
DEPARTMENT OF INFORMATICS

MENDELOVA UNIVERZITA V BRNĚ

Bachelor's Thesis

# Static Code Analysis Tool for the Apex Programming Language

Statický analyzátor jazyka Apex

**Author:** Belek Omurzakov

**Supervisor:** doc. Dr. Ing. Jiří Rybička

Brno 2024

● MENDELU
● Provozně
● ekonomická
● fakulta

Ústav informatiky
Akademický rok: 2023/2024

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

| | |
|---|---|
| Autor práce: | **Belek Omurzakov** |
| Studijní program: | Otevřená informatika |
| Zaměření: | Zaměření pro Otevřenou informatiku |
| Vedoucí práce: | doc. Dr. Ing. Jiří Rybička |
| Název práce: | **Statický analyzátor jazyka Apex** |
| Rozsah práce: | dle aktuální vyhlášky děkana |

Zásady pro vypracování:

1. Seznamte se s jazykem Apex firmy Salesforce a s typickými aplikacemi vyvíjenými v tomto jazyce. Seznamte se s principy statické analýzy zdrojových kódů a s již existujícími statickými analýzami jazyka Apex.
2. Navrhněte prvky statické analýzy vyplývající jak z kódu samotného jazyka, tak i z aplikačního využití. Zaměřte se na takové prvky, jejichž statická analýza není běžná (data flow) a které představují zefektivnění typických aplikací.
3. Implementujte prototyp statického analyzátoru s navrženými alespoň čtyřmi funkcemi, otestujte jej na reálných aplikacích a navrhněte vhodnou formu jeho integrace s existujícími vývojovými nástroji firmy Salesforce.

Seznam odborné literatury:


1. Aho, A., Ullman, J., Sethi, R., Lam, M. S. Compilers: Principles, Techniques, and Tools. 2nd. ed. Addison Wesley, 2006. ISBN 978-0321486813.
2. Cousot, P. Principles of Abstract Interpretation. MIT Press, 2021. ISBN 9780262044905.
3. Møller, A., Schwartzbach, M. Static Program Analysis [online]. Aarhus University, 2023. Dostupné na https://users-cs.au.dk/amoeller/spa/spa.pdf
4. Nielson, F., Nielson, H. R., Hankin, Ch. Principles of Program Analysis. Springer Berlin, Heidelberg, 1999. DOI https://doi.org/10.1007/978-3-662-03811-6

Datum zadání bakalářské práce: leden 2024

Termín odevzdání bakalářské práce: květen 2024

L. S.

Elektronicky schváleno dne 4. 1. 2024
**doc. Dr. Ing. Jiří Rybička**
Vedoucí práce

Elektronicky schváleno dne 4. 1. 2024
**prof. Ing. Cyril Klimeš, CSc.**
Vedoucí ústavu

Elektronicky schváleno dne 4. 1. 2024
**Ing. David Procházka, Ph.D.**
Garant studijního programu

Elektronicky schváleno dne 4. 1. 2024
**Belek Omurzakov**
Autor práce

# Acknowledgements

I would like to express my deepest gratitude to my thesis advisor, doc. Dr. Ing. Jiří Rybička, whose expertise, understanding, and patience, added considerably to my graduate experience. Your guidance helped me in all the time of research and writing of this thesis. I must acknowledge the support of my family, who provided me with moral and emotional encouragement throughout my study. Thank you for always believing in me. Finally, I would like to extend my thanks to my friends and everyone who supported me during the writing of this thesis.

# Declaration

I hereby declare that this thesis entitled **Static Code Analysis Tool for the Apex Programming Language** was written and completed by me. I also declare that all the sources and information used to complete the thesis are included in the list of references. I agree that the thesis could be made public in accordance with Article 47b of Act No. 111/1998 Coll., Higher Education Institutions and on Amendments and Supplements to Some Other Acts (the Higher Education Act), and in accordance with the current Directive on publishing of the final thesis. I am aware that my thesis is written in accordance to Act No. 121/2000 Coll., on Copyright, and therefore Mendel University in Brno has the right to conclude licence agreements on the utilization of the thesis as a school work in accordance with Article 60(1) of the Copyright Act. Before concluding a licence agreement on utilization of the work by another person, I will request a written statement from the university that the licence agreement is not in contradiction to the legitimate interests of the university, and I will also pay a prospective fee to cover the cost incurred in creating the work to the full amount of such costs.

In Brno on May 12, 2024 ----------

signature

# Abstract

This thesis presents the development of a static code analysis tool specifically designed for the Apex programming language. The study began with a detailed review of existing static analysis tools, which revealed limitations in their ability to perform complex data flow analysis for Apex. To address these shortcomings, a prototype tool was developed, starting by converting the source code's parse tree into an intermediate representation in the form of a control flow graph, and incorporating advanced static analysis features.

Designed as a command-line application, this tool integrates seamlessly into CI/CD pipelines, automating the analysis process and ensuring the consistent application of code quality checks throughout the development lifecycle. Results from the testing and evaluation phase indicate that the tool can effectively identify and mitigate issues, particularly in a subset of problems often overlooked by current analyzers. This work serves as a stepping stone towards developing a more comprehensive static analysis tool for the Apex programming language.



## Keywords

static analysis, Apex, Salesforce, data flow analysis, control flow graph, abstract interpretation, symbolic execution



# Abstrakt

Tato práce se zabývá vývojem softwarového nástroje pro statickou analýzu kódu, který je speciálně navržen pro programovací jazyk Apex. Úvodní část práce poskytuje detailní přehled již existujících řešení v této oblasti. Rozebrány jsou především nedostatky v provádění komplexní analýzy toku dat v jazyce. Převedením derivačního stromu zdrojového kódu do vnitřní reprezentace grafu toku řízení a vybavením pokročilými funkcemi statické analýzy byl vyvinut prototyp nástroje, který odstraňuje stávající nedostatky.

Vzniklý nástroj, navržený jako aplikace příkazového řádku, se bez problému integruje do CI/CD pipeline, automatizuje proces analýzy a zajišťuje důsledné uplatňování kontroly kvality kódu v průběhu celého vývojového cyklu. Z testovací fáze pak vyplývá, že nový nástroj je schopen účinně identifikovat zejména problémy, které současné analyzátory často přehlížejí a lze jej tedy považovat za významný krok směrem k účinnějším a efektivnějším nástrojům statické analýzy pro jazyk Apex.



## Klíčová slova

statická analýza, Apex, Salesforce, analýza toku dat, graf toku řízení, abstraktní interpretace, symbolické vykonávání

# Contents

# Chapter 1

# Introduction

## 1.1 Background and Motivation

In today's digital world, ensuring that software applications are reliable and secure is more important than ever. Software bugs can lead to a range of consequences, from minor disruptions to severe financial losses and damage to an organization's reputation.

A critical part of maintenance practices is code review—a systematic examination of program source code intended to find and fix mistakes overlooked during the initial development phase. Code reviews are vital because they not only improve the overall quality of software but also ensure adherence to coding standards and help identify potential security breaches before the software is deployed. Despite its importance, manual code reviews can be time-consuming and are subject to human error, particularly in complex systems with large codebases.

This is where static analysis tools become crucial. Static analysis involves examining the code without executing the program, allowing developers to detect errors or complex patterns that might not be obvious at first glance. By automating the detection of common problems early in the development process, these tools can save valuable human time and effort.

## 1.2 Problem Statement

The Salesforce platform, renowned for its robust CRM capabilities, extensively utilizes Apex, a programming language designed specifically for its ecosystem. Apex is crucial for implementing business logic and managing data transactions, making it fundamental to the platform's operation and customization. As businesses increasingly rely on Salesforce for critical operations, the demand for robust, secure, and efficient Apex applications has grown. Despite its significance, the ecosystem surrounding Apex development tools, particularly static analysis tools, is underdeveloped compared to those available for mainstream programming languages. Current static analysis offerings for Apex predominantly rely on basic pattern-matching techniques that fail to capture deeper, semantic errors, which can lead to critical vulnerabilities or performance issues. A few tools that support data and control flow analysis are either still in beta versions or support a limited number of analysis types.

## 1.3 Objectives and Scope

This thesis aims to design and implement a prototype static analysis tool tailored for Apex, addressing significant shortcomings identified in current static analysis practices for this language. The primary objective is to provide a proof of concept that introduces new types of analyses not currently supported by existing tools, such as queried field access analysis and detection of constant conditions, while enhancing the capabilities of existing analyses like nullness analysis and optimizations for governor limits in loops. By focusing on these specific areas, the prototype will:

1. Demonstrate the feasibility of incorporating sophisticated analysis techniques in Apex static analysis tools that go beyond basic pattern matching.
2. Offer potential pathways to mitigate critical vulnerabilities and optimize performance by detecting and resolving issues that current tools overlook.
3. Establish a foundation for further development and refinement of static analysis tools within the Apex development ecosystem.

This initiative will fill a crucial gap in the current toolset and set the stage for advanced developments in the static analysis of Apex. It could lead to safer and more efficient application development on the Salesforce platform.

## 1.4 Thesis Structure

This thesis is organized into six chapters. It begins with Chapter 2, where I review the current landscape of static code analysis tools, with a particular focus on those developed for Apex. In Chapter 3, I delve into the theoretical foundations essential for static analysis, introducing key methodologies and the specifics of the Apex programming language that are relevant to my study. Chapter 4 details the implementation of the static analysis prototype I developed, discussing the methods used for various types of analysis. Chapter 5 is dedicated to testing and evaluation, where I assessed the effectiveness of my tool through a series of carefully designed test cases. The thesis concludes with Chapter 6, where I summarize my findings, discuss the contributions my work makes to the field, and suggest directions for future work.

# Chapter 2

# Related Work

## 2.1 Overview of Static Code Analysis

One of the earliest and most influential static code analysis tools was Lint, written in 1978, for the C programming language by Stephen Johnson at Bell Labs. Lint, the tool, was designed to detect various types of programming errors, such as type mismatches, scope violations, and the use of uninitialized variables, which were not typically caught by compilers at the time [1].

After the introduction of Lint, the concept behind it sparked an array of similar tools tailored for other programming languages. Each was designed to capture language-specific nuances that could lead to bugs or inefficient code. For instance, the tool Splint, originally known as LCLint, extended Lint's ideas to provide more complex checks for C programs, including checks for memory leaks, use-after-free issues, and more sophisticated type checks beyond what Lint could handle [1].

During the 1990s and 2000s, as software development methodologies evolved and systems became more complex, the need for automated tools that could handle both large codebases and complex programming languages became more apparent. With the growth in processing power, a new generation of static analysis tools incorporated advanced techniques such as model checking, symbolic execution, and abstract interpretation. These techniques allowed for the analysis of potential execution paths, including those that might only manifest under very specific conditions, thus identifying hidden errors that could lead to failures in critical systems [1].

The development of integrated development environments (IDEs) in the 2000s also integrated static code analysis directly into the software development process. Modern IDEs, such as Eclipse, Visual Studio, and IntelliJ IDEA, typically include built-in static analysis tools that provide real-time feedback to developers as they write code, significantly reducing the time and effort required to identify and correct errors.

As we approached the 2010s, static analysis tools became even more sophisticated, incorporating machine learning techniques to predict problematic patterns based on vast amounts of code analysis, further refining the effectiveness and accuracy of static code analysis.

## 2.2 Previous Work on Static Code Analyzers for Apex

Static code analysis for Apex has been an area of active development, with several tools and methodologies proposed over the years. One of the earliest static code analyzers for Apex is PMD [2], an open-source cross-language static code analyzer. PMD includes a set of predefined

rules for Apex code, which can detect potential issues such as unused variables, empty catch blocks, and violations of naming conventions. PMD's rules can be customized and extended, allowing developers to tailor the tool to their specific needs. Some tools such as GearSet [3], Copado [4] and MegaLinter [5] have incorporated PMD's Apex engine into their analysis pipelines.

Another significant contribution to static code analysis for Apex is CodeScan [6], a commercial tool specifically designed for Salesforce development. CodeScan offers a comprehensive set of rules for Apex and integrates with popular continuous integration servers. It also provides a user-friendly interface for reviewing and managing detected issues.

Recently, Salesforce introduced the Salesforce Code Analyzer [7], a static code analysis tool that bundles some open-source code analyzing tools such as PMD, CPD, and the Salesforce Graph Engine. The latter leverages graph-based techniques to analyze Apex code and supports various types of data flow analysis, including `RemoveUnusedMethod`, `AvoidDatabaseOperationInLoop`, and `ApexNullPointerExceptionRule`.

In addition to these tools, several general-purpose static code analyzers support Apex, such as SonarSource [8], CheckMar [9] and Semgrep [10]. Semgrep uses a pattern-based approach to detect issues in the code but also supports more advanced analysis techniques such as constant propagation and taint analysis. Apex support in Semgrep is currently in beta, but the tool shows promise for detecting complex issues in Apex code.

These tools have made significant contributions to static code analysis for Apex, but there is still room for improvement.

## 2.3 Gap Analysis

Current tools predominantly utilize pattern-matching techniques, which scan the source code to identify patterns indicating potential issues. These tools define a set of such patterns that, when detected as present or absent, suggest possible code problems.

For example, a typical issue that can be identified is when a test method annotated with `@IsTest` lacks assert statements. This omission is critical; assert statements are essential for verifying that the code behaves as expected, thus ensuring the test's validity and reliability. Without them, the test method fails to fulfill its primary function.

Static analyzers parse the source code into an Abstract Syntax Tree (AST). As they traverse this tree, if they encounter a method marked with `@IsTest`, they specifically check for the inclusion of assert statements. The absence of such statements triggers the analyzer to flag the method as problematic. This approach is effective for detecting common coding errors and enforcing coding standards; however, it has several limitations.

First, it is not well-suited for detecting complex issues that require more sophisticated analysis. For example, consider a scenario where a method is called with a null argument, but the method itself does not check for nullity. This situation can lead to a `NullPointerException` at runtime. A pattern-matching analyzer would not be able to detect this issue because it lacks the capability to track data flow through the program.

Second, these tools often provide intra-procedural analysis, meaning they analyze each method in isolation. This approach is insufficient for detecting issues that span multiple methods. For example, common best practices in Salesforce development advise against performing resource-intensive operations within loops due to potential performance issues, such as hitting governor limits (discussed in Subsection 3.4.3). Most analyzers can detect this issue, but only if the resource-intensive operation and the loop are within the same method. If the resource-intensive operation is encapsulated in one method and the loop in another, the analyzer will not detect the issue.

Third, the effectiveness of these tools is inherently limited by the patterns they can recognize. If a pattern is not explicitly defined in the analyzer, it will not be detected. This limitation makes it difficult to identify resource-intensive operations in loops created by recursive methods, as the analyzer may not recognize the pattern.

The Salesforce Graph Engine offers significant improvements in detecting complex issues like null pointer exceptions, particularly when a variable is under a null check or is uninitialized. However, its capability to identify issues where the code might access potentially nullable references remains limited.

Furthermore, the analysis of queried field access and constant condition analysis—evaluating expressions within the code that always result in a true or false condition—is not addressed by current tools. This common oversight in Salesforce development—where developers access fields from database queries without ensuring these fields are present—can lead to runtime exceptions and compromises data integrity. Additionally, failing to identify constant conditions can result in inefficient code execution and missed optimization opportunities.

## 2.4 Proposed Approach

To address the limitations of existing tools, I propose the development of a prototype to showcase a proof of concept for enhanced static code analysis for Apex. The core concept behind this initiative is to apply more sophisticated analysis techniques, such as data flow analysis and symbolic execution. These techniques are designed to provide a deeper understanding of the code's behavior and help detect more complex issues that traditional pattern-matching techniques might miss.

This proposed approach will involve building a static code analyzer capable of performing inter-procedural analysis. This will allow for tracking data flow and control flow across multiple methods. Such capabilities will enable the detection of issues spanning multiple methods, for instance, resource-intensive operations within recursion that could affect performance. Additionally, it will facilitate the identification of complex issues related to code's control flow, such as null pointer dereferences and other potential runtime errors that could lead to system crashes or unexpected behavior, using approximation techniques with abstract interpretation.

The static code analyzer will be implemented as a command-line tool, making it easily integrable into existing development workflows. Developers will be able to run this tool on their codebase and receive a detailed report of detected issues. This approach not only aims to enhance current analysis capabilities but also introduces new methodologies to tackle complex coding challenges effectively.

# Chapter 3

# Theoretical Framework

## 3.1 Principles of Static Analysis

### 3.1.1 Abstract Interpretation

Soundness and completeness are two fundamental principles of program analysis. Soundness refers to the principle that a static analysis tool must report all true instances of a specified property, such as an error or a vulnerability, within the codebase. A sound analysis does not miss real defects, but it may also report false positives—instances where the tool predicts defects that do not exist. The importance of soundness lies in its ability to detect all true errors, thereby significantly improving code reliability and security [11].

Completeness, on the other hand, ensures that every warning or error reported by the analysis tool corresponds to an actual defect. An accurate tool eliminates false positives, thereby ensuring that developers do not waste time investigating non-existent issues. However, achieving complete accuracy in static analysis is a significant challenge due to the undecidability of many properties in general programming languages [12, 13]. This undecidability implies that for certain properties of programs, no algorithm can exist that always leads to a correct yes-or-no answer within finite time, thus impacting the feasibility of achieving both soundness and completeness in practice [12, 13]. Such an undecidability requires a trade-off between soundness, completeness, and termination, leading tools to sometimes approximate results rather than provide definitive conclusions.

To achieve a balance between soundness and completeness, static analysis tools often use approximation. This involves making educated guesses about the program's behavior based on the available code information. However, this method may result in false positives (over-approximation) or false negatives (under-approximation), depending on the tool's approach. The goal is to strike a balance that maximizes utility while minimizing the overhead of addressing false alarms.

The framework used to perform approximations is *abstract interpretation*, introduced by Cousot and Cousot in the 1970s [14]. This formal framework is used to perform approximations in program analysis. It simplifies the complexities of program behaviors into abstract domains that are easier to analyze while conservatively approximating the multitude of possible program states. Through abstract interpretation, static analysis can infer properties about infinite state spaces in a computationally feasible manner, thereby enabling the detection of potential errors in a wide variety of contexts [15].

### 3.1.2 Lattice Theory

A *lattice* is a partially ordered set (poset) $(L, \leq)$ in which every two elements $a, b \in L$ have a unique least upper bound, denoted as $a \vee b$ (join), and a unique greatest lower bound, denoted as $a \wedge b$ (meet). Formally, the set $L$ is a lattice if, for every $a, b \in L$, the following conditions hold:

- The element $a \vee b$ is the smallest element in $L$ such that $a \leq a \vee b$ and $b \leq a \vee b$.
- The element $a \wedge b$ is the largest element in $L$ such that $a \wedge b \leq a$ and $a \wedge b \leq b$.

*A complete lattice* further extends these properties to every subset $S \subseteq L$, ensuring that every such subset has both a join $\bigvee S$ and a meet $\bigwedge S$ [16].

Abstract interpretation relies heavily on concepts from lattice theory. Lattices provide a way to represent sets of possible values instead of single concrete values, given that real-world program values can be complex. A poset allows reasoning about the "precision" of abstract values by creating a hierarchy where some values are more specific than others. Moreover, as programs often involve branching, lattices provide operations to combine information from different parts of a program. This facilitates abstract interpretation to reason about the combined possibilities arising from different execution paths [14].

## 3.2 Static Analysis Techniques and Tools

### 3.2.1 Data Flow Analysis

Data Flow Analysis (DFA) is a static analysis technique used in compiler optimization and program verification to gather information about the possible values and states of variables throughout a computer program [17]. The analysis is performed by constructing a Control Flow Graph (CFG) that outlines the program's execution paths and then propagating data flow information through the graph.

A CFG is defined as a directed graph where each node represents a basic block—a sequence of consecutive statements or instructions that has one entry point and one exit point. The edges between these nodes illustrate the control flow paths, meaning they show the possible transitions from one block to another during the execution of the program [18].

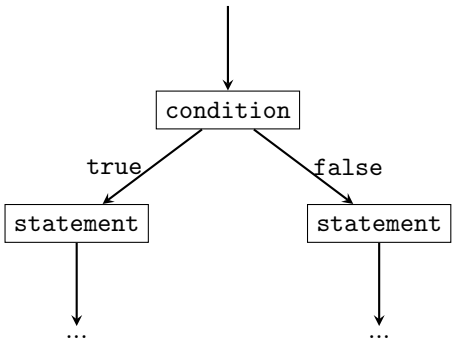



Figure 3.1: Simple Control Flow Graph

Certain types of analysis are *path-sensitive*, meaning that the analysis differentiates between the true and false branches of conditional statements. This distinction is crucial for analyzes

such as constant conditions (Section 4.3) and nullness analysis (Section 4.2), where outcomes may depend on the specific execution path taken. Conversely, *path-insensitive* analysis does not consider specific execution paths. An example of this is queried field access analysis, which aims to identify all fields that might be queried, irrespective of the execution path. The truth or falsity of a particular branch condition does not affect the set of fields that are queried (Section 4.4).

Different types of data flow problems, such as reaching definitions, available expressions, and constant propagation, characterize the data flow information analyzed. The complete set of data flow information is referred to as the domain of the analysis, $\mathcal{D}$. The way data flow information is propagated between program points is defined by a transfer function $f$, which characterizes the analysis.

**The Transfer Function** $f : \mathcal{D} \to \mathcal{D}$ is applied to each node $n$ in the CFG, defining how a node $n$ affects propagated information:

$$f_n(DF_{\text{entry}}(n)) = DF_{\text{exit}}(n)$$

where $DF_{\text{entry}}(n)$ and $DF_{\text{exit}}(n)$ symbolize the data flow information before and after node $n$ has been processed, respectively. Transfer function uses two key operations—*gen* and *kill*—to manipulate the data properties of interest:

- $gen(n)$: This function identifies new or important data flow information that comes from the actions taken at node $n$.
- $kill(n)$: This function determines which data flow information is no longer valid or useful after actions are performed at node $n$.

The computation of data flow information transformation through the transfer function at node $n$ is illustrated by the following equation:

$$DF_{\text{exit}}(n) = (DF_{\text{entry}}(n) \setminus kill(n)) \cup gen(n)$$

In this expression, $gen(n)$ and $kill(n)$ are functions that respectively compute the sets of generated and killed data flow information for node $n$, while $DF_{\text{entry}}(n)$ and $DF_{\text{exit}}(n)$ represent the data flow information at the entry and exit of the node. [19]

**Meet and Join Operators** When multiple nodes in a CFG converge on a single node, the analysis requires a synthesized consideration of all incoming paths to determine a unified data flow state at the convergence point. The specific operator used—either meet or join—depends on the type of analysis being performed.

For the **conservative approach**, which aims to ensure that only information confirmed by all predecessor paths is considered, the meet operator ($\wedge$) is utilized:

$$\text{Forward Analysis:} \quad \text{DF}_{\text{entry}}(n) = \bigwedge_{p \in \text{pred}(n)} \text{DF}_{\text{exit}}(p)$$

$$\text{Backward Analysis:} \quad \text{DF}_{\text{exit}}(n) = \bigwedge_{s \in \text{succ}(n)} \text{DF}_{\text{entry}}(s)$$

This equation applies the greatest lower bound to merge data flow states, ensuring a restrictive and safe integration of incoming information, which is critical for ensuring program correctness under all circumstances.

Conversely, the **optimistic approach** uses the join operator ($\vee$) to account for any possible information from incoming paths:

$$\text{Forward Analysis:} \quad \mathrm{DF}_{\mathrm{entry}}(n) = \bigvee_{p \in \mathrm{pred}(n)} \mathrm{DF}_{\mathrm{exit}}(p)$$

$$\text{Backward Analysis:} \quad \mathrm{DF}_{\mathrm{exit}}(n) = \bigvee_{s \in \mathrm{succ}(n)} \mathrm{DF}_{\mathrm{entry}}(s)$$

This application of the least upper bound allows the inclusion of the broadest set of potential outcomes, accommodating scenarios where broader possibilities need to be considered in the analysis [16, 19].

**Fixed Point** The transfer function is applied iteratively to the CFG until a *fixed point* is reached, at which point the analysis terminates. The fixed point is defined as the point where the data flow information no longer changes between iterations, indicating that the analysis has converged. The fixed point theorem [20] guarantees that the analysis will terminate and reach a stable state under conditions where the transfer function is monotonic[1] and the lattice is finite [16]. Once the fixed point is reached, the data flow information at each program point is used to detect potential errors or vulnerabilities in the code.

### 3.2.2 Symbolic Execution

Symbolic execution, initially introduced by James King [21] in 1976 and Lori Clarke [22] in the same year, is a technique for program analysis that uses symbolic rather than actual values during execution. This method allows the exploration of various execution paths and the generation of test cases that exercise different program behaviors. Symbolic execution represents program variables with symbolic values, allowing the analysis to consider all possible values these variables might assume during program execution. As execution proceeds symbolically, it generates constraints from encountered conditions such as if statements and loops. These constraints specify the necessary conditions for the program to proceed along a certain path. The collected path constraints are then analyzed using constraint solvers to find if there exist concrete input values that would satisfy the constraints, effectively leading the program down a particular path. For complex programs, the number of potential execution paths can grow exponentially, as the number of paths is proportional to the number of branches in the program. This challenge requires techniques like constraint simplification and path prioritization. [19]

In my thesis, I use symbolic execution techniques to analyze constant conditions in Apex code. The goal is to identify conditions that are always true or always false, which can help developers identify potential bugs or dead code in their programs. By symbolically executing the code and collecting path constraints, the analysis can determine if a condition is always true or always false, regardless of the input values across all possible paths. This approach is slightly different from traditional symbolic execution, which aims to explore the feasibility of specific paths. My analysis leverages data flow analysis to manage and trace the propagation of symbolic values, expressions, and constraints throughout the program. This type of analysis is path-sensitive, as it considers the program's execution paths and the predicates that govern them. Reaching a fixed point provides the most precise information possible about the program's data flow behavior at each point, enabling the detection of constant conditions.

[1] Monotonicity implies that higher input values will correspond to higher or equal output values [16]

## 3.3 Salesforce

Salesforce is a cloud-based Customer Relationship Management (CRM) platform that enables organizations to manage their customer interactions and data throughout the customer lifecycle. Established in 1999 by Marc Benioff, Parker Harris, Dave Moellenhoff, and Frank Dominguez, Salesforce has evolved from a single CRM product into a comprehensive suite of business applications designed to support various aspects of business operations, including sales, customer service, digital marketing, and e-commerce. For its most recent fiscal year ending January 31, 2024, Salesforce reported revenues of $34.9 billion [23].

Salesforce stands out in the CRM landscape primarily due to its cloud-based nature, robust ecosystem of third-party applications through its AppExchange, and the incorporation of advanced analytics and artificial intelligence capabilities. The platform is supported by a strong community and offers extensive educational resources through Trailhead, enhancing user engagement and learning. Another key aspect of Salesforce is its adaptability and customization capabilities. The platform allows businesses to tailor their CRM to meet specific needs, making it a versatile tool for a wide range of industries. To meet the demand for advanced customization and complex business logic integration within its cloud-based platform, Salesforce launched the Apex programming language in 2006.

## 3.4 The Apex Programming Language

The Apex programming language, developed by Salesforce.com, is a proprietary language specifically designed for the Salesforce environment. The official documentation notes [24] that it's a strongly typed, object-oriented language enabling developers to execute flow and transaction control on Salesforce servers, integrated with API calls. With Apex, developers can embed business logic into system events like button clicks, record updates, and Visualforce pages. This language can be triggered by web service requests or through object triggers.

### 3.4.1 Overview of Apex Programming Language

In the process of developing a static analyzer for Apex, it's essential to become familiar with the language's unique characteristics. This section explores the distinctive features, capabilities, and differences of Apex compared to other programming languages. The following discussion focuses on the key features of Apex that are particularly relevant.

As a strongly typed language, Apex requires explicit declaration of variable types, ensuring type safety at compile time. Being object-oriented, it supports classes, interfaces, and inheritance, much like Java. One of the unique features of Apex is its deep integration with Salesforce's database, which allows for querying and manipulation of stored data using Salesforce Object Query Language (SOQL), Salesforce Object Search Language (SOSL), and Data Manipulation Language (DML). Apex also allows variables to hold null values by default, and does not enforce compile-time checks for null assignments or usage, which requires careful null handling in static analysis. Unlike many other languages, Apex does not require import statements, as all files created are part of one package. This eliminates the need for explicit class imports for their usage. It's important to note that Apex code execution is subject to Salesforce's governor limits to ensure that it does not monopolize shared resources [25].

### 3.4.2 Apex Compiler

The Apex compiler is a key component of Salesforce's cloud-based development environment, uniquely optimized for multi-tenant architecture. This optimization is crucial for efficiently managing resources shared across various users. This section elaborates on the basic operation of the Apex compiler, emphasizing its unique attributes and the implications of its cloud-based infrastructure.

The workflow of the Apex compiler, as described by Salesforce Product Manager Josh Kaplan [26], begins with the submission of code to the Salesforce environment and includes several key phases:

- **Storage**: The Apex code is initially saved into Salesforce's database, ensuring secure storage and management within Salesforce's robust data management system.
- **Compilation**: The saved Apex code is then compiled into Java bytecode. This transformation is essential for executing the high-level Apex code efficiently on the Salesforce platform.
- **Interpretation**: At runtime, the Java bytecode is interpreted within Salesforce's dynamic runtime environment, crucial for the execution of compiled code.
- **Caching**: To enhance performance, the compiled Java bytecode is stored in a distributed cache, allowing for quicker retrieval and execution of frequently used code and significantly improving response times.
- **Execution**: An executable form of the bytecode is also stored on application servers, ensuring that the compiled code is ready for immediate execution when called upon.

However, the cloud-based nature of the Apex compiler introduces certain limitations that are important to consider. A key limitation is the inability to perform local compilations of Apex code. Reliance on remote servers to compile and execute code introduces latency into the development cycle. This also limits in-depth debugging that might be available in a local compilation environment. Furthermore, without local compilation, developers lack immediate feedback on syntax errors or simple logical mistakes typically caught during development. To effectively mitigate these issues, the implementation of static analysis tools becomes not merely beneficial but essential. By analyzing source code before it is pushed to the cloud, these tools reduce server interactions for basic error checks, significantly speeding up development and easing the load on cloud infrastructure.

### 3.4.3 Apex Governor Limits

Salesforce's multi-tenancy architecture ensures that code execution does not adversely impact the shared resources among users. As per the Apex documentation [25], Salesforce enforces strict limits on Apex code execution to prevent runaway code from monopolizing shared resources. If any Apex code surpasses these restrictions, the corresponding governor triggers an unmanageable runtime exception. These limits are counted on a per-transaction basis in Apex. In Batch Apex, the system resets these limits each time it executes a batch of records in the `execute` method.

The table below summarizes some of the synchronous Apex governor limits in Salesforce that developers should consider when developing applications:

| **Governor Limit** | **Limit Value** |
|---|---|
| Total number of SOQL queries issued | 100 |
| Total number of SOSL queries issued | 20 |
| Total number of DML statements issued | 150 |
| Maximum CPU time on Salesforce servers | 10,000 milliseconds |
| Maximum heap size | 6MB |
| Total number of records retrieved by SOQL queries | 50,000 |

Table 3.1: Main Salesforce Apex Governor Limits

Considering the restrictions inherent in programming with Apex, and following Salesforce's own recommendations[2], it is crucial to avoid performing resource-heavy tasks inside loops. Instead, these tasks should be moved outside loops to conserve the limits set by Salesforce for each transaction. Static analysis tools are particularly useful in these scenarios, as the compiler does not account for these specific limits. Without the aid of static analysis tools, these critical errors may not be detected and could become part of the final product.

[2]Apex Design Best Practices: https://developer.salesforce.com/wiki/apex_code_best_practices

# Chapter 4

# Implementation

## 4.1 Control Flow Graph

### 4.1.1 ANTLR

In developing the static analysis tool that I implemented in Java, I utilized the open-source ANTLR grammar `apex.g4` [27] as the foundational tool for parsing Apex code. This grammar, hosted in the grammars-v4 repository on GitHub, provides a syntactic specification of the Apex programming language. However, the original ANTLR grammar for Apex, derived from Java 1.7, included unnecessary constructs and was defined with excessive granularity. To enhance the grammar's utility and facilitate the creation of control flow graphs, I implemented several modifications.

One major area of revision was the simplification of the statement structure to better support parse tree traversal and control flow graph construction. Originally, the grammar explicitly defined every type of control and loop statement explicitly within the main `statement` rule. For instance, the grammar defined `IF`, `FOR`, `WHILE`, and `DO` statements explicitly within the `statement` rule:

```
1  statement :
2    | IF parExpression statement (ELSE statement)?
3    | FOR '(' forControl ')' statement
4    | WHILE parExpression statement
5    | DO statement WHILE parExpression ';'
```

In the revised grammar, these are categorized under generalized control flow constructs:

```
1  ifStatement :
2    | IF parExpression statement elseIfStatement?
3    | IF parExpression statement elseStatement?
```

```
1  iterationStatement :
2    | whileStatement
3    | doWhileStatement
4    | forStatement
```

This refactoring introduces clear categories like `ifStatement` and `iterationStatement`, making the grammar easier to understand and traverse. Another key modification was the introduc-

tion of the `jumpStatement` category, which consolidates `RETURN`, `THROW`, `BREAK`, and `CONTINUE` statements, enhancing the modularity of the grammar:

```
jumpStatement :
  | returnStatement
  | throwStatement
  | breakStatement
  | continueStatement
```

This change simplifies the grammar and enables the tool to identify and analyze jump statements more effectively. Furthermore, I consolidated the method invocation constructs in the `expression` rule of the grammar:

```
expression :
...
    | methodInvocation
...
```

```
methodInvocation :
    | (primary '.')? Identifier '(' expressionList? ')' ('.'
    ↪  methodInvocation)*
    | primary '[' expression ']' ('.' methodInvocation)*
```

### 4.1.2 CFG Construction

ANTLR generates a parser for Apex using the provided grammar file. This parser constructs parse trees, which are data structures that map the grammar rules to the given input. It also creates tree walkers, designed to traverse tree nodes and execute application-specific code. I used listeners to navigate through the parse tree and build the control flow graph. These listeners have methods that act as callbacks, which the parser triggers as it enters or exits specific grammar rules. This mechanism allows the listener to perform actions in response to different parts of the input, making it easy to decouple the parsing process from later data handling and processing tasks. [28, 29]

Building upon this foundational understanding of ANTLR, I drew significant inspiration from Eldar Timraleev's open-source project on GitHub [30]. Timraleev's tool, implemented in Kotlin, constructs control flow graphs for C programming language, primarily focusing on visual representations of C code structures. In contrast, my project adapts these foundational concepts to the Apex language, aiming solely to apply the analysis of control flows within Apex.

Given the broad and complex nature of the Apex programming language, I made simplifications to the initial CFG implementation to manage the project scope effectively and demonstrate the core functionality within the constraints of my thesis. In particular, I excluded the `switch` statement from my analysis, considering it syntactic sugar for a series of `if-else` statements and thus redundant in my initial prototype. Similarly, I omitted `for each` loop, focusing instead on the fundamental dynamics of basic loops such as `while-do`, `do-while`, and `for loop`, which are sufficient to represent iterative control structures in general. Additionally, I chose not to include cross-file control flow in the CFG because Apex lacks import statements (discussed in Section 2.1), making it hard to explicitly connect files. This avoids the difficulties and possible errors of manually tracking file dependencies, keeping the project scope clear and focused.

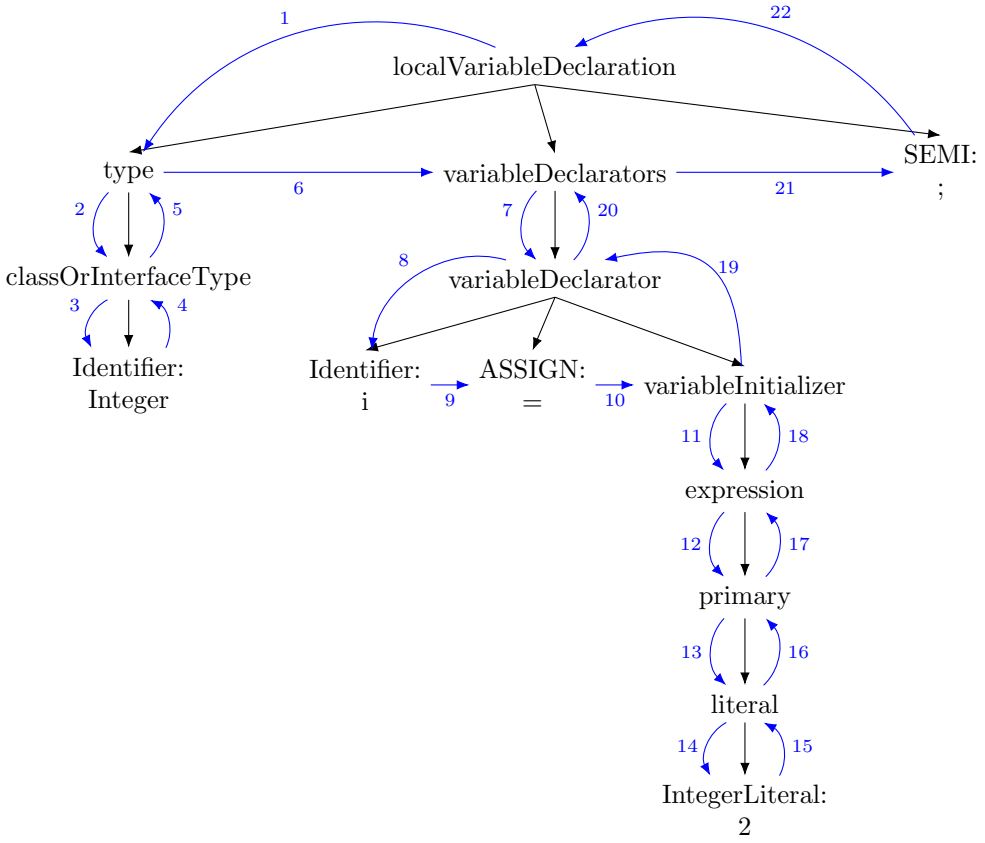



Figure 4.1: Depth-First Search Tree Traversal Demonstrating the Listener Pattern in ANTLR

In terms of node representation within the CFGs, each node corresponds to a single statement, unlike some analyzers where a node might represent a block of code consisting of multiple statements. This design choice ensures that all nodes are uniformly sized, facilitating easier debugging and smoother processing.

## 4.2 Nullness Analysis

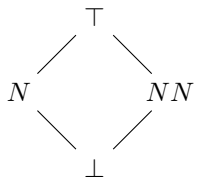
Null references, often referred to as "the billion-dollar mistake," were introduced by Sir Charles Antony Richard Hoare, a Turing Award laureate, in 1965. Hoare himself has acknowledged that the creation of null references was motivated more by ease of implementation than by a deliberate design choice[1].

Accessing a null reference can cause applications to crash due to null pointer exceptions. Various programming languages have adopted different strategies to enhance null safety. For instance, Kotlin mitigates the risk of null pointer exceptions by prohibiting null assignments to variables declared as non-nullable types [31]. Similarly, Rust incorporates an ownership system and the Option type, which requires developers to explicitly manage the presence or absence of values. This reduces the likelihood of runtime errors related to null references [32]. Apex does

[1] https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/

not inherently enforce null safety, which presents a challenge for developers. This gap highlights the importance of adopting specialized static analysis techniques for Apex to mitigate the risks associated with null pointer exceptions.

**Lattice Definition** In certain nullness analysis models, values are categorized into two distinct states, $\top$ and $NN$, where the bottom element $NN$ means definitely not null and the top element $\top$ represents values that may be null [16]. The model's soundness relies on its conservative approach, treating variables as possibly null unless it can guarantee they are not. However, I consider this approach to be somewhat imprecise. Its completeness may be compromised, which could overlook nuanced scenarios where a variable's nullness can be more precisely defined. This could result in potential false negatives. Therefore, I have identified four distinct states as the elements of this lattice, as shown in the figure below:




The refined states are outlined as follows:

- **Definitely Null ($N$):** Represents that a pointer is *guaranteed* to be null.
- **Possibly Null ($\top$):** Indicates that a pointer may or may not be null.
- **Definitely Non-Null ($NN$):** Signifies that a pointer is *guaranteed* not to be null.
- **Unknown ($\bot$):** This state is used for pointers whose nullability status cannot be determined.

The use of an "Unknown" state may be questioned as nullability is essentially a binary concept, consisting of either null or non-null. The nullable state already addresses the uncertainty between these values. This is particularly relevant when interacting with external libraries or components for which source code may not be available, or the analysis thereof is not feasible due to complexity or licensing restrictions. In situations like these, pointers received from these external components may need to be marked as "Unknown" to be cautious. This is because their behavior and the circumstances under which they return null pointers may not be entirely clear. A state of "Unknown" enables a conservative approach. If the analysis cannot guarantee the nullability status of a pointer, it refrains from making potentially incorrect assumptions and generating false positives.

When control flow paths converge in a program, the lattice's partial ordering allows for the merging of nullability states from different paths. The join operation ($\sqcup$) is defined as follows:

| $\sqcup$ | $N$ | $NN$ | $\top$ | $\bot$ |
|---|---|---|---|---|
| $N$ | $N$ | $\top$ | $\top$ | $N$ |
| $NN$ | $\top$ | $NN$ | $\top$ | $NN$ |
| $\top$ | $\top$ | $\top$ | $\top$ | $\top$ |
| $\bot$ | $N$ | $NN$ | $\top$ | $\bot$ |

**Operations Generating Data Flow Facts** The domain of nullness analysis involves a set of variables, with the analysis proceeding forward. The semantic constructs responsible for generating data flow facts about a variable include:

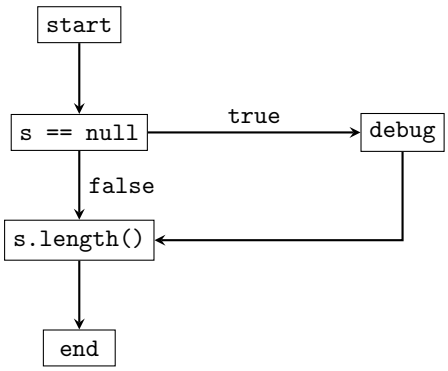
- **Null Check:** This operation generates two data flow facts: one indicating that the variable is null, and another indicating that it is not null. These facts are then propagated to the respective branches.
- **Variable Declarations without Initialization:** This operation generates a null state for the variable, signifying that it has not been initialized.
- **Explicit Null Assignments:** This operation generates a data flow fact indicating that the variable is null.
- **Explicit Non-null Assignments:** Assigning a non-null value to a variable (whether through a literal, a non-null expression, or a method call guaranteed to return non-null) establishes a $NN$ state.

Each operation resets any previously inferred states of a variable. For any other operations not manipulating the variable, the algorithm will propagate data flow facts further in the graph.

**Illustrative Examples** Let's take an example of the following small illustrative code snippet:

```
1  public class MyClass {
2      public void foo(String s) {
3          if (s == null) {
4              System.debug('s is null!');
5          }
6          s.length();
7      }
8  }
```

Control Flow Graph of the code will be represented as following:




The condition produces two facts about variable `s`, overriding any previously inferred state, and propagating to the respective branches. The true branch contains a debug statement, which neither generates nor eliminates any facts about the variable's state, thus allowing the propagation of existing facts. The node `s.length()` receives two states on input, which are combined into one nullable state. The analyzer identifies operations that can potentially produce null pointer

| Node | GEN | KILL | IN | OUT |
|---|---|---|---|---|
| `if (s == null)` | $\{(s, N), (s, NN)\}$ | $\{(s, \bot)\}$ | $\emptyset$ | $\{(s, N), (s, NN)\}$ |
| `System.debug('s is null!')` | $\emptyset$ | $\emptyset$ | $\{(s, N)\}$ | $\{(s, N)\}$ |
| `s.length()` | $\emptyset$ | $\emptyset$ | $\{(s, \top)\}$ | $\{(s, \top)\}$ |

Table 4.1: Summary of Nullability Analysis for Variable $s$ Across Program Nodes

exceptions. If these operations are performed in a nullable state, it alerts with a warning that the variable *may* throw a null pointer exception. In cases where the variable is in a null state, the analyzer confidently warns that the variable *will* throw a null pointer exception. The result of the data flow analysis for each program node can be summarized in Table 4.1. The example below illustrates the scenario of an "Unknown" state value:

```
1  public class MyClass {
2      public void bar(String s) {
3          s.trim();
4      }
5  }
```

We have no information regarding the variable `s` it may still contain a null value. However, the context in which the method is called remains unknown - the argument could be validated before being passed to the method. Therefore, adding null check before accessing the variable might be redundant. In such cases, the analyzer should not issue a warning. [33]

## 4.3 Constant Condition Analysis

A constant condition refers to a boolean expression within a program's code that evaluates to a fixed value (`true` or `false`) and remains unchanged regardless of the program's state or input. These conditions, often resulting from programming oversights or logical redundancies and can lead to unreachable code paths (dead code) or unnecessary evaluation, impacting the application's efficiency and clarity.

Traditionally, compilers eliminate unreachable code through *peephole optimization* by removing conditions whose values are known at compile time through *constant propagation*. The optimizer evaluates the condition based on the constants involved. For example, if the condition involves a comparison between two constant values, the optimizer can determine the outcome of this comparison at compile time. If the condition evaluates to `true`, the code block associated with the condition is kept, while any alternative code blocks (such as those in the `else` part of an `if-else` statement) are removed. Conversely, if the condition evaluates to `false`, the code block associated with the condition is removed, and the optimizer will decide whether any alternative code blocks should be executed instead. [34] The following is an example of a constant condition identified through peephole optimization:

```
1  Integer x = 3;
2  Integer y = x * x;
3  Integer z = x * 2;
4
5  if (x + z == y) {
```

```
6       System.debug('Impossible');
7   }
```

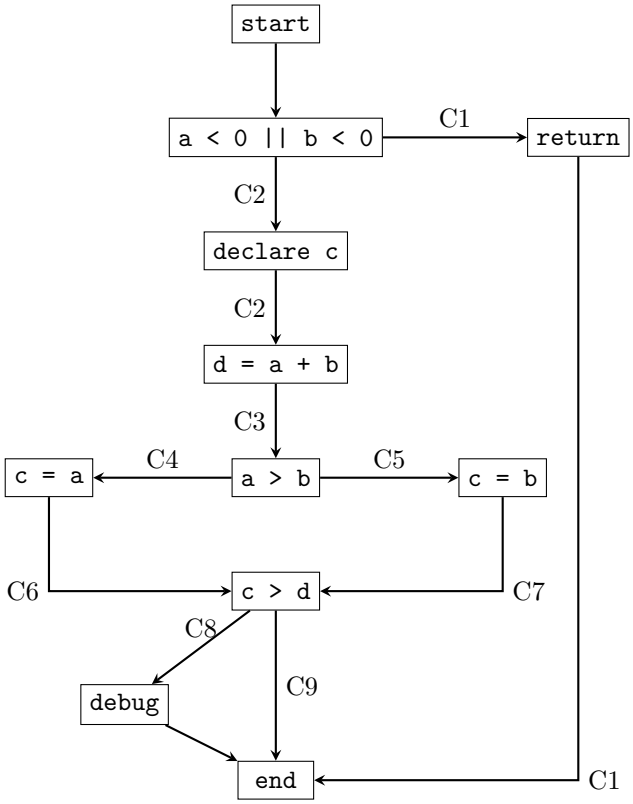
However, in my thesis, I was particularly interested in magical cases, as follows:

```
1   public void foo(Integer a, Integer b) {
2       if (a < 0 || b < 0) {
3           return;
4       }
5
6       Integer c;
7       Integer d = a + b;
8
9       if (a > b) {
10          c = a;
11      } else {
12          c = b;
13      }
14
15      if (c > d) {
16          System.debug('Impossible!');
17      }
18  }
```

In this code snippet, the condition `if (c > d)` will always evaluate to `false`. This is because `c` is determined to be the greater (or equal) of two non-negative integers `a` and `b`, and `d` is the sum of `a` and `b`. Logically, the sum of two non-negative integers (`d`) will always be equal to or greater than either of the integers individually (`c`). Therefore, it is impossible for `c` to be greater than `d`, making the statement `System.debug('Impossible!')` unreachable and the condition a constant false.

This type of problem cannot be effectively tackled by traditional algorithms or brute force methods. An effective alternative is *symbolic execution*, a technique designed to analyze programs by executing them on symbolic inputs instead of concrete values (discussed in Subsection 3.2.2).

**Implementation** In my tool, I utilized the Z3 Theorem Prover, a satisfiability modulo theories (SMT) solver developed by Microsoft Research. I chose Z3 due to its efficiency and robustness in solving constraints. My program systematically explores a CFG and generates constraints based on variable manipulations and conditional statements encountered along the paths. For each conditional statement, it traverses abstract syntax trees to identify the operands and operators involved, then formulates constraints accordingly. This analysis is path-sensitive, meaning the program propagates different constraints along the various branches originating from conditional nodes. When paths in the CFG converge, the join operator combines constraints from different paths by creating a single constraint. Let's revisit a previous example and construct its CFG, with constraints generated at each node:




| Identifier | Constraint Expression |
|---|---|
| C1 | $a < 0 \lor b < 0$ |
| C2 | $a \geq 0 \land b \geq 0$ |
| C3 | $(a \geq 0 \land b \geq 0) \land d = a + b$ |
| C4 | $(a \geq 0 \land b \geq 0) \land d = a + b \land a > b$ |
| C5 | $(a \geq 0 \land b \geq 0) \land d = a + b \land a \leq b$ |
| C6 | $(a \geq 0 \land b \geq 0) \land d = a + b \land a > b \land c = a$ |
| C7 | $(a \geq 0 \land b \geq 0) \land d = a + b \land a \leq b \land c = b$ |
| C8 | $((a > b \land c = a) \lor (a \leq b \land c = b)) \land c > a + b \land (a \geq 0 \land b \geq 0)$ |
| C9 | $((a > b \land c = a) \lor (a \leq b \land c = b)) \land c \leq a + b \land (a \geq 0 \land b \geq 0)$ |

The algorithm queries the solver for each conditional node to determine if a given constraint is satisfiable—that is, whether a solution exists for such a constraint. The solver can return one of three possible responses: yes, no, or unknown. The third option suggests that the solver is unable to provide an answer within a reasonable timeframe. If the solver indicates that a given constraint is unsatisfiable, the program will output a warning stating that the condition will always be false. Additionally, the program also queries the negation of the given constraint. If the solver finds this negation unsatisfiable, it implies that the original condition is a tautology and will always evaluate to true.

## 4.4 Queried Field Access Analysis

In Salesforce development, accessing fields of SObject rows retrieved via SOQL queries without querying the requested fields can lead to runtime exceptions and affect the robustness of applications. To illustrate this, consider the following example for inter-procedural analysis:

```
public static void foo(Id opportunityId) {
    Opportunity record = [
        SELECT Id, Name
        FROM Opportunity
        WHERE Id =: opportunityId
    ];
    bar(record);
}

private static Decimal bar(Opportunity record) {
    Decimal currentAmount = record.Amount;
    return currentAmount * 0.23;
}
```

In the above code snippet, the `foo` method queries an `Opportunity` record and passes it to the `bar` method. The `bar` method accesses the `Amount` field of the `Opportunity` record. However, the `Amount` field is not queried in the SOQL query within the `foo` method. This can lead to a `SObjectException` at runtime, as the `Amount` field is not populated in the queried `Opportunity` record. In complex codebases, it's common for developers to initially query certain fields from an object, and later, while accessing the record, they might need to access additional fields. However, they might forget to include these additional fields in the initial query.

**Implementation** The domain of queried field access analysis involves tracking the set of queried fields for each `SObject` variable and ensuring that all accessed fields are queried in the SOQL query. To achieve this, the algorithm traverses the CFG and maintains a mapping of queried fields for each `SObject` variable. It is important to note that SOQL queries do not support join clauses, so all queried fields must belong to a single `SObject`, with the exception of subqueries. Consider an Apex method that queries different sets of fields from the Account object based on various conditions:

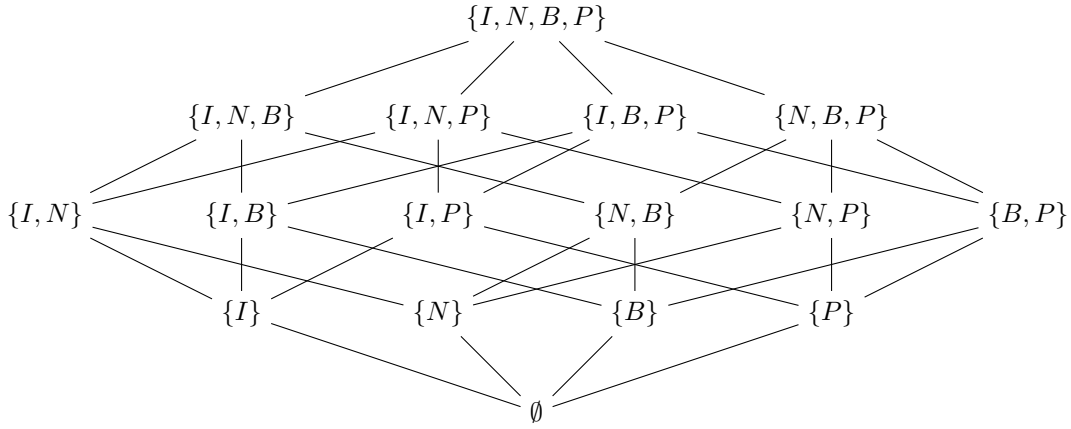
```
Account acc;

if (firstFlag) {
    acc = [
        SELECT Id, Name
        FROM Account
        WHERE Id =: PARTNER_ID
    ];
} else if (secondFlag) {
    acc = [
        SELECT Id, BillingCity
        FROM Account
        WHERE Id =: CUSTOMER_ID
    ];
```

```
15      return acc;
16  } else {
17      acc = [
18          SELECT Id, Name, Phone
19          FROM Account
20          WHERE Id =: INTERNAL_ID
21      ];
22  }
23
24  if (thirdFlag) {
25      return acc.Phone.startsWith('+420');
26  } else {
27      return acc.Name.contains('CZ');
28  }
```

To represent the queried fields in the program, I use a powerset lattice, which is a mathematical structure useful for tracking combinations of queried fields. Let's denote a set $A = \{a_1, a_2, \ldots\}$ that represents all possible fields which can be queried. The powerset $\mathcal{P}(A)$ includes every possible subset of $A$, ranging from the empty set $\emptyset$ to the full set $A$ itself. We define a complete lattice on $\mathcal{P}(A)$ with set inclusion $\subseteq$ as the partial order. In this lattice, the least element $\bot$ is $\emptyset$, representing no fields queried, and the greatest element $\top$ is $A$, representing all fields queried. The join $\sqcup$ and meet $\sqcap$ operations are defined as the union $x \cup y$ and intersection $x \cap y$ of subsets, respectively. [16] For simplicity, I will denote the attributes as follows:

- $I$ for Id,
- $N$ for Name,
- $B$ for BillingCity,
- $P$ for Phone.

Given that I have four distinct attributes, the powerset will have $2^4 = 16$ elements (subsets), ranging from the empty set to the full set.




The algorithm propagates the queried fields through the CFG, intersecting the queried fields for each `SObject` variable at merge points. The meet operator ($\sqcap$) is used to combine the queried

fields from different paths. The algorithm issues a warning if an accessed field is not queried in the SOQL query for the respective `SObject` variable.

In the example, the access to `acc.Phone.startsWith('+420')` is performed outside the conditional branches, assuming that the `Phone` field is always present. This assumption fails for the first and second queries where the `Phone` field is not queried. The access to `acc.Phone` triggers a warning, as the `Phone` field is not queried in the respective SOQL queries and may result in a runtime exception. However, although `Name` is not queried in the second query, `acc.Name + ' - CZ'` does not trigger a warning due to the early return statement in the second conditional branch. The result of the queried field access analysis for each program node can be summarized in Table 4.2.

| Node | GEN | KILL | IN | OUT |
|---|---|---|---|---|
| `Account acc` | $\{\perp\}$ | $\{I, N, B, P, \perp\}$ | $\emptyset$ | $\{\perp\}$ |
| `if (firstFlag)` | $\emptyset$ | $\emptyset$ | $\{\perp\}$ | $\{\perp\}$ |
| `SELECT Id, Name...` | $\{I, N\}$ | $\{I, N, B, P, \perp\}$ | $\{\perp\}$ | $\{I, N\}$ |
| `else if (secondFlag)` | $\emptyset$ | $\emptyset$ | $\{\perp\}$ | $\{\perp\}$ |
| `SELECT Id, BillingCity...` | $\{I, B\}$ | $\{I, N, B, P, \perp\}$ | $\{\perp\}$ | $\{I, B\}$ |
| `return acc` | $\emptyset$ | $\emptyset$ | $\{I, B\}$ | $\{I, B\}$ |
| `else` | $\emptyset$ | $\emptyset$ | $\{\perp\}$ | $\{\perp\}$ |
| `SELECT Id, Name, Phone...` | $\{I, N, P\}$ | $\{I, N, B, P, \perp\}$ | $\{\perp\}$ | $\{I, N, P\}$ |
| `if (thirdFlag)` | $\emptyset$ | $\emptyset$ | $\{I, N\}$ | $\{I, N\}$ |
| `return acc.Phone.starts...` | $\emptyset$ | $\emptyset$ | $\{I, N\}$ | $\{I, N\}$ |
| `else` | $\emptyset$ | $\emptyset$ | $\{I, N\}$ | $\{I, N\}$ |
| `return acc.Name.contai...` | $\emptyset$ | $\emptyset$ | $\{I, N\}$ | $\{I, N\}$ |

Table 4.2: Summary of Queried Field Access Analysis for `acc` Across Program Nodes

## 4.5 Loop Optimization in Apex for Governor Limits

Loops, by their nature, can easily increase the use of resources due to the repetitive running of their code. In Apex, where strict resource limits are enforced, performing resource-intensive operations like SOQL queries or DML operations within loops can quickly exhaust these limits. This exhaustion can result in runtime exceptions, transaction rollbacks, and a decline in both application reliability and user experience (discussed in Subsection 3.4.3).

To optimize resource utilization, it is advisable to aggregate data before executing queries or DML operations outside of loops. This approach reduces the frequency of queries and DML operations, thereby enhancing performance and resource efficiency. An illustrative example includes shifting from executing queries within a loop to aggregating required IDs for a single bulk query:

```
1  for (Contact c : [
2          SELECT Id, FirstName, LastName
3          FROM Contact
4          WHERE AccountId = :someAccountId]) {
5
```

```
    List<Opportunity> opportunities = [
        SELECT Id, Name, ContactId
        FROM Opportunity
        WHERE ContactId = :c.Id
    ];

    // Additional processing of opportunities
}
```

Refined approach with a bulk query outside the loop:

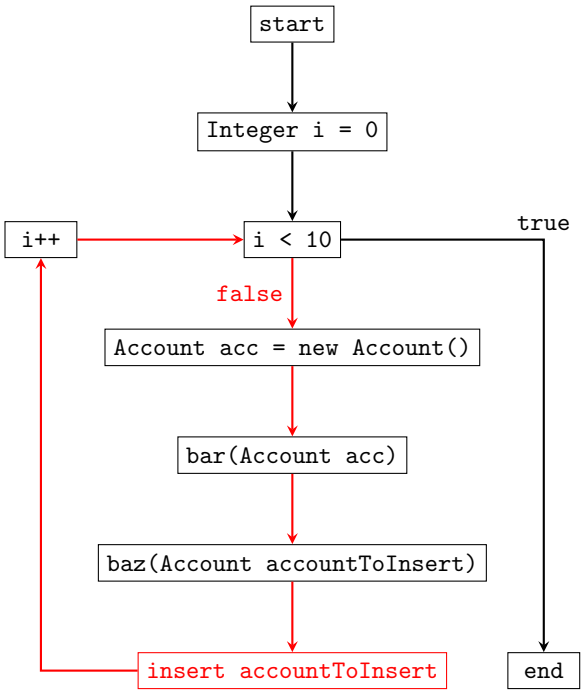
```
Map<Id, Contact> contacts = new Map<Id, Contact>([
        SELECT Id, FirstName, LastName
        FROM Contact
        WHERE AccountId = :someAccountId
]);

Map<Id, List<Opportunity>> oppsByContact = new Map<Id, List<Opportunity>>();

for (Opportunity opp : [
      SELECT Id, Name, ContactId
      FROM Opportunity
      WHERE ContactId IN :contacts.keySet()
]) {
    if (!oppsByContact.containsKey(opp.ContactId)) {
        oppsByContact.put(opp.ContactId, new List<Opportunity>());
    }
    oppsByContact.get(opp.ContactId).add(opp);
}

for (Contact cont : contacts.values()) {
    List<Opportunity> opportunities = oppsByContact.get(cont.Id);
    if (opportunities != null) {
        // Additional processing on opportunities
    }
}
```

**Implementation** During the construction of the Control Flow Graph, the algorithm identifies and marks all nodes that are located inside loops. Detecting recursive loops was relatively straightforward. The general strategy involved generating a unique signature for each method, which included the method's name and its parameters. Whenever a method was called or defined, the algorithm would push its signature onto a stack. If this signature was already present in the stack, the algorithm would identify and mark the node as part of a recursive loop. It then traverses the CFG, visiting each node within a loop to check for resource-intensive operations, such as SOQL queries or DML operations. If the algorithm identifies such operations, it will issue a warning to the developer, suggesting that they move the operation outside the loop to optimize resource utilization. This analysis has added value as constructing control flow graphs enables potential for inter-procedural analysis. This allows the algorithm to identify operations encapsulated within method calls and provide optimization suggestions accordingly. This feature is especially valuable because in large codebases, developers may not always be aware of the

context in which a method is called and the potential impact of the method's operations on resource utilization. To illustrate, let's consider the following example:

```
1  public void foo() {
2      for (Integer i = 0; i < 10; i++) {
3          Account acc = new Account();
4          bar(acc);
5      }
6  }
7
8  private void bar(Account account) {
9      // some logic
10     baz(account);
11 }
12
13 private void baz(Account accountToInsert) {
14     // some logic
15     insert accountToInsert;
16 }
```

In this example, the `foo` method contains a loop that creates new `Account` objects and calls the `bar` method to perform additional logic. The `bar` method, in turn, calls the `baz` method to insert the `Account` object into the database. The `baz` method contains a DML operation (`insert`) within a loop, which can lead to resource exhaustion due to the repetitive execution of the operation. The algorithm identifies this pattern and suggests moving the DML operation outside the loop to optimize resource utilization. CFG of the code will be represented as following:




Compared to the pattern-matching method (discussed in Section 2.3), constructing a CFG

allows for a more detailed analysis of code structure. This enables the identification of resource-intensive operations within method call chains and loops created by recursion. Such an approach delivers more precise optimization suggestions and enhances the overall efficiency of the codebase. Let's consider the following example of a recursive method:

```
1  public void foo(Account accountToEvaluate) {
2      if (accountToEvaluate != null) {
3          bar(accountToEvaluate);
4      }
5  }
6
7  private void bar(Account acc) {
8      if (acc.ParentId != null) {
9          Account parent = [
10             SELECT Id, ParentId
11             FROM Account
12             WHERE Id = :acc.ParentId
13         ];
14         bar(parent);
15     }
16     baz(acc);
17 }
```

In this example, the `foo` method calls the `bar` method, which recursively queries the parent `Account` records. The `bar` method contains a SOQL query within a recursive loop, which can lead to resource exhaustion due to the repetitive execution of the query. The CFG of the code will be represented as following:

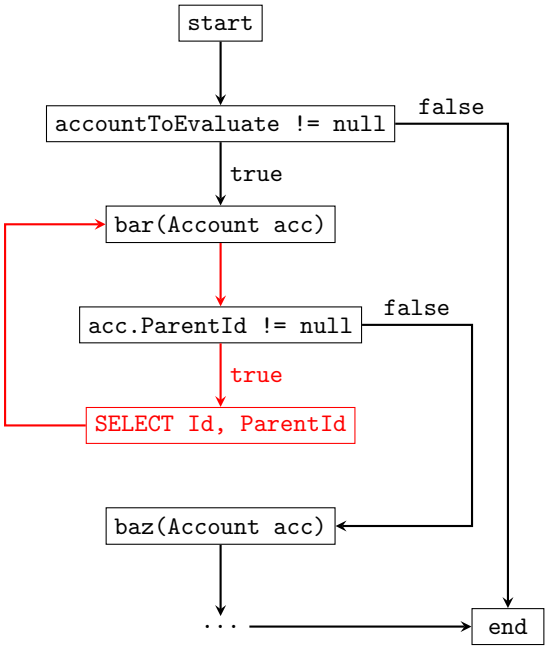



Figure 4.2: Control Flow Graph of the Recursive Code Snippet

## 4.6 Integration

Integrating static analysis tools into the software development process can be achieved through various methods, such as IDE plugins and CI/CD pipelines. IDE plugins can offer real-time feedback on potential issues within the codebase, enabling developers to address them immediately during development. However, the implementation of IDE plugins involves complications as they are specific to particular IDEs, which can be problematic since developers may prefer different IDEs. On the other hand, CI/CD pipelines offer a centralized and consistent approach to running static analysis tools. They ensure that static analysis tools are executed on the codebase with every commit, identifying potential issues early in the development process. While CI/CD pipelines do not provide immediate feedback to developers as they are coding, using a combination of IDE plugins and CI/CD pipelines leverages both approaches' strengths and mitigates their weaknesses, ensuring comprehensive code quality checks.

In this thesis, I have developed a Command Line Interface (CLI) tool that can be integrated into CI/CD pipelines to provide automated static analysis specifically for Salesforce Apex codebases. This tool was chosen for its ease of use and compatibility with various development environments. Future work could include extending the integration of this tool to IDEs, which would enhance its accessibility and immediate usability for developers.

### 4.6.1 Design and Implementation using Picocli

In my thesis, I have chosen to develop the Command Line Interface tool named *Apex Insight* using Picocli, which, according to its official documentation [35], is a lightweight framework for building Java command-line tools. It offers a concise and user-friendly API with a comprehensive feature set, including command autocompletion, subcommand handling, and parameter validation. Its unique single-file architecture removes the need for external libraries. Moreover, Picocli supports ahead-of-time compilation into GraalVM native images, which greatly improves startup speed and minimizes memory usage, enabling the distribution of applications as standalone executables.

*Apex Insight* is designed to perform analysis on single files, with options to output results in various formats such as JSON and XML. The tool supports multiple types of analysis, including Nullness, Constant Condition, Constant Propagation, Live Variable, Queried Field Access, Loop Optimization, and Uninitialized Variable Access. Its modular architecture allows new analysis types to be added and the analysis process to be customized. The output can be generated in various formats, providing developers with flexibility in tracking code quality metrics and identifying areas for improvement. The following is command structure for the CLI tool:

- `-a, --analysis=<analysis>`: Specifies the type of analysis to perform. Available options include `Nullness`, `ConstantCondition`, `ConstantPropagation`, `LiveVariable`, `SoqlFieldUsage`, `GovernorLimit`, and `UninitializedVariableUsage`.
- `-eq, --equations`: Prints the data flow equations when used.
- `-f, --format=<format>`: Specifies the format of the output file. Options are `JSON` and `XML`.
- `-h, --help`: Displays the help message and exits the program.
- `-o, --output=<outputPath>`: Specifies the path for the output file.
- `-p, --path=<filePath>`: Specifies the path of the file to analyze.
- `-V, --version`: Prints the version information and exits the program.

The tool is invoked from the command line with the `apex-insight` command, followed by the desired options. For example, to perform a `Nullness` analysis on a file located at `/path/to/file` and output the results in `JSON` format to `/path/to/output`, the following command can be used:

```
apex-insight -a ConstantCondition -p /path/to/file -f JSON -o /path/to/output
```

The tool processes the input file, performs the specified analysis, and generates the output file in the specified format. The output file contains the analysis results, including the analyzed file path, analysis type, results, and timestamp. This information can be used to track code quality metrics.

```
{
  "filePath": "src/main/resources/example/Test.cls",
  "analysisType": "ConstantCondition",
  "results": [
    {
      "condition": "(5!=5)",
      "always": false,
      "line": 3
    }
  ],
  "timestamp": "2024-04-11T07:43:22.718295"
}
```

Listing 1: Example Output of the `ConstantCondition` Analysis in JSON Format

### 4.6.2 Integration with CI/CD Pipelines

To enable the integration into CI/CD pipelines, the *Apex Insight* tool has been containerized using Docker and published to a registry. Per the official documentation [36], Docker is a popular containerization platform that provides a standardized, lightweight environment for running applications, ensuring consistent behavior across various environments. By containerizing the *Apex Insight* tool with Docker, it can be encapsulated along with all its dependencies and configurations. This eliminates the need for manual setup and installation, simplifying deployment processes. Teams can easily pull the Docker container image and execute the tool within their CI/CD pipelines, enabling automated static analysis of Salesforce Apex codebases. To ensure consistency and reproducibility across different environments, the container image can be tagged with a version number [36]. In my thesis, I published the Docker container image using the GitHub Packages Container Registry, making it accessible across repositories within my private organization. The process of building the image and publishing it to the registry has been automated using GitHub Actions, which runs on every push to the main branch. Docker image can be pulled and executed using the following commands:

```
docker pull ghcr.io/belekomurzakov/apex-insight:latest
docker run ghcr.io/belekomurzakov/apex-insight:latest -p /path -a <analysis>
```

Details on integrating the tool into CI/CD pipelines using GitHub Actions are provided in Appendix A. This workflow is triggered on every push to the main branch and consists of a job named `static-analysis` that runs in an Ubuntu environment. The job includes the following steps:

1. Checking out the repository code.
2. Logging into the GitHub Container Registry using a GitHub token.
3. Listing the `.cls` files that have been changed in the current push or pull request.
4. Pulling the `apex-insight` Docker container image, mounting the repository code as a volume, and running the tool on each changed `.cls` file using the `Nullness` analysis. The output of the tool is displayed in the pipeline logs.

The tool can be extended to include additional analyzes and customized configurations based on project requirements.

```
49  Status: Downloaded newer image for ghcr.io/belekomurzakov/apex-insight:latest
50  ghcr.io/belekomurzakov/apex-insight:latest
51  [INFO]  Analyzing file: /data/MyClass.cls
52  [INFO]  Performing analysis: Nullness
53  [SUCCESS]  No bugs found this time. Remember, absence of evidence is not evidence of absence!
```

Figure 4.3: Successful Execution of the Static Analysis in CI/CD Pipeline

```
49  Status: Downloaded newer image for ghcr.io/belekomurzakov/apex-insight:latest
50  ghcr.io/belekomurzakov/apex-insight:latest
51  [INFO]  Analyzing file: /data/Test.cls
52  [INFO]  Performing analysis: Nullness
53  Error:   s.trim() might throw NullPointerException on line 7
54  Error: Process completed with exit code 1.
```

Figure 4.4: Error Message in the Static Analysis Output

# Chapter 5

# Testing and Evaluation

Testing is an essential part of the software development process, which I regard as crucial for ensuring that software functions correctly and reliably. In this chapter, I explore the testing methodologies that I applied to validate the prototype and ensure its operational correctness. The discussion covers unit testing, integration testing, and the application of specific frameworks and tools such as JUnit and GitHub Actions. I highlight how these methodologies contribute to the overall software quality assurance, helping to identify deficiencies in the code while ensuring its robustness. Following an overview of the testing strategies, I detail the specific test cases used, present the outcomes obtained, and discuss the implications of these findings for the prototype's functionality.

## 5.1 Testing Methodology

### 5.1.1 Testing Frameworks and Tools

In this thesis, I chose JUnit as the primary testing framework due to its popularity and extensive support in the Java community. JUnit provides a simple yet effective way to write and run tests, which is essential for any rigorous software testing effort. I utilized JUnit's annotations to define test methods and classes, as well as its assertions to verify the expected behavior of the code under test.

Additionally, I integrated GitHub Actions into my development process to set up a CI/CD pipeline. This automation ensures that my tests are executed reliably whenever changes are pushed to the repository, maintaining the stability and functionality of the code throughout the development process.

### 5.1.2 Unit Testing

Unit testing formed a fundamental part of my testing strategy for the prototype. My approach involved writing tests for each functionality within my prototype, particularly focusing on the construction of control flow graphs. Each unit test aimed to isolate individual components to verify their correctness against predefined expectations.

For example, the test for the method `shouldHandleIfStatement`, designed to ensure the CFG's accurate representation of conditional branches:

```
1  @Test
2  private void shouldHandleIfStatement() {
```

```
        String apex = """
                    public class Test {
                        public void foo(Integer x) {
                            x = 1;
                            if (x > 1) {
                                System.debug('x > 1');
                            }
                        }
                    }
                     """;
        MethodGraph cfg = buildMethodGraph(apex);

        Node entryNode = cfg.getEntry();
        assertEquals("foo(Integer) :: start",  entryNode.getName());

        // Other assertions
    }
```

This test verifies that the CFG correctly handles both the true and false conditions of an `if` statement, demonstrating my tool's robustness in simulating control flow accurately. Similarly, I implemented other tests such as `shouldHandleIfElseIfElseStatement`, `shouldHandleForLoop`, and `shouldHandleWhileLoop` to ensure comprehensive coverage of all potential control flow scenarios within the application. These tests collectively ensure that each component not only meets its functional requirements but also integrates smoothly within the larger system context.

### 5.1.3 Integration Testing

Integration testing was crucial to ensure that all components of the static analysis tool worked seamlessly together. This testing phase focused on the interactions between the command-line interface (CLI) and the analysis engine, particularly in handling user input and executing analysis commands accurately.

#### Setup and Teardown

To maintain test integrity, I set up a clean testing environment for each test case:

```
@BeforeEach
public void setUpTestEnvironment() throws IOException {
    // Setup code for creating a temporary file to simulate user input
    mockFilePath = Files.createTempFile("mockFile", ".cls");
}
```

```
@AfterEach
public void cleanUpTestEnvironment() throws IOException {
    // Cleanup code to remove the temporary file after tests
    Files.deleteIfExists(mockFilePath);
}
```

#### Test Scenarios

I designed various test scenarios to cover both common use cases and error handling within the CLI, such as missing file paths or analysis types. These tests ensure the tool's usability and error

resilience when interacting with users.

**Example Scenario: Missing File Path**

```
@Test
private void shouldHandleMissingFilePathError() {
    // Testing response to missing file path
    int exitCode = cmd.execute("-a", "ConstantCondition");
    assertMissingFilePathError(exitCode, sw.toString());
}
```

## 5.2 Test Cases and Results

During the testing phase, I achieved a significant milestone by validating my prototype against a comprehensive suite of test cases, covering 90% of the lines of code within the codebase. This high coverage rate highlights not only the extensive reach of the testing but also the depth of scenario validation, ensuring that the software performs reliably under a wide range of conditions.

However, the effectiveness of testing is measured not merely by the coverage rate but by the precision and relevance of the test cases themselves. Therefore, I carefully designed the tests to cover a wide array of specific scenarios, each aimed at validating particular aspects of the prototype's behavior. In the following subsections, I will detail some of the key test cases used, outlining their objectives, methods, expected results, and the actual outcomes observed.

### 5.2.1 Nullness Analysis

**Test Case 1: Guaranteed Null Pointer Dereference Detection**

- **Objective:** To validate the tool's capability to handle jump (return) statements correctly and perform accurate inter-procedural analysis to identify guaranteed null pointer dereferences.
- **Method:**

```
public class Test {
    public void foo(Integer i) {
        if (i != null) {
            return;
        }
        bar(i);
    }

    private void bar(Integer i) {
        i.format();
    }
}
```

- **Expected Result:** The tool is expected to accurately detect operations where a null value is definitively dereferenced, by correctly interpreting return statements and analyzing the method interactions.

- **Outcome:** The analysis successfully identifies a critical line in the `bar` method where `i.format()` is called. The tool conclusively determines that a null pointer dereference *will* occur due to its thorough inter-procedural analysis, highlighting the method's handling of the jump statement and later method interaction.

**Test Case 2: Conditional Initialization and Null Pointer Dereference**

- **Objective:** To verify the tool's ability to detect null pointer dereferences that occur due to conditional initialization of objects.
- **Method:**

```
1  public class Test {
2      public void foo(Boolean flag) {
3          String s;
4          if (flag) {
5              s = "Hello";
6          }
7          s.length();  // Potential null pointer dereference
8      }
9  }
```

- **Expected Result:** The tool is expected to identify that the variable `s` *might* not be initialized if `flag` is false, and thus the dereferencing call `s.length()` *could* result in a null pointer exception.
- **Outcome:** The tool successfully detected the uninitialized variable `s` when `flag` is false, indicating a potential null pointer exception at `s.length()`.

### 5.2.2 Constant Condition Analysis

**Test Case 1: Detection of Impossible Branches**

- **Objective:** To validate the tool's capability to detect unreachable branches of code resulting from constant conditions produced by prior checks.
- **Method:**

```
1  public class Test {
2      public void foo(Integer x, Integer y) {
3          int a, b;
4          if (x > 0) {
5              a = 1;
6          } else {
7              a = 0;
8          }
9          if (y > 0) {
10             b = 1;
11         } else {
12             b = 0;
13         }
14         if (a + b == 2) {
15             if (x < 0) {
```

```
                // Unreachable branch due to constant condition
            }
        }
    }
}
```

- **Expected Result:** The tool is expected to detect the unreachable branch `if (x < 0)` under the condition that both `x > 0` and `y > 0` must be true for the code to reach this point. The tool should identify this as a constant condition leading to a dead code path.
- **Outcome:** The analysis effectively identifies the unreachable `if` statement, confirming the tool's ability to analyze conditional dependencies and flag constant condition.

**Test Case 2: Detection of Contradictory Conditions**

- **Objective:** To validate the tool's capability to detect contradictory conditions that cannot logically coexist, leading to constant conditions.
- **Method:**

```
public class Test {
    public void foo(Integer x, Integer z, Integer y) {
        if (x < y) return;
        if (y < z) return;
        if (x < z) {
            // This block is potentially unreachable
        }
    }
}
```

- **Expected Result:** The tool should recognize that the conditions `x < y` and `y < z` collectively imply `x < z`. Thus, the block inside `if (x < z)` should be identified as unreachable if the previous conditions were true, indicating a contradictory path.
- **Outcome:** The analysis successfully highlights the logical contradiction in the conditions, emphasizing the tool's proficiency in detecting constant conditions across multiple conditional statements and their implications.

### 5.2.3 Queried Field Access Analysis

**Test Case 1: Nested Conditional Field Access**

- **Objective:** To test the tool's ability to handle nested conditional queries and correctly identify field access violations where a field may not be queried in all branches.
- **Method:**

```
public class Test {
    public void foo(Boolean flag, Boolean subCondition) {
        Account a;
        if (flag) {
            if (subCondition) {
                a = [SELECT Id, Name, Email, Phone FROM Account LIMIT
                ↪ 1];
```

```
            } else {
                a = [SELECT Id, Name, Email FROM Account LIMIT 1];
            }
        } else {
            a = [SELECT Id, Name, Phone FROM Account LIMIT 1];
        }
        a.Phone.length();
    }
}
```

- **Expected Result:** The tool should analyze and determine that `Phone` is not queried in the branch where `subCondition` is false but `flag` is true. Therefore, it should identify that accessing `Phone` can potentially lead to a runtime exception when `flag` is true and `subCondition` is false.
- **Outcome:** The tool accurately identified the absence of the `Phone` field in the branch where `subCondition` is false, successfully flagging the line `a.Phone.length()` as a potential error under those conditions. This outcome confirms the tool's capability in handling nested conditional logic and precisely detecting field access violations across varying execution paths.

**Test Case 2: Multiple Queries with Conditional Early Exits**

- **Objective:** To validate that the tool accurately handles multiple conditional paths with early exits and does not generate false positives when fields are properly queried before access.
- **Method:**

```
public class Test {
    public void foo(Boolean flag, Boolean subFlag) {
        Account a;
        if (flag) {
            a = [SELECT Id, Name, Email FROM Account LIMIT 1];
            if (!subFlag) {
                return; // Early exit if subFlag is false
            }
        } else {
            a = [SELECT Id, Phone FROM Account LIMIT 1];
            return; // Always exits early in this branch
        }
        a.Email.toLowerCase();
    }
}
```

- **Expected Result:** The tool should determine that the field `Email` is accessed only under conditions where it has been queried and subFlag is not false, ensuring no access violation occurs.
- **Outcome:** The analysis correctly identified that the `Email` field is accessed only when it is queried, and not in any paths leading to an early exit that would bypass its querying. This validation affirms the tool's robust handling of conditional logic and early exits, effectively avoiding any false positives related to field access.

### 5.2.4 Loop Optimization for Governor Limits

**Test Case 1: Resource-Intensive Operations within Recursive Loops**

- **Objective:** To assess the tool's ability to identify and optimize resource-intensive operations, such as database queries and updates, within recursive loops. The focus is on preventing potential violations of runtime limits due to these operations executed recursively.
- **Method:**

```
public class Test {
    public void processAccount(Account account) {
        if (account == null) return;

        // Recursive database query inside the loop
        Account childAccount = [SELECT Id, Name FROM Account
                                WHERE ParentId = :account.Id LIMIT 1];
        processAccount(childAccount);

        // Update operation inside the loop
        account.Processed__c = true;
        update account;
    }
}
```

- **Expected Result:** The tool should identify the recursive implementation of resource-intensive operations as a potential risk for exceeding runtime limits.
- **Outcome:** The analysis successfully highlighted the inefficiencies and identified specific instances where the recursion led to repeated database queries and updates.

**Test Case 4: Indirect DML Operations in a Traditional For Loop**

- **Objective:** To assess the tool's ability to identify inefficiencies and suggest optimizations for indirect DML operations performed within a traditional for loop. This test case focuses on the evaluation of how inserting new database entries iteratively through a helper method affects performance and resource usage.
- **Method:**

```
public class Test {
    public void createMultipleAccounts(Integer totalAccounts) {
        for (Integer i = 0; i < totalAccounts; i++) {
            createAccount(i);
        }
    }

    private void createAccount(Integer i) {
        insert new Account(Name = 'Test Account ' + i);
    }
}
```

- **Expected Result:** The tool should identify the execution of DML operations (insert statements) within a loop via an indirect method call as a potential risk for exceeding governor limits.
- **Outcome:** In testing, the prototype successfully identified the inefficient use of DML operations performed within the loop.

## 5.3 Evaluation

While the test cases presented in this chapter are demonstrative and do not represent the complexity of real-world codebases with multiple paths, variables, and method calls, they were carefully chosen to pinpoint the specific mechanisms of the algorithm. The core logic of the prototype remains the same when handling different functionalities and edge cases in a fully functional application. The extension of the same algorithm to handle more coding structures in larger codebases is more of a mechanical approach. Therefore, I have focused on showcasing the creative part of the prototype, providing a proof of concept that validates its effectiveness and robustness in a simplified context. This approach allows for a clear demonstration of the prototype's capabilities, while also acknowledging that its application in more complex scenarios would require additional considerations and adaptations.

# Chapter 6

# Conclusions

## 6.1 Summary of Work

In this thesis, I have developed a static code analysis tool tailored specifically for the Apex programming language. While commercial tools are often developed by teams of engineers, my work demonstrates a general approach for addressing different types of analyzes, focusing on the creative process of building and testing cases that effectively validate the concept.

I began my work by carefully inspecting the strengths and weaknesses of existing static analysis tools to understand their limitations, particularly in conducting sophisticated types of analysis for this language. My investigation identified important shortcomings in current market tools—their capacity to manage complex data flow and control flow analysis proved insufficient for ensuring the resilience and security of applications written in Apex.

Inspired to fill these gaps, I designed a prototype that incorporates advanced static analysis capabilities. This prototype serves as a proof of concept, showcasing the potential benefits of these advanced features. The results from the testing and evaluation phase indicate that the tool can effectively identify and mitigate issues, particularly in a subset of problems that are often overlooked by current analyzers. Developed as a command-line application, the tool integrates effortlessly into CI/CD pipelines, automating the analysis process and ensuring that code quality checks are consistently applied throughout the development lifecycle.

## 6.2 Future Work

The prototype developed in this thesis serves as a stepping stone toward a more comprehensive static analysis tool for the Apex programming language. The tool's current capabilities are limited to detecting a subset of issues through static analysis. Future work could focus on expanding these capabilities to encompass more edge cases within the existing types of analyzes. There is significant room for improvement, which could be achieved by enhancing the tool's foundational capabilities. For example, analysis could be improved by adding more sophisticated criteria and refining the *generate* and *kill* functions to better manage data flow facts. The tool could also be extended in the control flow graph to support inter-procedural analysis across files, various language constructs, and more complex control flow patterns.

In addition, the command-line application that currently bundles the analysis engine can be extended to support the simultaneous execution of multiple analyzes. This would provide developers with comprehensive insight into different aspects of their codebase in a single pass.

Finally, future efforts could focus on integrating the tool with popular IDEs, such as Visual Studio Code or IntelliJ IDEA. By integrating the tool into these IDEs, developers would receive real-time feedback on the quality of their code as they write it. This integration would help developers catch issues early in the development process, reducing both the time and effort required to address them.

# Bibliography


[1] THOMSON P. Static analysis: An introduction: The fundamental challenge of software engineering is one of complexity. *Queue*, 19(4):29–41, sep 2021.

[2] PMD. Apex rules. Available at: https://pmd.github.io/pmd/pmd_rules_apex.html. [online; cited 2024-04-22].

[3] Gearset. Static code analysis for apex. Available at: https://gearset.com/assets/Static-code-analysis-for-apex.pdf. [online; cited 2024-04-22].

[4] Copado. Maximize your code quality, security and performance with copado salesforce code analyzer. Available at: https://www.copado.com/resources/blog/maximize-your-code-quality-security-and-performance-with-copado-salesforce-code-analyzer. [online; cited 2024-04-22].

[5] OX Security. Megalinter. Available at: https://megalinter.io/latest/. [online; cited 2024-04-22].

[6] AutoRABIT. Codescan rule list. Available at: https://knowledgebase.autorabit.com/product-guides/codescan/quality-rules/codescan-rule-list. [online; cited 2024-04-22].

[7] Salesforce. Salesforce code analyzer. Available at: https://developer.salesforce.com/docs/platform/salesforce-code-analyzer/overview. [online; cited 2024-04-22].

[8] SonarSource S.A. Apex static code analysis. Available at: https://rules.sonarsource.com/apex/RSPEC-1871/. [online; cited 2024-04-22].

[9] Checkmarx. Checkmarx. Available at: https://checkmarx.com/. [online; cited 2024-04-22].

[10] Semgrep. Semgrep. Available at: https://semgrep.dev/. [online; cited 2024-04-22].

[11] NIELSON F., NIELSON H. R., and HANKIN C. *Principles of Program Analysis*. Springer Berlin, Heidelberg, 1999.

[12] RICE H. G. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[13] TURING A. M. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[14] COUSOT P. and COUSOT R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. Association for Computing Machinery, 1977.

[15] COUSOT P. and COUSOT R. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, 1992.

[16] MØLLER A. and SCHWARTZBACH M. I. Static program analysis. Available at: https://cs.au.dk/~amoeller/spa/, 2023. Department of Computer Science, Aarhus University.

[17] MUCHNICK S. S. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[18] ALLEN F. E. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, jul 1970.

[19] PRADEL M. Program analysis. lecture notes. Available at: https://software-lab.org/teaching/winter2020/pa/. [online; cited 2024-02-18].

[20] KLEENE S. C. *Introduction to Metamathematics*. North-Holland Publishing Company, Amsterdam, 1952.

[21] KING J. C. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[22] CLARKE L. A. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, 1976.

[23] TechTarget. Salesforce. Available at: https://www.techtarget.com/searchcustomerexperience/definition/Salesforcecom. [online; cited 2024-04-10].

[24] Salesforce. What is apex? Available at: https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_intro_what_is_apex.htm. [online; cited 2024-03-10].

[25] Salesforce. Execution governors and limits. Available at: https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_gov_limits.htm. [online; cited 2024-03-10].

[26] KAPLAN J. Peek under the hood of the new apex compiler. Available at: https://www.salesforce.com/video/303091/. [online; cited 2024-02-21].

[27] PARR T. and HARWELL S. Apex grammar. Available at: https://github.com/antlr/grammars-v4/blob/master/apex/apex.g4, 2013. [online; cited 2024-03-02].

[28] PARR T. and HARWELL S. Antlr documentation. Available at: https://www.antlr.org/. [online; cited 2024-04-15].

[29] PARR T. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2 edition, 2013.

[30] TIMRALEEV E. Control flow graph construction for c in kotlin. Available at: https://github.com/CRaFT4ik/c-control-flow-graph/tree/master. [online; cited 2024-03-25].

[31] JetBrains. Null safety. Available at: https://kotlinlang.org/docs/null-safety.html. [online; cited 2024-04-01].

[32] KLABNIK S. and NICHOLS C. Rust documentation: The option enum and its advantages over null values. Available at: https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html#the-option-enum-and-its-advantages-over-null-values. [online; cited 2024-04-05].

[33] VALEEV T. Data flow analysis in intellij idea: How the ide perceives your code. Available at: https://www.youtube.com/watch?v=xOgGhF4OB3U. [online; cited 2024-04-22].

[34] AHO A. V., LAM M. S., SETHI R., and ULLMAN J. D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[35] Picocli. Picocli: A mighty tiny command line interface. Available at: https://picocli.info/. [online; cited 2024-03-12].

[36] Docker. Docker documentation. Available at: https://docs.docker.com/. [online; cited 2024-02-17].

# Appendices

# Appendix A

# GitHub Actions Workflow for Apex Static Analysis

```
name: Apex Insight Static Analysis

on:
  push:
    branches: [master]
  pull_request:
    branches: [master]

jobs:
  static-analysis:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: read

    steps:
      - name: Checkout repo
        uses: actions/checkout@v3
        with:
          fetch-depth: 0

      - name: Log in to GitHub Container Registry
        run: |
          echo ${{ secrets.GITHUB_TOKEN }} |
          docker login ghcr.io -u ${{ github.actor }} --password-stdin

      - name: List changed .cls files
        id: files
        run: |
          files=$(git diff --name-only \
                  ${{ github.event.before }} '*.cls' || \
                  git diff --name-only \
```

```
                ${{ github.base_ref }}...${{ github.head_ref }} '*.cls')
        echo "files=$files" >> $GITHUB_ENV
        echo "::set-output name=all_files::$files"

    - name: Run Static Analysis
      if: steps.files.outputs.all_files != ''
      run: |
        docker pull ghcr.io/belekomurzakov/apex-insight:latest
        for file in ${{ steps.files.outputs.all_files }}
        do
          docker run --rm -v $PWD:/data \
          ghcr.io/belekomurzakov/apex-insight:latest \
          -p "/data/${file}" -a Nullness
        done
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```