FSMM for the comprehended program by querying SourceKit. The FSMM instance is then transformed into an instance of CMM by replacing frontend-specific entities and relations with their counterparts from CMM according to the mapping defined in section 4.3.2. The generated CMM instance can then be consumed by high-level analysis engine (HLA) as a data repository for analyses (e.g. clustering, metric calculations) and visualizations. Figure 15 provides an overview of the solution.

Figure 15 The proposed architecture of Swift program comprehension tool
5 Implementation

This chapter describes the implementation of the solution proposed in section 4. Section 5.1 details the process of extraction of Swift FSMM instances from the source code of analyzed programs. Section 5.2 describes how the mapping of Swift FSMM to CMM is implemented. Section 5.3 describes how the resulting implementation of Swift frontend was validated.

5.1 Extracting Swift FSMM Instances from Source Code

VizzAnalyzer accepts the source code of programs as input. Mapping from Swift FSMM to CMM has been defined in the previous section; the next step is to produce FSMM instances from the source code. This section describes the ways to achieve this, each subsection discussing one of the possible approaches.

5.1.1 Parsing the AST Dump Produced by Swift Compiler

The Swift compiler has an option which instructs it to generate a dump of AST of the compiled program: -dump-ast. The generated dump uses S-expression-like notation (S-expressions, short for "symbolic expressions," are a notation used for representing tree-structured data). The grammar describing S-expressions is simple (it consists of just four parser and seven lexer rules).

Consider a trivial Swift program in Figure 16. Provided that Swift compiler and tools are installed, AST dump for this program can be generated by invoking the compiler (swiftc) with the following command: swiftc trivial.swift -dump-ast. Dump produced for the program from Figure 16 is given in Figure 17.

```
// trivial.swift
print("Hello world!")
```

*Figure 16* A trivial Swift program serving as input for AST dump generation

```
(source_file
 (top_level_code_decl
 (brace_stmt
 (call_expr type='()' location=trivial.swift:2:1 range=[trivial.swift:2:1 -
 line:2:21] nothrow arg_labels=_:
 (declspec_expr type='(Any..., String, String) -> ()'
 location=trivial.swift:2:1 range=[trivial.swift:2:1 - line:2:1]
 decl=Swift.(file).print(_:separator:terminator:) function_ref=single)
 (tuple_shuffle_expr implicit type='(Any..., separator: String, terminator:
 String)' location=trivial.swift:2:7 range=[trivial.swift:2:6 - line:2:21]
 source_is_scalar elements=[-2, -1, -1] variadic_sources=[0]
 default_args_owner=Swift.(file).print(_:separator:terminator:)
 (paren_expr type='Any' location=trivial.swift:2:7
 range=[trivial.swift:2:6 - line:2:21]
 (erasure_expr implicit type='Any' location=trivial.swift:2:7
 range=[trivial.swift:2:7 - line:2:7]
 (string_literal_expr type='String' location=trivial.swift:2:7
 range=[trivial.swift:2:7 - line:2:7] encoding=utf8 value="Hello world!"
 builtin_initializer=Swift.(file).String.init(_builtinStringLiteral:utf8CodeUnitCou
 nt::isASCII:) initializer=**NULL**))))))))
```

*Figure 17* AST dump for the trivial Swift program

Expressions enclosed in parentheses are tree nodes with their children, each expression a=b is a node attribute (where a is attribute name, b is its value). Unfortunately, as the above example shows, the generated AST dump contains a significant number of irregularities even for trivial programs. For example, strings can be enclosed in either single or double quotes (type='()' and value="Hello world!"), attribute values can take many different forms (as in type='()',
5.1.2 Parsing Source Code Using ANTLR Parser

An alternative approach is to parse the source code of programs using an ANTLR parser. Swift grammar is documented (the list of production rules is available on Apple’s website, see [32]), but quite complex (559 rules for lexer and parser in total), therefore reimplementing Swift parser is a complex and time-consuming task. One existing implementation has been found (see [33]), but, according to documentation, it is not completely compatible with Swift compiler. Besides that, it only supports the older version of Swift (3.0.1, the latest version as of March 2018 is 4.1), and has certain limitations (specifically, some cases of ambiguous code are parsed incorrectly, see examples in [33]).

While all of the above issues can be solved, this approach has another problem. According to Swift compiler architecture (see [34]), parsing of the source code is just the first stage of the compilation process. AST produced after this stage lacks type information (which is added at later stages). Parser implementation from [33] corresponds to the first stage of the Swift compiler. Therefore AST it produces does not contain type information either. CMM contains certain types of relations which are deduced based on type information (for example, “type implements interface” or “method accesses field”), therefore if extraction of Swift FSMM were to rely on parser implementation from [33], type information would have to be added manually. As evident from the section 3.1, the type system of Swift is quite intricate. Therefore resolution of types would require an amount of effort significant enough to be out of the scope of this work.

Considering all above mentioned, implementing extraction of Swift FSMM by parsing Swift code using a custom parser is not feasible within the scope of this project, so this approach is not considered further.

5.1.3 Using SourceKit

SourceKit is a framework used by Xcode (an IDE for Swift, Objective-C, and C developed by Apple) for source code parsing, syntax highlighting, navigation, autocompletion. It is a standalone service, and Xcode communicates with it by forming special requests and getting the required information as a response.

The list of available SourceKit request types includes:

- **Code completion** (suggestions of possible ways to complete code at a specific location).
- **Indexing** (information about the kinds of source code within various text ranges of text in a source file).
- **Documentation** (symbols in source file and documentation attached to them using comments).
- **Mangling and demangling** (i.e., converting between human-readable and machine-readable names of symbols).
- **Cursor information** (information about a symbol at a particular location in the source file).
• **Structure** (information about the source file necessary for a text editor, such as mapping of text ranges to types of syntax, as well as the structure of the file).

This list is not exhaustive, but indexing and structure requests are of interest. The index request contains information about dependencies, symbol declarations in the file (free variables and functions, types and their members), their containment, references to other declarations, and, most importantly, USRs of the referred symbols. USR (Unique Symbol Resolution) is a string which can be used to identify each symbol uniquely.

Consider slightly more complex Swift program in Figure 18. The result of index request for this program is given in Figure 19.

```swift
func double(_ num: Int) -> Int {
    return num * 2
}
print(double(13))
```

**Figure 18** A Swift program serving as input for structure and index requests

```json
{
  "key.entities" : [{
    "key.line" : 1,
    "key.name" : "double(_:)",
    "key.usr" : "s:5basic6doubleS2iF",
    "key.kind" : "source.lang.swift.decl.function.free",
    "key.entities" : [{
      "key.line" : 1,
      "key.name" : "Int",
      "key.usr" : "s:Si",
      "key.kind" : "source.lang.swift.ref.struct"
    }, {
      "key.line" : 2,
      "key.name" : "*(__:_:)",
      "key.usr" : "s:S1moiS2i_SitFZ",
      "key.kind" : "source.lang.swift.ref.function.method.static"
    }]
  }, {
    "key.line" : 4,
    "key.name" : "print(_:separator:terminator:)",
    "key.usr" : "s:ssprintySayypGd_SS9separatorSS10terminatortF",
    "key.kind" : "source.lang.swift.ref.function.free"
  }]
}, {
  "key.line" : 4,
  "key.name" : "double(_:)",
  "key.usr" : "s:5basic6doubleS2iF",
  "key.kind" : "source.lang.swift.ref.function.free"
},
"key.dependencies" : [{
  "key.name" : "Swift",
  "key.kind" : "source.lang.swift.import.module.swift",
  "key.is_system" : true
}]
```

**Figure 19** Result of index request for the program in Figure 18

Some required information is missing from the index request, in particular information about the control flow structures, method parameters, size and access level of declarations. This information can be retrieved using the structure request.
The result of structure request for the program from Figure 18 is given in Figure 20.

```json
{
  "key.diagnostics.stage": "source.diagnostics.stage.swift.parse",
  "key.offset": 0,
  "key.length": 70,
  "key.substructure": [{
    "key.accessibility": "source.lang.swift.accessibility.internal",
    "key.length": 51,
    "key.name": "double(_:)",
    "key.kind": "source.lang.swift.decl.function.free",
    "key.offset": 0,
    "key.substructure": [{
      "key.typeName": "Int",
      "key.length": 10,
      "key.name": "num",
      "key.kind": "source.lang.swift.decl.var.parameter",
      "key.offset": 12
    }]
  }, {
    "key.length": 17,
    "key.name": "print",
    "key.kind": "source.lang.swift.expr.call",
    "key.offset": 52,
    "key.substructure": [{
      "key.length": 10,
      "key.name": "double",
      "key.kind": "source.lang.swift.expr.call",
      "key.offset": 58
    }]
  }]
}
```

Figure 20 An example of structure request result

Combination of output of the index and structure requests is sufficient for creating FSMM instances. The output of index request contains USR, line, name and kind of each declaration and reference, while the output of structure command contains offset from the beginning of file, length, name and kind.

The best way to combine the outputs of the structure and index requests would be to use the USRs of symbols which are guaranteed to identify symbols unambiguously. Unfortunately, the result of the structure request does not include the USRs of the program symbols, which makes the direct approach impossible. Less optimal, but still functional approach is to rely on positions of the symbols in the containing source file. For index requests, the position of each symbol is available as the number of the line on which its declaration starts, for symbols in structure request this information is expressed as the offset in characters from the start of the source file. While these two formats are not directly compatible, offsets in characters can be converted to line numbers using offsets of line ends calculating which is a trivial task. Some required information is not available in any of the SourceKit requests, namely the size of symbols in lines of code. However, this information can easily be calculated based on the start and end offsets of symbols and offsets of the line ends.

An algorithm which takes all of the above into consideration is detailed in Figure 21. Function analyzeProject is the entry point invoked from the command line. It accepts the path to the file which contains information about the source files, dependencies, and configuration of the analyzed project. The first step is to determine the character offsets of the start of each line. These offsets are used later to calculate the line locations and line count of program symbols. After calculating the line start offsets for a file, the algorithm issues index and structure requests to SourceKit and merges their results.
// Entry point of the utility, called from command line
function analyzeProject(pathToProjectFile) do
    let sourceFiles, dependencies = extractInfoFromProject(pathToProjectFile)
    let mergedResults
    for each file in sourceFiles do
        let lineStartOffsets
        for each line in file do
            lineStartOffsets.add(offset of line start)
        end
        let index = indexRequest(file, sourceFiles, dependencies)
        let structure = structureRequest(file, sourceFiles, dependencies)
        let mergedResult = merge(index, structure, lineStartOffsets)
        mergedResults.add(mergedResult)
    end
    return mergedResults
end

// Merges structure and index info for an entity
function merge(index, structure, lineStartOffsets) do
    let structureLineNumber = calculateLineNumber(structure, lineStartOffsets)
    let nestedIndices = index.valueForKey("nestedEntities")
    let nestedStructures = structure.valueForKey("nestedEntities")
    let nestedEntities
    for each nestedStructure in nestedStructures do
        let lineNumber = calculateLineNumber(nestedStructure, lineStartOffsets)
        let matchingIndex = find in nestedIndices
        where nestedIndex.valueForKey("lineNumber") == lineNumber
        // recursive call
        let nestedEntity = merge(matchingIndex, nestedStructure, lineStartOffsets)
        nestedEntities.add(nestedEntity)
    end
    let mergedResult
    copyAllKeysAndValues(index, mergedResult)
    copyAllKeysAndValues(structure, mergedResult)
    mergedResult.setValueForKey("lineNumber", structureLineNumber)
    mergedResult.setValueForKey("nestedEntities", nestedEntities)
    return mergedResult
end

// Calculates line number for an entity based on its start offset and offsets of line starts in the containing source file
function calculateLineNumber(structure, lineStartOffsets) do
    let structureStartOffset = structure.valueForKey("startOffset")
    for each lineStartOffset, index in lineStartOffsets do
        if lineStartOffset > structureStartOffset then
            return index
        end
    end
end

// adds all keys and values from object to receiver
function copyAllKeysAndValues(object, receiver) do
    for each key, value in object do
        receiver.setValueForKey(key, value)
    end
end

Figure 21 Algorithm of extraction of program model from source code

Merging of the index and structure information for an entity (source file, program symbol, control flow structure) is performed by the merge function. The first step of the merge process is to calculate the line number of the entity using the calculateLineNumber function. Then the algorithm iterates the structure information for the nested entities, calculating the line number for each of them using the same function. After that, it finds the matching index information by
iterating the index information for the nested entities and finding the entity which located on the same line. Once the pair of matching structure and index information is found, the merge function is invoked recursively for it. Then all structure and index information is copied into a single dictionary by calling the copyAllKeysAndValues function. After that, the line number and merged structure and index information for the nested entities are added. The result is then returned.

Function calculateLineNumber determines the line number of an entity based on the character offset of its start. It does this by iterating over the offsets of line starts and finding the first one which is greater than the start offset of the entity.

Function copyAllKeysAndValues accepts two dictionaries, iterating the key and value pairs of the first one and setting them in the second one. If the receiver already contains a key, the corresponding value is overwritten.

Figure 22 contains an example of the combined output produced by this algorithm. In this example information about the double(_:) method from the index request output (name, USR, line, type and method references) was augmented with information from the structure request (parameters, access level).

```json
{
  "key.moduleName": "basic",
  "key.sourceFiles": [
    {
      "key.hash": "2LV46HIQEGRB",
      "key.filepath": "basic/Sources/basic/main.swift",
      "key.dependencies": [
        {
          "key.hash": "1GTC8L0RGFA8Z",
          "key.name": "Swift",
          "key.kind": "source.lang.swift.import.module.swift",
          "key.is_system": true
        }
      ],
      "key.entities": [
        {
          "key.accessibility": "source.lang.swift.accessibility.internal",
          "key.line": 1,
          "key.lineCount": 2,
          "key.name": "double(_:)",
          "key.usr": "s:5basic6doubleS2iF",
          "key.kind": "source.lang.swift.decl.function.free",
          "key.entities": [
            {
              "key.typename": "Int",
              "key.line": 1,
              "key.name": "num",
              "key.kind": "source.lang.swift.decl.var.parameter"
            },
            {
              "key.line": 1,
              "key.name": "Int",
              "key.usr": "s:Si",
              "key.kind": "source.lang.swift.ref.struct"
            },
            {
              "key.line": 2,
              "key.name": "*(_:_:)",
              "key.usr": "s:Si moiSi2i_SitFZ",
              "key.kind": "source.lang.swift.ref.function.method.static"
            }
          ]
        },
        {
          "key.line": 4,
          "key.name": "print(_:separator:terminator:)",
          "key.usr": "s:5printySayypGd_SS9separatorSS10terminatortF",
          "key.kind": "source.lang.swift.ref.function.free"
        },
        {
          "key.line": 4,
          "key.name": "double(_:)",
          "key.usr": "s:5basic6doubleS2iF",
          "key.kind": "source.lang.swift.ref.function.free"
        }
      ]
    }
  ]
}
```

*Figure 22* An example of combined result of index and structure requests for the program in Figure 18 produced with algorithm in Figure 21
The files containing the information about the analyzed project are the *\.xcodeproj files. These files can be either created by the Xcode IDE or generated by the Swift package manager from the package definition (the Package.swift file) by executing the following command: swift package generate-xcodeproj.

For the reason that querying SourceKit requires information available only in the *\.xcodeproj file, it makes sense to use technology which makes extracting this information more manageable. Even though the project file format is XML-based, it is quite complicated, so using existing libraries capable of extracting information from it is better than implementing the same logic from scratch. Xcode.swift (see [35]) is an example of such library.

SourceKit is implemented as a separate process which provides C API for communication, so queries to SourceKit have to be written in C. In this case, manual memory management is required, as well as a way of communicating with the earlier mentioned Xcode.swift library which is written in Swift. While both issues can be solved, communication with SourceKit can be implemented in a more straightforward way using the SourceKitten library (see [36]). This framework wraps the C API of SourceKit and provides an equivalent API in Swift.

After choosing the libraries for extracting project information and communicating with SourceKit, a Swift command line utility was developed which implements the extraction algorithm described in Figure 21.

5.1.4 Creating Swift FSMM from the Output of the Extraction Utility

Once the model of the analyzed program is extracted using the extraction utility described in section 5.1.3, the next step is to transform it into Swift FSMM. The process of transformation is straightforward. The output produced by the extraction utility is in JSON format, entities of the program (such as source files, imports, declarations, references) are expressed as JSON objects. The root object corresponds to the program, child entities of each entity are expressed as an array of JSON objects stored in the corresponding JSON object. FSMM graph of the program can be produced by traversing this tree-like structure depth-first and creating graph nodes corresponding to its elements and edges expressing containment relations between them. Figure 23 describes the algorithm of conversion of the extracted program model to Swift FSMM.

```swift
function traverse(entity) do
    // create a node in FSMM graph of type
    // corresponding to the kind of entity
    let entityNode = createNode(entity)
    for each nestedEntity in entity.nestedEntities do
        // recursive call
        let childNode = traverse(nestedEntity)
        // create an edge in FSMM graph
        // expressing the containment relation
        createEdge(entityNode, childNode)
    end
    return entityNode
end
```

Figure 23 Algorithm of conversion of program model JSON into Swift FSMM

5.2 Conversion of Swift FSMM to CMM

This section explains the implementation of conversion of Swift FSMM instances to instances of CMM. Most of the entities (nodes, edges, and attributes) in FSMM
are copied to the CMM as is. According to the mapping defined in the section 4.3.2, some node and edge types may change. Other entities may require special processing:

- **Extension** nodes are processed differently depending on whether the extended type is a protocol or not. In case it is, extensions are mapped to *PartialClass*, otherwise, all properties and methods defined in the extension are copied to the extended type.
- **TypeAlias** nodes are not included in the CMM, all references to them are replaced by references to the aliased type (or types in case the aliased type is compound).
- **CallRef** nodes are mapped to *Loop* in case the called function is one of the *forEach, filter, map, flatMap, reduce* functions defined in the standard *Sequence* type.
- References to symbols defined in other modules (standard library or third-party frameworks) are removed.
- Both *class* and *static* keywords which denote type members in Swift FSMM correspond to *static* in CMM.
- *open, public* and *internal* access levels in Swift FSMM correspond to *public* in CMM, *fileprivate* to *package-local*, *private* to *private*.

The conversion is performed in two stages. Each stage is a depth-first traversal which follows the containment relations between nodes. During the first stage all symbol declarations are collected into a map by their USRs so that they can be easily looked up. During the second stage, the transformations described above are applied. Appendix 1 describes the conversion algorithm in more detail.

### 5.3 Verification of Swift Frontend for VizzAnalyzer

Before the Swift frontend can be considered complete, it has to be verified. Verification can be done by running the frontend against real-world Swift projects and checking the resulting CMM. Validation includes checking that all CMM constraints are satisfied, i.e., that it contains only entities from the set $T$, all relations between the entities conform to rules defined in the rule sets $P$ (containment relations) and $R$ (other relations).

However, these checks are not sufficient to make sure that the resulting CMM can be used successfully by all analyses. For instance, one of the analyses in VizzAnalyzer calculates the depth of inheritance tree metric for the program. This analysis builds the inheritance tree of the program by filtering the type nodes (*ClassType, InterfaceType*) and inheritance relations between them (*ExtendsRef, ImplementsRef*). After that, this analysis calculates the depth of the inheritance tree by finding the longest path in it. As stated in section 3.1, protocol extensions in Swift enable trait inheritance, which is a form of multiple inheritance. The formal definition of the CMM does not specify any constraints on the number of parent classes from which a class can inherit. While finding the longest path in an inheritance tree with classes inheriting from multiple parents is technically possible, it is unclear whether the result remains consistent with the original definition of the depth of inheritance tree metric.

Another aspect of the CMM which cannot be formally validated at the moment is the attributes of the nodes and edges. To enable this, the CMM has to be expanded with a formal definition of valid attributes for each node and edge type.

Performing the complete validation of the CMM produced by the Swift frontend will require examining every analysis in the VizzAnalyzer to make sure it
can be applied and possibly extending the definition of CMM with additional rules that ensure its applicability. This requires a significant amount of work which is out of scope of this project.

Instead, a different strategy was used for verification of Swift frontend. The strategy was to ensure that the basic building blocks of Swift programs are handled correctly. To do this, the Swift frontend was used to build CMM models of simple programs that use these basic blocks. The resulting CMM graphs are simple enough to be examined manually. Given that CMM graphs for simple programs are generated correctly, an assumption can be made that CMM for other programs is correct as well.

In addition to this, a set of heuristics can be applied to the count of graph nodes of specific types. A CMM graph contains nodes of the following types:

- **file nodes** (correspond to source files in the project);
- **type nodes** (classes, interfaces, and enumerations);
- **member nodes** (fields, methods, constructors, and enumeration cases);
- **reference nodes** (accesses, method calls, library and type references);
- **inheritance nodes** (describe the “extends”, “implements”, “overrides” relationships between types);
- **control flow structure nodes** (if, switch statements and their branches, as well as loop statements).

The following node count heuristics describing non-trivial Swift programs can be defined based on the author’s Swift programming experience:

- **file nodes**: number of files is more than 5;
- **type nodes**: there are at least as many type nodes as file nodes;
- **member nodes**: there are at least as many member nodes as type nodes;
- **reference nodes**: the number of reference nodes is greater than 0;
- **inheritance nodes**: there cannot be more inheritance nodes than type and member nodes combined;
- **control flow nodes**: the number of control flow nodes is greater than 0.

The first program, given in Figure 24, demonstrates the modeling of a simple class and its members (stored and calculated properties, methods, initializers, deinitializers, subscript functions), as well as access and method call references. The resulting CMM graph for this program is shown in Figure 25. The graph contains 1 file, 1 type, 8 member, and 3 reference nodes.

```swift
class SimpleClass {
    var aStoredProperty: Int
    var aComputedProperty: Int {
        get { return aStoredProperty + 100 }
        set { aStoredProperty = newValue - 100 }
    }
    init() { aStoredProperty = 20 }
    deinit {
        print("Deinitializing...")
        someMethod()
    }
    func someMethod() { print("Just printing...") }
    subscript(multipliedBy number: Int) -> Int { return aComputedProperty * number }
}
```

*Figure 24 A test Swift program featuring a simple class*
The second program features an example of inheritance between classes and protocols, as well as class extensions. The program can be found in Figure 26, the resulting CMM graph for it is in Figure 27. The graph contains 3 file, 3 type, 4 member, and 3 inheritance nodes.

// P.swift
protocol P {
    func someMethod()
}

// A.swift
class A: P {
    func someMethod() {
        print("Inside someMethod()")
    }
}

// B.swift
class B: A {
    override func someMethod() {
        print("Inside overridden someMethod()")
    }
}

extension B {
    func someMethodInExtension() {
        print("Inside someMethodInExtension()")
    }
}

Figure 26 A test Swift program featuring inheritance and extensions
The third program demonstrates an example of usage of more advanced features of type system, such as enumerations, type aliases, protocol extensions and value types. The program can be found in Figure 28, the resulting CMM graph for it is in Figure 29. The graph contains 1 file, 4 type, 7 member, 1 inheritance, and 5 reference nodes.

```swift
enum E {
    case foo, bar
}

typealias ThisIsAlsoE = E

protocol P {
    func someMethod()
}

extension P {
    func someMethod() {
        print("Default implementation of someMethod()")
    }
    func additionalMethod(e: ThisIsAlsoE) {
        print("additionalMethod(e:)")
    }
}

struct S: P {
}

let s = S()

s.additionalMethod(e: .foo)
```

**Figure 28** A test Swift program demonstrating advanced type system features
Another program features various control flow structures, such as guard, if, switch, for, while, and repeat…while statements, as well as an example of a loop statement expressed as a call to the forEach function defined in the Swift standard library. The program can be found in Figure 30, the resulting CMM graph for it is in Figure 31. The graph contains 1 file, 1 reference, and 24 control flow nodes.

```swift
import Foundation

var x = Int(arc4random_uniform(100))
guard x >= 0 else {
  fatalError("this should not have happened")
}
if x % 2 == 0 {
  print("even")
  if x % 5 == 0 {
    print("divisible by 5")
    if x % 10 != 0 {
      fatalError("this should never be executed")
    }
  }
}
if x % 3 == 0 {
  print("divisible by 3")
} else if x % 4 == 0 {
  print("divisible by 4")
} else if x % 5 == 0 {
  print("divisible by 5")
} else {
  print("not divisible by 3, 4 or 5")
}
while x < 4 {
  print("while loop")
  x *= 1
}
repeat {
  print("repeat while!")
  x -= 1
} while x >= 0
switch x % 3 {
  case 0: print("0")
  case let x where x > 10: print(">10")
  default: print("something else: \(x)")
}
for i in 0...5 {
  print("for loop: \(i)")
}
(0...5).forEach { i in
  print("for loop: \(i)")
}
```

Figure 30 A test Swift program featuring control flow structures
All of the above CMM graphs are consistent with the formal CMM definition given in section 2.3.4 and can be considered correct. Thus, the Swift frontend has been successfully tested with the programs using the basic building blocks of the Swift programming language.

In addition to the above simple programs, the Swift frontend has also been tested with 23 Swift projects of varying size, popularity and with a different number of contributors found on GitHub. Project size is indicated by the number of lines of source code; project popularity can be measured by the number of stars (for comparison, the top 10 most popular repositories with code on GitHub according to [37] have ~60-120 thousand stars). The list of additional projects used to verify the Swift frontend can be found in Appendix 2.

CMM graphs for the additional Swift projects has been generated successfully. Due to the size of the resulting graphs, it is impractical to include them in this report. Since the analyzed projects use the same building blocks as the simpler programs tested before, it was assumed that the generated models of the additional projects are correct as well. The heuristics defined earlier in this section have been applied as well, no violations were detected. Node count information can be found in Appendix 3.
6 Results and Evaluation

This chapter contains the summary of the results of this thesis project and evaluation of these results.

6.1 Results

Review of the current state of Swift ecosystem revealed lack of program comprehension tools for Swift. An attempt has been made to rectify this problem by adding support for Swift to VizzAnalyzer, an existing program analysis framework supporting Java and C# and featuring a variety of analyses (metrics calculations, clustering) and visualization.

Support for Swift has been added by defining a frontend-specific metamodel (FSMM) for Swift programs and mapping of this model to the common metamodel (CMM) used by VizzAnalyzer as the basis for the analyses and visualizations it features.

The core features of Swift such as modules, classes, protocols, enumerations, structs, variables, functions, initializers, properties, static and class members, control flow statements could be mapped to CMM directly. Property getters, setters, and observers, as well as deinitializers, subscripts were considered functions with special semantics and were mapped as such. Tuples were deemed equivalent to anonymous classes. Protocol extensions were expressed as partial classes. Extensions of non-protocol types were handled by moving all declarations inside them to the extended class. Source files were expressed as packages.

Some Swift features could not be expressed with CMM. These features include enumeration cases with associated values, optionality, the immutability of value types.

After the mapping between Swift FSMM and CMM was defined, a tool capable of extracting the CMM from Swift programs has been designed and implemented. Several different approaches to extracting the FSMM models from Swift programs have been tried: parsing the AST dump, parsing the source code, using the SourceKit framework. The last of these approaches was deemed the most appropriate and was implemented in a command line utility which was used by the added VizzAnalyzer frontend.

The use of the newly added Swift frontend was demonstrated on four simple Swift programs covering the considered Swift features. The resulting CMM was proven to be correct by checking its conformance to the formal CMM definition. In addition to that, the Swift frontend was tested using several open-source Swift projects of varying scale (between ~1000 and ~50 000 lines of code) and purpose (libraries, development tools, various applications). In all cases, the CMM was generated successfully.

6.2 Evaluation

The proposed way of expressing Swift programs using CMM covers the most used, core features of the language. However, due to significant differences between Swift and other programming languages supported by VizzAnalyzer and thus defining the features of CMM (Java and C#), the direct mapping could not be achieved.

Some simplifications had to be made to be able to express certain language features such as extensions. In case of non-protocol type extensions, copying the declarations in an extension to the extended type achieves the intended goal
functionality to existing types). However, since extensions can be declared in other files (or even other modules), the contained declarations may have access to a different set of types, variables, and functions, which may affect the object-oriented design of programs in a way that needs to be studied closer. As for the protocol extensions, even though they resemble trait inheritance, this fact is not mentioned anywhere in the language documentation. This might mean that there are subtle differences which may also affect the object-oriented design characteristics of programs.

Swift features like optionality and immutability could not be expressed in CMM due to lack of corresponding concepts. This means that characteristics of Swift programs affected by these features (namely, null safety and thread safety) cannot be analyzed using VizzAnalyzer. Although VizzAnalyzer does not currently have any analyses that address thread and null safety, this means that CMM has to be expanded with the required concepts before such analyses can be added.

Some of the features of Swift which were expressed using CMM might require further work. For example, statements and non-type declarations (variables, functions) in the global scope are consistent with the formal definition of CMM, but, because none of the languages currently supported by VizzAnalyzer has this feature, it is unclear how it affects the existing analyses. These statements and declarations might be ignored, which affects the correctness of the analysis results.

The proposed algorithm of extracting Swift FSMM from the source code is sufficient, but not optimal, as it requires merging the outputs of two SourceKit requests based on the location of the program entities in the source code. An algorithm that identifies the program entities unambiguously by their USR identifiers would be optimal but would require modifying the Swift compiler, which requires a significant amount of work.

Even though the evaluation focused on the problems and shortcomings of the proposed solution, the core features of Swift could be expressed successfully using VizzAnalyzer. This means that the proposed solution can be used to analyze real-world Swift programs and produce useful results; therefore this project can be considered at least partially successful.
7 Conclusion and Future Work

7.1 Conclusion

Program comprehension tools are essential for successful maintenance of any software, even more so in case of large-scale programs. However, for the reason that Swift development is not yet complete most of the development team's effort is directed towards completing the language, which means that other parts of the ecosystem do not receive due attention.

Lack of advanced comprehension tools may seem an obstacle to adoption of Swift. This project aims to fill this gap by enabling support for Swift in VizzAnalyzer framework which includes a set of program analyses and visualizations.

This, however, is not the most important contribution. Most importantly, this project enables a more in-depth study of Swift, a programming language which is claimed to combine the best ideas from many other programming languages with several innovations. Program comprehension tools make it possible to evaluate this claim from the scientific point of view by applying the same set of well-defined, rigorous analyses to programs in Swift and other programming languages, as well as studying the impact of the innovative features on the quality of programs. This project makes the first important step towards this goal.

7.2 Future Work

Weaknesses of the proposed solution discussed in section 6.2 hint at the potential directions for future work:

- Expand CMM so that it can express all features of Swift such as optionality and mutability. Besides supporting these features on the CMM level, it is also useful to understand how they can be used in existing and new analyses and visualizations.
- Verify the analyses and visualizations in the VizzAnalyzer framework to ensure the correctness of their results with the CMM models produced by the Swift frontend.
- Define a set of validity criteria for CMM instances generated by language frontends and create an automated test suite based on it.
- Propose better and more reliable way of extracting FSMM from the source code, possibly by modifying the Swift compiler or SourceKit.

Some other potentially interesting directions for future work include:

- Using VizzAnalyzer and its set of analyses to compare Swift with other programming languages (adding support for them if needed).
- Determine how particular Swift features impact the quality attributes of software by applying the same set of analyses to programs which do and do not make use of these features.
References


Appendix 1. The Algorithm of Conversion of Swift FSMM to CMM

```swift
function convertToCMM(graph) do
    let declarationNodesByUSR
    findDeclarationNodes(graph, graph.rootNode, declarationNodesByUSR)
    transformGraph(graph, graph.rootNode, declarationNodesByUSR)
end

// traverses the FSMM graph starting from node
// and collects all declaration nodes into declarationNodesByUSR
function findDeclarationNodes(graph, node, declarationNodesByUSR) do
    if node.usr is null then
        declarationNodesByUSR.setValueForKey(node.usr, node)
    end
    for each childNode in getChildrenNodes(graph, node) do
        // recursive call
        findDeclarationNodes(graph, childNode, declarationNodesByUSR)
    end
end

// traverses the FSMM graph starting from node
// and transforms it into a CMM graph
function transformGraph(graph, node, declarationNodesByUSR) do
    if node.type in [CallRef, AccessRef, TypeRef, MemberRef, ExtendsRef, ConformsRef, OverridesRef] and declarationNodesByUSR.valueForKey(node.targetUSR) is null then
        deleteNode(graph, node)
        return
    end
    if node.type is Extension then
        let extendedTypeNode = declarationNodesByUSR.valueForKey(node.targetUSR)
        if extendedTypeNode.type is Protocol then
            node.type = PartialClass
        else
            for each childNode in getChildrenNodes(graph, node) do
                addChildNode(graph, extendedTypeNode, childNode)
            end
            deleteNode(graph, node)
            return
        end
    end
    if node.type is TypeAlias then
        let aliasedTypeNode = declarationNodesByUSR.valueForKey(node.targetUSR)
        for each edge in graph.getInEdges(node) where edge.type is not contains do
            edge.toNode = aliasedTypeNode
        end
        deleteNode(graph, node)
        return
    end
    transformNodeAttributes(node)
    for each edge in graph.getIncidentEdges(node) do
        transformEdgeAttributes(edge)
    end
    for each childNode in getChildrenNodes(graph, node) do
        // recursive call
        transformGraph(graph, childNode, declarationNodesByUSR)
    end
end

// returns the neighbors of the node in graph
// which are connected to it with "contains" edges
function getChildrenNodes(graph, node) do
    let childNodes
```
for edge in getOutEdges(graph, node)
where edge.type is contains
do
    childNodes.add(edge.toNode)
end

return childNodes
end

// creates a "contains" edge from node to childNode
function addChildNode(graph, node, childNode) do
    graph.createEdge(contains, node, childNode)
end

// deletes node and all incident edges from graph
function deleteNode(graph, node) do
    for each edge in graph.getIncidentEdges(node) do
        graph.deleteEdge(edge)
    end

graph.deleteNode(node)
end

// transforms the attributes of node
// by replacing the FSMM attributes with their CMM counterparts
function transformNodeAttributes(node) do
    node.type = mapNodeType(node.type)
    node.accessLevel = mapAccessLevel(node.accessLevel)
    if node.isClass then
        node.isStatic = true
    end
end

// transforms the attributes of edge
// by replacing the FSMM attributes with their CMM counterparts
function transformEdgeAttributes(edge) do
    edge.type = mapEdgeType(edge.type)
end
Appendix 2. Projects Used for Verification of the Swift Frontend

*Note:* Data in the table below (lines of code, number of stars and contributors) reflects the state of each project as of 7 February 2018. All repository URLs were accessed on the same date.

<table>
<thead>
<tr>
<th>Project name</th>
<th>Description, URL</th>
<th>LOC</th>
<th>Stars</th>
<th>Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Libraries and frameworks</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kitura</td>
<td>Web framework (including web server), developed by IBM <a href="https://github.com/IBM-Swift/Kitura">https://github.com/IBM-Swift/Kitura</a></td>
<td>6197</td>
<td>~6200</td>
<td>55</td>
</tr>
<tr>
<td>vapor</td>
<td>Web framework <a href="https://github.com/vapor/vapor">https://github.com/vapor/vapor</a></td>
<td>6597</td>
<td>~12400</td>
<td>89</td>
</tr>
<tr>
<td>swifter</td>
<td>Small HTTP server framework <a href="https://github.com/httpswift/swifter">https://github.com/httpswift/swifter</a></td>
<td>3060</td>
<td>~2000</td>
<td>49</td>
</tr>
<tr>
<td>Alamofire</td>
<td>HTTP networking library <a href="https://github.com/Alamofire/Alamofire">https://github.com/Alamofire/Alamofire</a></td>
<td>7243</td>
<td>~26700</td>
<td>150</td>
</tr>
<tr>
<td>CoreStore</td>
<td>Wrapper library for CoreData (built-in data persistence framework) <a href="https://github.com/JohnEstropia/CoreStore">https://github.com/JohnEstropia/CoreStore</a></td>
<td>31848</td>
<td>~1800</td>
<td>16</td>
</tr>
<tr>
<td>DVR</td>
<td>Framework for stubbing HTTP requests in tests <a href="https://github.com/venmo/DVR">https://github.com/venmo/DVR</a></td>
<td>765</td>
<td>~500</td>
<td>11</td>
</tr>
<tr>
<td>Kingfisher</td>
<td>Library for downloading and caching images from the web <a href="https://github.com/onevcat/Kingfisher">https://github.com/onevcat/Kingfisher</a></td>
<td>5894</td>
<td>~10800</td>
<td>82</td>
</tr>
<tr>
<td>LayoutKit</td>
<td>View layout library, developed by LinkedIn, Inc. <a href="https://github.com/linkedin/LayoutKit">https://github.com/linkedin/LayoutKit</a></td>
<td>3663</td>
<td>~2300</td>
<td>22</td>
</tr>
<tr>
<td>Quick</td>
<td>Behavior-driven development framework <a href="https://github.com/Quick/Quick">https://github.com/Quick/Quick</a></td>
<td>1603</td>
<td>~7200</td>
<td>107</td>
</tr>
<tr>
<td>RxSwift</td>
<td>Library providing support for reactive programming [<a href="https://github.com/">https://github.com/</a> ReactiveX/RxSwift](<a href="https://github.com/">https://github.com/</a> ReactiveX/RxSwift)</td>
<td>17278</td>
<td>~12200</td>
<td>226</td>
</tr>
<tr>
<td>SwiftSVG</td>
<td>Parser for SVG vector graphics format <a href="https://github.com/mchoe/SwiftSVG">https://github.com/mchoe/SwiftSVG</a></td>
<td>5029</td>
<td>~900</td>
<td>5</td>
</tr>
<tr>
<td><strong>Developer tools</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BuildTime Analyzer-for-Xcode</td>
<td>Build tool which integrates with Swift compiler to show the amount of time it took to compile each source file, function.</td>
<td>1112</td>
<td>~2200</td>
<td>9</td>
</tr>
<tr>
<td>App Name</td>
<td>Description</td>
<td>Repository Link</td>
<td>Repository Links</td>
<td>Stars</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>----------------</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td><strong>Carthage</strong></td>
<td>Decentralized dependency manager for macOS and iOS projects</td>
<td><a href="https://github.com/RobertGummesson/BuildTimeAnalyzer-for-Xcode">https://github.com/RobertGummesson/BuildTimeAnalyzer-for-Xcode</a></td>
<td><a href="https://github.com/Carthage/Carthage">https://github.com/Carthage/Carthage</a></td>
<td>1355</td>
</tr>
<tr>
<td><strong>Sourcery</strong></td>
<td>Tool for Swift code generation</td>
<td><a href="https://github.com/krzysztofzablocki/Sourcery">https://github.com/krzysztofzablocki/Sourcery</a></td>
<td></td>
<td>3753</td>
</tr>
<tr>
<td><strong>Applications</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>firefox-ios</strong></td>
<td>An iOS version of Firefox browser</td>
<td><a href="https://github.com/mozilla-mobile/firefox-ios">https://github.com/mozilla-mobile/firefox-ios</a></td>
<td></td>
<td>38526</td>
</tr>
<tr>
<td><strong>focus-ios</strong></td>
<td>An application which blocks advertisements and user tracking scripts in web browsers</td>
<td><a href="https://github.com/mozilla-mobile/focus-ios">https://github.com/mozilla-mobile/focus-ios</a></td>
<td></td>
<td>5676</td>
</tr>
<tr>
<td><strong>ello-ios</strong></td>
<td>Client application for Ello, a publishing and collaboration platform for artists</td>
<td><a href="https://github.com/ello/ello-ios">https://github.com/ello/ello-ios</a></td>
<td></td>
<td>54745</td>
</tr>
<tr>
<td><strong>eidolon</strong></td>
<td>Kiosk application for art auction company</td>
<td><a href="https://github.com/artsy/eidolon">https://github.com/artsy/eidolon</a></td>
<td></td>
<td>9536</td>
</tr>
<tr>
<td><strong>WWDC</strong></td>
<td>Application for developer conference</td>
<td><a href="https://github.com/insidegui/WWDC">https://github.com/insidegui/WWDC</a></td>
<td></td>
<td>9289</td>
</tr>
<tr>
<td><strong>Antidote</strong></td>
<td>Client for secure messaging platform</td>
<td><a href="https://github.com/Antidote-Tox/Antidote">https://github.com/Antidote-Tox/Antidote</a></td>
<td></td>
<td>17619</td>
</tr>
<tr>
<td><strong>breadwallet-ios</strong></td>
<td>Bitcoin wallet application</td>
<td><a href="https://github.com/breadwallet/breadwallet-ios">https://github.com/breadwallet/breadwallet-ios</a></td>
<td></td>
<td>29684</td>
</tr>
<tr>
<td><strong>iina</strong></td>
<td>Video player for macOS</td>
<td><a href="https://github.com/lhc70000/iina">https://github.com/lhc70000/iina</a></td>
<td></td>
<td>22410</td>
</tr>
<tr>
<td><strong>swift-2048</strong></td>
<td>An implementation of &quot;2048&quot; game in Swift</td>
<td><a href="https://github.com/austinzheng/swift-2048">https://github.com/austinzheng/swift-2048</a></td>
<td></td>
<td>1178</td>
</tr>
</tbody>
</table>
### Appendix 3. Results of Verification of the Swift Frontend

<table>
<thead>
<tr>
<th>Project</th>
<th>Total nodes</th>
<th>File nodes</th>
<th>Type nodes</th>
<th>Member nodes</th>
<th>Reference nodes</th>
<th>Inheritance nodes</th>
<th>Control flow nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Libraries and frameworks</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kitura</td>
<td>1481</td>
<td>44</td>
<td>55</td>
<td>473</td>
<td>345</td>
<td>39</td>
<td>525</td>
</tr>
<tr>
<td>vapor</td>
<td>1587</td>
<td>97</td>
<td>83</td>
<td>563</td>
<td>266</td>
<td>35</td>
<td>543</td>
</tr>
<tr>
<td>swifter</td>
<td>1653</td>
<td>19</td>
<td>29</td>
<td>506</td>
<td>426</td>
<td>23</td>
<td>650</td>
</tr>
<tr>
<td>Alamofire</td>
<td>2279</td>
<td>17</td>
<td>72</td>
<td>580</td>
<td>749</td>
<td>129</td>
<td>732</td>
</tr>
<tr>
<td>CoreStore</td>
<td>6144</td>
<td>148</td>
<td>200</td>
<td>1969</td>
<td>1711</td>
<td>1023</td>
<td>1093</td>
</tr>
<tr>
<td>DVR</td>
<td>301</td>
<td>9</td>
<td>11</td>
<td>92</td>
<td>55</td>
<td>28</td>
<td>106</td>
</tr>
<tr>
<td>Kingfisher</td>
<td>1723</td>
<td>22</td>
<td>83</td>
<td>507</td>
<td>526</td>
<td>117</td>
<td>468</td>
</tr>
<tr>
<td>LayoutKit</td>
<td>1349</td>
<td>38</td>
<td>52</td>
<td>386</td>
<td>447</td>
<td>119</td>
<td>307</td>
</tr>
<tr>
<td>Quick</td>
<td>490</td>
<td>20</td>
<td>30</td>
<td>185</td>
<td>140</td>
<td>30</td>
<td>85</td>
</tr>
<tr>
<td>RxSwift</td>
<td>7184</td>
<td>148</td>
<td>617</td>
<td>2139</td>
<td>2055</td>
<td>1046</td>
<td>1179</td>
</tr>
<tr>
<td>SwiftSVG</td>
<td>1383</td>
<td>44</td>
<td>55</td>
<td>349</td>
<td>398</td>
<td>193</td>
<td>344</td>
</tr>
<tr>
<td><strong>Developer tools</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BuildTime Analyzer-for-Xcode</td>
<td>517</td>
<td>14</td>
<td>18</td>
<td>177</td>
<td>139</td>
<td>45</td>
<td>124</td>
</tr>
<tr>
<td>Carthage</td>
<td>410</td>
<td>14</td>
<td>22</td>
<td>158</td>
<td>142</td>
<td>7</td>
<td>67</td>
</tr>
<tr>
<td>Sourcery</td>
<td>1211</td>
<td>21</td>
<td>42</td>
<td>296</td>
<td>194</td>
<td>12</td>
<td>646</td>
</tr>
<tr>
<td><strong>Applications</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>firefox-ios</td>
<td>16381</td>
<td>171</td>
<td>524</td>
<td>5184</td>
<td>4747</td>
<td>1518</td>
<td>4237</td>
</tr>
<tr>
<td>focus-ios</td>
<td>2465</td>
<td>55</td>
<td>86</td>
<td>874</td>
<td>831</td>
<td>179</td>
<td>440</td>
</tr>
<tr>
<td>ello-ios</td>
<td>20956</td>
<td>489</td>
<td>811</td>
<td>6072</td>
<td>4373</td>
<td>2652</td>
<td>6559</td>
</tr>
<tr>
<td>eidolon</td>
<td>3659</td>
<td>108</td>
<td>145</td>
<td>1434</td>
<td>1078</td>
<td>353</td>
<td>541</td>
</tr>
<tr>
<td>WWDC</td>
<td>4254</td>
<td>72</td>
<td>98</td>
<td>1082</td>
<td>1747</td>
<td>306</td>
<td>949</td>
</tr>
<tr>
<td>Antidote</td>
<td>6657</td>
<td>165</td>
<td>294</td>
<td>2504</td>
<td>1448</td>
<td>797</td>
<td>1449</td>
</tr>
<tr>
<td>breadwallet-ios</td>
<td>12892</td>
<td>199</td>
<td>393</td>
<td>4001</td>
<td>4102</td>
<td>956</td>
<td>3241</td>
</tr>
<tr>
<td>iiina</td>
<td>11505</td>
<td>102</td>
<td>230</td>
<td>3736</td>
<td>3516</td>
<td>610</td>
<td>3311</td>
</tr>
<tr>
<td>swift-2048</td>
<td>444</td>
<td>9</td>
<td>30</td>
<td>138</td>
<td>89</td>
<td>40</td>
<td>138</td>
</tr>
</tbody>
</table>