2023-07

Efficient Extension of a Software Analysis Framework to Additional Languages

Soltanpour, Shahryar

UNIVERSITY OF CALGARY

Efficient Extension of a Software Analysis Framework to Additional Languages

by

Shahryar Soltanpour

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA
JULY, 2023

Abstract

In the current era of software development, multi-language codebases are common, and change propagation in these codebases is challenging. The existing change propagation tool ModCP is a solution that can assist software developers with propagating changes across several languages, but only one at a time. However, ModCP has some architectural problems in that make supporting new languages hard to develop and maintain for a long time. In addition, supporting change propagation across code snippets consisting of a programming language embedded inside a different programming language would be a useful feature for ModCP. To achieve this, we must detect the embedded code snippets in a code being analyzed by ModCP.

In this thesis, we develop a new, more efficient architecture for ModCP, involving a single abstract model that each language extends for its usage, resulting in complete isolation between language results. We compare our approach with a baseline version that uses the same, concrete model for all languages and adds new models when necessary. Our approach reduces code complexity and development time and makes code more compatible with best practices of development compared to the baseline.

Moreover, we design a system for ModCP to guess and validate the programming language used in code snippets, based on the initial detection of keywords, as input to execute change propagation for multi-language codes embedded inside each other. We compare our keyword detection approach with existing deep learning and brute force approaches and show that our method is the best choice if accuracy, performance, and scalability are needed simultaneously.

Acknowledgments

I want to express my heartfelt gratitude to Professor Robert Walker, my academic supervisor, for his unwavering guidance, expertise, support, and encouragement throughout my thesis journey. His insightful feedback and invaluable support have been instrumental in shaping the direction and quality of my research.

I would also like to extend my deepest appreciation to my loving family, my mother, father, and sister, for their constant support and encouragement, even from a distance. Their emotional support has been a constant source of strength and motivation.

To my partner and best friend, Samin, I am incredibly grateful to have you by my side and for your support, understanding, and patience. Your encouragement and belief have inspired me in this journey.

I would also like to thank all my friends, colleagues, and mentors who have contributed to my growth and provided valuable insights and support throughout my thesis journey.

During the period of writing this thesis, tragically, more than one hundred individuals lost their lives, and over a thousand others were unjustly arrested by the Islamic Republic, simply for demanding their fundamental human rights in Iran. In deep reverence and with a heavy heart, I dedicate this thesis to the memory of those brave souls who made the ultimate sacrifice to make our lives better.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	**Definition**
API	application programming interface
AST	abstract syntax tree
CIA	change impact analysis
CP	change propagation
SDAT	software development analysis tool

Chapter 1

Introduction

Software maintenance is the process of modifying and updating systems to improve performance, fix bugs, and satisfy new requirements (Bennett and Rajlich, 2000). Maintenance is a crucial part of a software's life cycle, and over time it gets more expensive and complex as the code base grows: software maintenance can account for up to 90% of the total cost of software ownership (Dehaghani and Hajrahimi, 2013). As a result, software maintenance activities can be a significant source of risk, impacting the cost of the software development process. The "manual" maintenance of software by software developers (i.e., without the use of specialized software tools possessing syntactic or semantic knowledge of the programming language in use) can be prone to various types of errors; to mitigate this risk, utilizing *software development and analysis tools* (SDATs) can aid in semi-automating the process, by identifying potential changes, implementing changes, and detecting errors (Fenton and Neil, 2000; Johnson et al., 2013).

SDATs play an important role in the software development process by providing automated support for tasks such as code analysis, testing, and debugging, which can significantly reduce the time and cost associated with finding software bugs or defects compared to manual inspections (Johnson et al., 2013). Dependencies are the relationships and interactions between various entities within a software system, such as components, objects, packages, and functions; dependencies manifest themselves through direct or indirect means and in contexts that are static, dynamic, or a hybrid of both (Arias et al., 2008; Angerer, 2014). Such dependencies may result in subsequent changes to dependent entities as a result of modifications to the source code of the entities upon which they depend. The management of dependencies within the system is critical in maintenance tasks such as change propagation, refactoring, and code reuse (Arnold and Bohner, 1996). Developers use SDATs to conduct such tasks in different development contexts such as code review and continuous integration (Vassallo et al., 2019).

Software *change impact analysis* (CIA) involves the identification of potential consequences of change, or estimating what needs to be modified to accomplish a change (Basri et al., 2015) — "estimating" due to the imprecise knowledge of the impending changes and to the limits of computability. Static change impact analysis techniques involve an examination of the program's syntax, semantics, relationships with other components of the system, and dependencies. The analysis process often encompasses a review of program artifacts such as code comments, data flow diagrams, control flow graphs, and other relevant components. By analyzing these elements, potential impacts of changes can be identified, enabling developers to identify and mitigate any issues that may arise from modifying the code (Sun et al., 2015).

In the context of software development, *change propagation* (CP) refers to the updating of program outputs or results due to a modification in its inputs (Acar et al., 2005, 2006); this technique aims to ensure that the system remains consistent and functional despite changes made to its components or dependencies. We can think of CIA as a way to prevent potential problems before making changes to a system, a proactive approach aimed at avoiding unexpected software defects or system failures that could arise due to the change, whereas CP is a reactive approach that aims to ensure that changes made to a system are correctly reflected in its output.

Once the change impact analysis is performed and the potential consequences of changes are estimated, developers may decide to implement the changes. However, in some cases, the implementation of changes may not be easy due to strongly coupled code entities or the relevant implementations being spread across multiple locations. Therefore, developers may resort to refactoring to make changes easier to implement (Fowler, 2019). It is important to exercise caution during the refactoring process to ensure that unintended consequences do not arise. This is where change propagation becomes useful, as it identifies unintended changes and corresponding secondary changes needed to keep the system in a consistent state (Men, 2018).

ModCP is a change propagation tool designed by Men (2018) that utilizes a unique dependency model to detect statement-level dependencies and to update the model incrementally. This model allows ModCP to efficiently analyze large code bases in terms of space and time. This tool performs incremental maintenance that makes it able to synchronize with changes in large-scale models with a 96% faster rate on average compared to re-building the model from scratch. In addition, ModCP has been shown to enhance developer productivity by completing tasks up to 50% faster (Men, 2018). Initially, the tool was specifically designed for the analysis of Java source code.

1.1 Supporting multiple languages in ModCP

In the next step of the development of ModCP, the industrial partner of the ModCP project (Find it EZ Software Corporation[1]) required an extension in order to accommodate new languages. We aimed to support some common and popular languages in use by the customers of Find it EZ: Java (Arnold et al., 2005), C# (Microsoft, 2022a), and Transact-SQL (T-SQL) (Microsoft, 2022b). The implementations for Java, C#, Python (Python Software Foundation, 2022), and T-SQL were initially created by Afzal (2020). This expansion allowed for the analysis of multilingual code bases, increasing the tool's utility and versatility.

Yet another language/technology of particular significance to Find it EZ is Crystal Reports (SAP, 2023), a business intelligence application that enables the creation and generation of reports from various data sources. Crystal Reports offers a specialized programming language referred to as Crystal Reports Formula Language (CRL). This language is used for implementing complex formulas and custom functions within Crystal Reports. With the use of CRL, data manipulation and analysis within reports are made possible; it allows for the execution of calculations, design, and layout control of reports, as well as the capability to create custom charting and graphing features.

1.1.1 Previous work

Previous efforts by Afzal (2020) to incorporate new languages in ModCP employed a methodology that aimed to maximize the utilization of existing language models and classes for new languages, with extensions only being implemented as a last resort. The ModCP codebase comprised a Visual Studio solution with a set of projects therein. The main project, called *Core*, included models that represented abstract syntax trees and parsers for each language supported. The solution utilized by Afzal to support multiple languages relied on the same models for all supported languages and, in specific cases, added supplementary classes to provide the necessary functionality, subclassing wherever possible to minimize code duplication. This approach was effective and efficient when supporting a limited number of similar languages, such as C# and Java. However, this approach might result in excessive use of anti-patterns such as Swiss army knife (Din et al., 2012) or spaghetti code (Moha et al., 2010) and an increase in code complexity, making it difficult to support subsequent new languages.

1.1.2 Our solution

As an alternative, we implement a new approach to support all existing languages, including the addition of Crystal Reports. We compare this approach to the previous methodology through the examination of

[1] https://www.finditez.com/ [accessed 2023/03/28]

complexity measures and factors involving the speed of development to determine which approach is more promising.

Refactoring to Separate Core from Extensions

Professor Robert Walker performed a major refactoring of the original codebase to eliminate all language-specific details from the *Core* project, which retained the central CP algorithms and a generic framework for extension to specific languages and technologies; all the language-specific details were moved to the *Extensions* project, with one sub-project dedicated per supported language/technology; the sub-projects retain dependencies on the *Core* project but not on each other. The test suite from the original implementation by Men (2018) was extended and used to validate the refactored codebase. The goal was to improve the maintainability of the core functionality of ModCP, while increasing the cohesion of the language-specific sub-projects and decreasing the coupling between the various projects and sub-projects as much as possible.

Supporting multiple languages based on the architecture of Walker's version

Our development approach continues the work done by Professor Walker. The strategy we employ within the *Extensions* project is to keep each language model separate from one another. The languages will only extend the base models defined in the *Core* project, and no files or models will be shared among them. This approach will help to minimize the complexity of both *Core* and *Extensions* projects, hopefully improving their maintainability and scalability for the future addition of new languages. Prior to the work undertaken in this thesis, it remained to be seen whether the refactored design for ModCP was an improvement in practice.

We implement support for C#, T-SQL, and CRL (in addition to Java, already provided therein), and we compare our development metrics with those for Afzal's version to determine whether our alternative is successful in terms of reducing complexity and increasing scalability or not.

1.1.3 Our evaluation

For evaluating our approach, we use various metrics. To evaluate the efficiency of our ModCP extension, we compare the maintainability and code complexity metrics of our ModCP version with that of the version developed by Afzal (2020). Our designed research question to evaluate this section is:

- RQ1: Is our approach able to simplify the codebase and decrease its complexity when compared to Afzal's version?

We conclude that our approach makes the code less complex and more maintainable.

1.2 Identifying embedded languages

1.2.1 Problem

During the implementation of the support for Crystal Reports within ModCP, several instances were encountered where SQL code was embedded within CRL in the form of string literals. This pattern was also observed with the use of SQL in Java and C# code. As the development of ModCP centres around a change propagation tool, the ability to parse, model, and analyze code embedded within string literals is a crucial but missing feature of this tool. To establish a relationship between the hosting code and code embedded within string literals, it is necessary to determine whether a string literal in question represents a piece of code or simply a regular string. To address this, we design and implement an approach that takes a string as input and outputs a determination as to which programming language it is written in; performance and accuracy are essential factors in the implementation of this tool, which has been integrated into the ModCP codebase. We evaluate its performance and accuracy in comparison to a brute force model (a model that tries all available parsers against the given input), an existing tool that employs deep learning approaches (called Guesslang, described in Section 2.2.4), and a Unix command named *file* to guess the programming language in which a string is written.

1.2.2 Previous work

To determine the programming language of a given string, a method is the use of the Guesslang (Somda, 2021) repository, which employs artificial intelligence and deep learning techniques. This method has the advantage of supporting multiple languages with minimal implementation effort simply by training large numbers of samples. However, its downside is that it is slower than desired for our specific uses in ModCP.

Another useful tool for this use case is the Unix *file* command (Darwin, 2011). This tool determines the type of the given input file and supports a wide range of file types, from multimedia to text files, and can also specify the programming language of some of the input text files. It uses a predefined database of file signatures, and whenever it faces a new file, it tries to match the existing patterns in the file with its database and provides a guess for the type of the given file.

A potential alternative for identifying the programming language of a given text is the brute force method. This approach involves running the parser of each language generated from the ANTLR grammar against the input text. If some existing parser is able to parse the input text without any errors, it is considered the correct parser for that language. While this approach has a precision of 100%, meaning no false positives, its main disadvantage is the difficulty in supporting new languages. This is because implementing the grammar

and parser for each version of a language is time-consuming, and adding more languages exponentially increases the time it takes to run all parsers, making the approach non-scalable.

1.2.3 Our solution

To identify an embedded language used within a string, we employ a technique known as *keyword detection*. We have a set of expected languages, and for each language, we have a set of keywords specific to that language. When a string input is received, the input stream is tokenized into a set of words, and the number of common tokens between the input stream and each set of language keywords is counted. If the count of shared keywords for a particular language exceeds a predetermined threshold, that language is considered a potential match. Following this stage, the languages are sorted in descending order based on the number of tokens matched from the input string. For each language, its parser is applied to the given input stream, and the first parser that is able to parse the string without error is identified as the language used within the input stream.

1.2.4 Our evaluation

We evaluate our keyword detection approach by comparing it with Guesslang, *file*, and the brute force approaches in terms of accuracy, performance, and scalability.

Our designed research question to evaluate this section is:

- RQ2: How accurate is our approach compared to the brute force approach, Guesslang, and *file*?
- RQ3: How does the performance of our approach compare to the brute force approach, Guesslang, and *file*?
- RQ4: How scalable is our approach in terms of performance for supporting new languages?

Our findings suggest that our keyword detection approach is superior to the other approaches if all three aspects need to be achieved. This means that if the primary concern is accuracy while scalability is not a major consideration, the brute-force approach can be employed. However, if both accuracy and scalability are important but response time is not a concern, Guesslang can be utilized as it doesn't require additional implementation for new languages. Our proposed approach strikes a balance between these factors.

1.3 Non-disclosure agreement with the industrial partner

The theses of Men (2018), Afzal (2020), and Singh (2021) are subject to the terms of the Research Agreement between Find it EZ Software Corporation and the Governors of the University of Calgary made effective

on 1 November 2017. This thesis is subject to the terms of the Research Agreement between Find it EZ Software Corporation and the Governors of the University of Calgary made effective on 1 September 2022. In addition, all four students (including Shahryar Soltanpour) have agreed individually to non-disclosure and intellectual property agreements with Find it EZ Software Corporation. In a nutshell, we agree to not disclose confidential information of Find it EZ Software Corporation, and the title to all research results are the property of Find it EZ Software Corporation (including the software and other artifacts produced during these projects); we retain the right to use these artifacts for further research and to publish any research results arising from these research projects not divulging confidential information in the process. As such, we agree not to publish detailed accounts of the software created by each student, providing only high-level overviews that do not suffice to reproduce the software in question: this is closed-source software, as agreed to by all the parties. The four students received financial incentives for agreeing to these terms, and Professor Robert Walker received grant funding.

1.4 Thesis statement

The thesis of this dissertation is: (a) that our approach to extending ModCP results in a reduction in complexity, and therefore an increase in its maintainability and scalability; and (b) that our approach for identifying the language of a given string, compared to currently existing machine learning model, *file*, and the brute force baseline, shows superior accuracy and performance.

1.5 Thesis outline

The remainder of the thesis is structured as follows. Chapter 2 describes relevant literature and details of the development of ModCP, highlighting the issues encountered during implementation utilizing prior versions; it also describes scenarios in which the detection of the language of embedded strings and their relationship to the surrounding code is necessary, mentioning tools for detecting programming languages in strings. Chapter 3 examines the design and implementation of the new extension strategy in ModCP and the implementation of Crystal Reports using this method, as well as the implementation of the embedded language detection algorithm. In Chapter 4, measurements are presented to demonstrate the accuracy, performance, and overall quality of the code in comparison to previous implementations. Chapter 5 discusses the potential limitations and implications of the work. Chapter 6 concludes the thesis.

Chapter 2

Related Work and Background

In this chapter, we begin by presenting situations where it is necessary to identify and parse embedded code to determine its connection with the hosting code, as well as the tools used for detecting embedded languages. We also examine the versioning process of ModCP, outlining the changes made in each version and addressing any architectural issues that arise. Additionally, the chapter explores how our implementation of ModCP extends previous versions.

2.1 Embedded language occurrences

There are numerous instances within the C#, Crystal Reports, and Java languages in which SQL code is embedded within a string and interacts with the embedding code. In such scenarios, the ability to accurately identify the embedded language proves invaluable, as it allows for the selection of the appropriate parser and the execution of ModCP, ultimately resulting in the creation of a highly beneficial change propagation graph between the embedding and embedded code.

2.1.1 SQL embedded in Java

As shown in Listing 2.1, the function findPersonById(..) takes two input arguments, the id of the requested user and an instance of the Connection class to be able to connect to the database. The SQL query is embedded at line 2; lines 2–5 set the value of id in the query and fill the ResultSet variable with the query execution result. If a record with this id is found, the condition expression in line 6 will be **true**. Fields defined in line 2 are used in lines 8–11 to extract id, first_name, last_name, and age fields from the ResultSet object; these are then inserted into a Person object that is returned.

If it is possible to parse the string content within the rawQuery variable, a correlation can be established between the SQL query on line 2 of the code and the Java code on lines 8–11.

2.1.2 SQL embedded in Crystal Reports

The embedding of SQL within Crystal Reports represents a frequent use case within the code examples of our industrial partner; thus, the ability to accurately detect this occurrence holds significant industrial value.

Listing 2.2 is a simplified example of a Crystal Reports file containing embedded SQL. We can see that a variable named RenewalDate is defined in line 3 in the Crystal Reports code; this variable is used in the embedded SQL query in line 12 and is also used in the Crystal Reports formula in line 20. As we can see, these two code snippets share the use of this variable; we should be able to establish a connection between these in ModCP.

2.1.3 JavaScript embedded in HTML.

A prevalent example of embedded languages is the utilization of JavaScript within HTML to create dynamic web pages. In Listing 2.3, a header element is defined with id greeting in line 7, and a button is defined with id changeBtn in line 8. Line 12 in the JavaScript code tries to get the document with greeting id and line 15 tries to get the element with changeBtn id. By writing lines 18–21 we will update the text value of the header element whenever the button is clicked. If we update the id of elements in line 7 or 8, we should update the JavaScript codes at lines 12 and 15, respectively. Therefore, a change propagation tool can monitor modifications to these codebases, alerting the developer to the need for updates to one language when changes are made to the other.

2.2 Potentially useful means to identify embedded languages

In this section, we explore existing techniques for detecting embedded language code snippets. We begin by discussing the brute force approach and its pros and cons. Then, we delve into machine learning concepts and their applications. Next, we describe Guesslang, which utilizes machine learning, and Linguist, a tool from a related field, and finally, we will describe *file*, a Unix command used to determine the programming language of the given input.

```
1  public Person findPersonById(int id, Connection conn) {
2      String rawQuery = "SELECT id, first_name, last_name, age FROM sample_table WHERE id = ?";
3      PreparedStatement pstmt = conn.prepareStatement(rawQuery);
4      pstmt.setInt(1, id);
5      ResultSet rs = pstmt.executeQuery();
6      if (rs.next()) {
7          Person person = new Person();
8          person.setId(rs.getInt("id"));
9          person.setFirstName(rs.getString("first_name"));
10         person.setLastName(rs.getString("last_name"));
11         person.setAge(rs.getInt("age"));
12         return person;
13     }
14     else {
15         return null;
16     }
17 }
```

Listing 2.1: SQL embedded in Java.

```
1  Main Body
2      Report Parameters
3          Parameter:    @RenewalDate
4              Type:    DateTime
5              ListType:    Static
6      Tables
7          Table:    Command
8              Connection:    Connection #1
9              SQL Command:    SELECT * FROM
10                             DentalFee
11                             WHERE
12                             DentalFee_RenewalDate = {?@RenewalDate}
13             Parameters:    @RenewalDate
14
15     Formula Fields
16         Field:    lblMsgBody4
17             Formula:    WhilePrintingRecords;
18                         shared stringVar lblMsgBody4;
19
20                         lblMsgBody4 := REPLACE( lblMsgBody4, '<!--RenewalDate-->', CStr( {?
           @RenewalDate}, "d-MM-yyyy" ) )
```

Listing 2.2: SQL embedded in Crystal Reports.

```
<!DOCTYPE html>
<html>
<head>
  <title>DOM Manipulation</title>
</head>
<body>
  <h1 id="greeting">Hello, World!</h1>
  <button id="changeBtn">Change Greeting</button>

  <script>
    // Select the greeting element
    let greeting = document.querySelector('#greeting');

    // Select the change button
    let changeBtn = document.querySelector('#changeBtn');

    // Add a click event listener to the change button
    changeBtn.addEventListener('click', function() {
      // Change the greeting text
      greeting.textContent = 'Welcome to my website!';
    });
  </script>
</body>
</html>
```

Listing 2.3: JavaScript embedded in HTML.

2.2.1 Brute force

One potential solution for identifying the language of an embedded code snippet is a brute-force approach. This approach involves implementing the grammar of each language and testing the input string against all the generated parsers to identify any matches. If a match is found, the language of the input can be determined, while the lack of a match indicates that the input is not written in any language currently supported by the system. This approach provides 100% precision with no false positives. However, the process of running the input string against each grammar is resource-intensive and not easily scalable as the number of supported languages increases. In addition, we may need to develop and maintain different grammars for different versions of the same language which makes this approach more difficult to maintain and scale.

2.2.2 Machine learning

The field of machine learning investigates how computers can replicate human learning processes, acquire new knowledge and skills, recognize existing information, and continually enhance their performance. In contrast to human learning, machine learning can learn at a faster rate and rapidly accumulate knowledge, which

can be easily disseminated to produce tangible results. As a result, any advancements made by humans in machine learning can significantly improve the capacity of computers and thereby impact society as a whole (Rebala et al., 2019). Another important aspect of machine learning is its potential for knowledge transfer. Once a machine learning model has been trained on a particular dataset or task, it can be reused or adapted for other similar tasks or domains. This means that the knowledge acquired through machine learning can be easily shared, replicated, and scaled, leading to faster progress and more rapid innovation.

Supervised and unsupervised learning are two major types of machine learning approaches, each with its own strengths and applications. Supervised learning is a type of machine learning in which the algorithm learns to make predictions or decisions based on labelled data. In other words, the algorithm is trained on a dataset in which the correct answers or outcomes are already known, and it learns to make accurate predictions by identifying patterns in the input data that are associated with the correct answers. Unsupervised learning is a type of machine learning in which the algorithm learns to identify patterns and structures in unlabelled data; unlike with supervised learning, there is no predefined "correct" answer or outcome for the algorithm to learn from. Instead, the algorithm must discover hidden structures and relationships in the data on its own, and use this information to organize the data or make predictions (Dahiya et al., 2022).

2.2.3 Neural networks

Neural networks (Lawrence, 1993; Müller et al., 1995; Gurney, 1997) are computational models used in machine learning that mimic the structure and function of the human brain. They are composed of interconnected processing units, called neurons, which are organized in layers. Neurons receive input signals from the previous layer and apply a mathematical operation to these inputs to produce an output signal that is then transmitted to other neurons in the next layer. Each connection between neurons in a neural network is assigned a weight, which determines the strength of the connection. During the training phase, the weights are adjusted using an optimization algorithm based on the input–output pairs presented to the network. The goal of the training process is to minimize the difference between the predicted output and the true output for a given input. Once the weights have been learned, the neural network can be used to make predictions on new input data.

Neural networks have found applications in diverse fields such as science, medicine, and engineering, delivering state-of-the-art solutions in some cases. However, some researchers believe that neural networks have been used indiscriminately in situations where simpler methods could have been more effective, leading to a somewhat negative perception of them (Krogh, 2008).

2.2.4 Deep learning

Deep learning (LeCun et al., 2015; Goodfellow et al., 2016) is a technique that employs multiple layers of artificial neurons to learn complex patterns and relationships in data, thereby enabling accurate predictions or decisions in a variety of applications. Supervised and unsupervised learning methods are used in deep learning, depending on the task and dataset. Supervised learning involves training deep learning algorithms using labelled data to recognize and classify objects in images, to transcribe speech into text, or to translate between languages, while unsupervised learning involves learning useful features or representations of the data without explicit supervision or feedback.

One of the key benefits of deep learning is its ability to learn from large and complex datasets, such as images, videos, or natural language text, without requiring manual feature engineering. Deep learning algorithms automatically extract relevant features from raw data, reducing the amount of manual labor and expertise needed to develop effective machine learning models. Additionally, deep learning algorithms are able to generalize well to new and unseen data due to their ability to learn high-level representations of the data, making them well-suited for real-world scenarios that require robust and reliable performance.

Overall, deep learning has enabled significant advancements in a wide range of fields, including computer vision, natural language processing, robotics, and self-driving cars, making it a powerful and flexible approach to machine learning.

2.2.5 TensorFlow

TensorFlow (Abadi et al., 2015) is a free and open-source software library developed by Google for carrying out numerical computation and machine learning tasks. Its primary purpose is to assist in the creation of machine learning models, with a focus on neural networks, and to perform large-scale numerical tasks efficiently. At its core, TensorFlow provides a range of tools that enable the creation and execution of computational graphs, which are a way of representing mathematical operations as a directed graph. Each node in the graph represents a mathematical operation, while the edges between them indicate the flow of data between nodes. By utilizing the distributed computing capabilities of modern hardware, TensorFlow allows for the efficient execution of complex computational graphs. In addition to computational graph capabilities, TensorFlow includes high-level APIs that simplify the process of building neural networks and other machine learning models. These APIs provide pre-built components for common tasks, such as data preprocessing, model training, and inference, making it easier for developers to create complex models. With a large and active user community, TensorFlow has become one of the most popular machine-learning libraries in the world. It has found application in a wide range of fields, including image and speech recognition, natural

language processing, and robotics (Singh et al., 2020).

2.2.6 Linear classification

Linear classification is a valuable machine learning technique due to its efficiency in both training and testing procedures. One of its significant advantages is its ability to handle large-scale applications, making it a popular choice in many fields (Yuan et al., 2012). In linear classification, the goal is to separate data points into different classes by creating a linear decision boundary based on input features. The decision boundary can be a line or a hyperplane, depending on the number of features. By finding the best parameters for the linear function, the classifier can accurately classify new data points. Since linear classification has a simple structure, it is computationally efficient, making it particularly useful in large-scale applications. Additionally, linear classifiers are easy to interpret and explain, making them suitable for applications where interpretability is important.

2.2.7 Identifying the language of a given string

There is a limited amount of research in the field of identifying the programming language of a given string; however, there have been some notable open-source and industrial efforts in this area.

Guesslang

Guesslang is a program that is designed to identify the programming language used in a given source code. With its support for over 50 programming languages, Guesslang claims a detection accuracy rate of over 90% (Somda, 2021). Guesslang is the closest tool to our use case. The Guesslang model utilizes a TensorFlow-based deep learning architecture, which is reported as having been trained on a dataset of 1,900,000 unique source code files sourced from 170,000 public GitHub projects. The model is a combination of a deep neural network classifier and a linear classifier, tuned to gain both performance and accuracy.

The developers of Guesslang assert that their deep learning model demonstrates a high level of accuracy, specifically 93.45%, as determined through testing on a dataset of 230,000 distinct source files.

The primary limitation of the Guesslang model in its intended usage is its performance, or the time required to determine the programming language of a given string. The developers prioritize accuracy over performance, which can result in longer processing times. Additionally, they acknowledge that the model may make false guesses for code that is at the boundary between two languages. For example, a valid JavaScript source code is also a valid TypeScript source code. Listing 2.4 shows a code that can be executed by both Typescript and JavaScript interpreters without any errors.

```
function greet() {
    return 'Hello, world!';
}

try {
    const greeting = greet();
    console.log(greeting);
} catch (error) {
    console.error(error);
}
```

Listing 2.4: A valid code for Javascript and Typescript.

2.2.8 Linguist

Linguist (GitHub, 2022) is used to determine the breakdown of programming languages used in a GitHub repository. It is implemented by GitHub and the statistics it generates are displayed on the front page of each GitHub repository. This tool is aimed at allowing users to quickly understand the dominant languages used in the project and the makeup of the codebase.

To use Linguist, one must define a list of languages and give it as the input to Linguist. Then it tries to exclude the files that seem to be auto-generated, analyzing the remainder. It also permits the user to override its predefined steps. It generates the aforementioned language bar that is shown on the first page of each GitHub repository.

The primary limitation of Linguist is its reliance on file extensions as a means of identifying programming languages. This method proves to be inadequate in situations where the input is in the form of a literal string, since there is no file extension in such cases, or in languages embedded within others since the extension would only apply to the host language. This renders Linguist ineffective in determining the presence of embedded languages within other languages. We, therefore, do not consider it further.

2.2.9 Unix *file* command

Another tool that aims to detect the type of an input string is the Unix *file* command (Darwin, 2011). The *file* command reads the patterns of a given input stream and searches the patterns inside its database for different types of files. Based on the patterns found, it returns the most probable guess for the type and the programming language of the input. The major advantage of this command over Linguist is that it does not consider the extension of the given file, which is useful in our use case. This tool supports a variety of types of files, from multimedia and text files to some programming languages. The main GitHub repository for the *file* command (File, 2011) does not provide support for C# files, but it does provide support for Java files. Based on this, we can compare the accuracy and performance of this command with other approaches.

2.3 Modelling heterogeneous software systems

Software development and analysis tools (SDATs) are used by software developers for the purpose of creating, modifying, and maintaining software programs. The process of software maintenance involves making modifications and updates to systems in order to enhance performance, address defects, and meet emerging requirements. Maintenance is an essential aspect of the software development lifecycle and is known to become increasingly costly and complex as the codebase grows. Research suggests that software maintenance may account for as much as 90% of the overall cost of software ownership (Dehaghani and Hajrahimi, 2013). A *heterogeneous software system* is a system that is composed of multiple subsystems that are interdependent and interact with each other, with each subsystem potentially being coded in a different programming language. This is a common occurrence in the development of various types of software systems, including embedded systems, mobile applications, web applications, and J2EE patterns, where multiple technologies and languages are utilized (Mushtaq and Rasool, 2015).

In accordance with recent trends, applications have increasingly transitioned from utilizing a single language to utilizing a combination of multiple languages (such as Java, XML, SQL, etc.) (Chikofsky and Cross, 1990) and various technologies. The prediction of change propagation within source code represents a critical challenge in the analysis and maintenance of multilingual enterprise applications (Aryani et al., 2011). Before any maintenance work can be done, it is important to fully understand the systems involved; this understanding often takes up the majority of the total time and effort needed for maintenance, often ranging from 40 to 90 percent (Strein et al., 2007).

Various methods exist for modelling heterogeneous software systems in order to analyze them. These models aid in identifying entities within each language and representing their relationships with one another. The selection of the optimal approach for modelling largely depends on the researcher or developer's specific needs and use case. In the following, we will discuss two of the most commonly utilized methods for modelling heterogeneous systems.

2.3.1 Universal model approach

In this methodology, a single model is employed to depict all entities and their relationships across all languages. The benefit of this methodology is that it simplifies the development process by saving resources and time; however, the conversion of a language from its original model to the universal model may result in the loss of certain language-specific properties.

One methodology that employs this model is FAMIX (Synytskyy et al., 2003), which utilizes a language-independent meta-model for refactoring code written in object-oriented languages. However, a limitation

of FAMIX is that it has not undergone extensive testing across all languages and further experimentation across a broader range of languages is identified as an area for future research. The FAMIX model forms the fundamental aspect of the Moose platform for software analysis (Nierstrasz et al., 2005).

MoDISCO is an open-source tool that helps developers cope with legacy code by converting legacy code into meta-models to describe them. This tool is considered a suitable tool for small and medium-sized projects but it needs more evaluation data on industrial projects (Harman, 2010).

Moise and Wong (2005) used GXL (Graph eXchange Language) schemas to represent languages such as C/C++, Java, Tcl, Fortran, and Cobol. Their model is designed in a way to contain all needed features for all languages, but some features are not supported in all languages. For example, their model supports macros that are only available in C/C++. They also believe that their tool can be expanded more in control and data integration mechanisms.

2.3.2 Multiple models approach

An alternative method for representing different languages is to utilize a model specific to each language. This can also be achieved by grouping closely related languages and utilizing a single model for that grouping. The benefit of this approach is the ability to retain a higher level of detail compared to the previously mentioned methodology; however, the disadvantage is that implementing a new model for each new language requires a significant investment of time and resources.

Tools such as X-DEVELOP (Strein et al., 2007) group languages based on their similarities and then provide a model to represent each group. However, X-DEVELOP lacks support in generalization, dynamic language contents, upcoming languages, and low-level languages (Schink, 2013).

DATRIX (Lapierre et al., 2001) is developed by The Datrix team within Bell Canada. It uses a language-specific model that can be extended to new languages only with difficulty, due to the presence of many language-dependent properties (Strein et al., 2007).

COMPOST (Ludwig and Heuzeroth, 2001) is the implementation of a metaprogramming model in which they try to provide an approach to adapt software systems with continuously changing requirements. Metaprogramming is a programming technique that empowers computer programs to treat other programs as data, granting them the ability to possess self-awareness and manipulate themselves in various ways. It encompasses a range of methods through which a program can acquire knowledge about its own structure or modify itself dynamically. (Czarnecki and Eisenecker, 2000). In our case what ModCP does can be considered as metaprogramming since it parses and analyzes the given code.

The utilization of a universal model approach has witnessed growth, aligning with the rise of multi-

language programming practices. However, it is worth noting that many existing resources and tools employing multi-model representations are outdated and may not adequately cater to current needs.

2.4 ModCP version history

The ModCP framework is a change propagation system developed within the Laboratory for Software Modification Research at the University of Calgary, in collaboration with Find it EZ Software Corporation. To date, various modifications and revisions have been implemented to the ModCP codebase, which will be discussed in detail, in order to reach the latest version before our work.

A total of four versions of ModCP were previously created, and this thesis introduces a fifth one. Our objective is to label each version so that it can be easily distinguished throughout the remainder of this thesis. In Figure 2.1, a solid line between two nodes shows that the source node was developed from the target node, either as an extension or refactoring thereof as per the label, and a dotted line shows that the target node re-implements the novel features from the source node ("influenced by"). We see that Men (2018) has implemented the initial version of ModCP (labelled Men's version). Afzal (2020) and Singh (2021) have implemented their own versions based on Men's version, which we label as Afzal's version and Singh's version, respectively. Professor Walker refactored the code from Men's version resulting in a heavily restructured code base, which we refer to as Walker's version. In this thesis, we describe our re-implementation of the features of Afzal's version and Singh's version atop Walker's version; we refer to the resulting, novel version as Soltanpour's version.

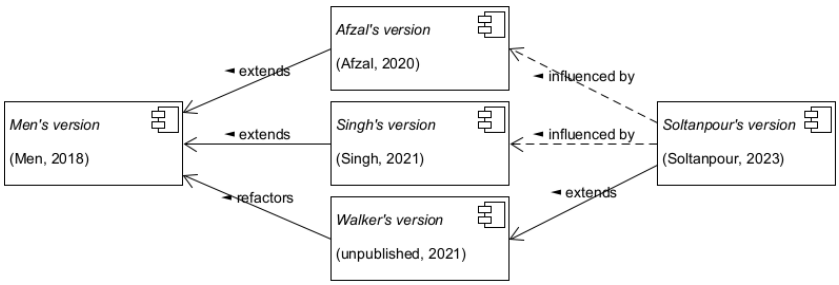

Figure 2.1: Versioning of ModCP.

2.4.1 Initially: Men's version

As put forth by Men (2018), ModCP is the SDAT that is used to identify change propagation in a given single or multi-language system and assist programmers in their daily development tasks. Men asserts that the model is capable of effectively handling large code bases while maintaining efficiency in terms of memory

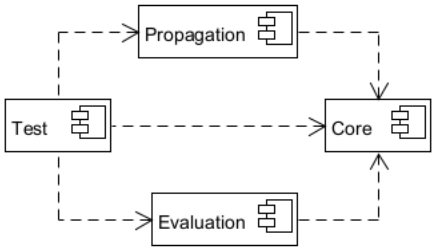

Figure 2.2: Projects and their relationships in Afzal's version of ModCP.

usage and performance. ModCP uses a technique called program slicing (Weiser, 1981), a method for simplifying programs by identifying and removing unnecessary code based on the values of certain variables, referred to as the *slicing criterion*. The resulting simplified program is referred to as a *program slice*. This technique has been extensively studied and has been applied to a variety of fields since its proposal.

However, the initial implementation of ModCP did not offer support for multiple programming languages, only supporting the analysis of programs written in Java.

2.4.2 Adding support for multiple languages: Afzal's version

Afzal (2020) proposed a method for change propagation by utilizing a dependency analysis model and creating a standard meta-model referred to as the "unified meta-model." This approach utilizes abstract syntax representation models as modular components for each language, which can be integrated into the meta-model for any new languages that are introduced.

In Afzal's version, the ModCP codebase contains a Microsoft Visual Studio solution, and the solution contains four projects, as illustrated in Figure 2.2 as a component diagram in Unified Modeling Language (the arrows represent dependencies and the boxes with the "plug" icon represent projects). The *Core* project keeps the parsers and grammars of each language and the model classes to represent the abstract syntax tree (AST) of the given program. The grammars are compatible with ANTLR 4 (Parr, 2013), and ANTLR is able to generate the C# parser for the grammar file. Therefore, the *Core* project contains C# parser files for other languages as well. The other projects in this solution are: *Evaluation*, which is responsible for performing an empirical evaluation of ModCP compared to more traditional alternatives; *Propagation*, which is responsible for generating call graphs and data graphs for the given AST; and *Test*, which is responsible for testing the functionalities in the other projects.

2.4.3 Adding persistence to ModCP: Singh's version

According to Singh (2021), the computation of ModCP's complex models is excessively demanding, rendering it impractical to recalculate them each time the system is restarted and resulting in undesirable downtime in the developer's daily workflow. As a result, Singh sought to identify the optimal strategy for persistently storing and restoring the models as needed. To this end, they evaluated various data persistence tools and selected the one most suitable for ModCP use cases. Initially, the author considered databases such as Python-CSV, MySQL, Neo4j, and PostgreSQL, but later determined that object serialization methods may be more efficient for this purpose. The evaluated object serialization methods included BinaryFormatter, DataContract, NewtonSoft JSON.NET, and Protobuf-net. Ultimately, the authors determined that Data-Contract serialization technology was the most cost-effective in terms of the time required to execute the processes for storing and retrieving the model and required the least amount of disk space to store the translated models.

2.4.4 Redesigning a universal model: Walker's version

Following the work of Men (2018), Professor Robert Walker refactored the ModCP codebase with the aim of decoupling language models from one another. Specifically, he redesigned the architecture such that a base abstract model, comprising interfaces and abstract classes, is located within the *Core* project, with all slicing algorithms dependent on this base model. Additionally, he created another project, named *Extensions*, in which models and parsers for specific languages can be implemented in a mutually independent manner by extending the base model in the *Core* project. This design adheres to the universal model approach for representing different languages and conversely, the *Extensions* project comprises distinct languages that are not incorporated into the universal model of the *Core* project. Figure 2.3 shows the projects' dependencies after adding the *Extensions* project and moving language-related files to this project.

Figure 2.4 represents the packaging of models of each language in the refactored version and how they extend the base interfaces of the *Core* model. As represented, each model is separated from other languages and can have its own language-specific classes, such as ImportStatement in Java and UsingStatement in C#.

Limitations

The limitations of Walker's version include:

- The base model defined in the Core project may not be able to provide a sufficient base model for all languages. Some languages may not have concepts such as statements, expressions, etc. but they have to extend the Core models and fill them with some irrelevant details.

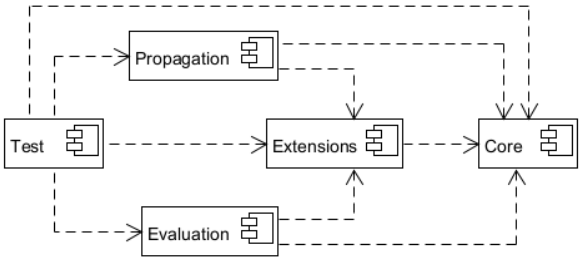

Figure 2.3: Project structure after introduction of the *Extensions* project.

- The isolation of each language inside *Extensions* project increases code duplication, especially for similar languages. For instance, C# and Java have lots of similarities but they cannot reuse each other's models since they are supposed to be kept isolated from each other. However, strictly speaking, Walker's version does not require this; in principle, one could have a hierarchy of languages, sharing their commonalities. We chose the route of strict language separation to avoid the potential, added complexities of managing such a hierarchy.

- The design of ModCP constrains the abstract syntax trees arising from languages to involve a class hierarchy for different node types. The alternative of using a single class for nodes would require either encoding the type information in a different way (replicating the support for polymorphism within the implementation language, C#) or expansion of the polymorphism into nested if-expressions each of which would likely need to repeat complex checks (such as determining the relative location of a node within its surrounding structures). Since pursuing one of these alternatives would have required extensive re-implementation of the ModCP code base, we do not see this as a realistic alternative presently.

2.5 Integrating Afzal's and Singh's versions into Walker's version

After Prof. Walker's refactoring, it was necessary to port the novel features of Afzal's version and Singh's version into Walker's version. The lessons learned from Singh's version were relatively easy to port, as described in Section 2.5.1. Integrating Afzal's version turned out to be far more complicated and necessitated completely re-implementing the extensions, leading to the opportunity for much of this thesis; the issues encountered are detailed in Section 2.5.2.

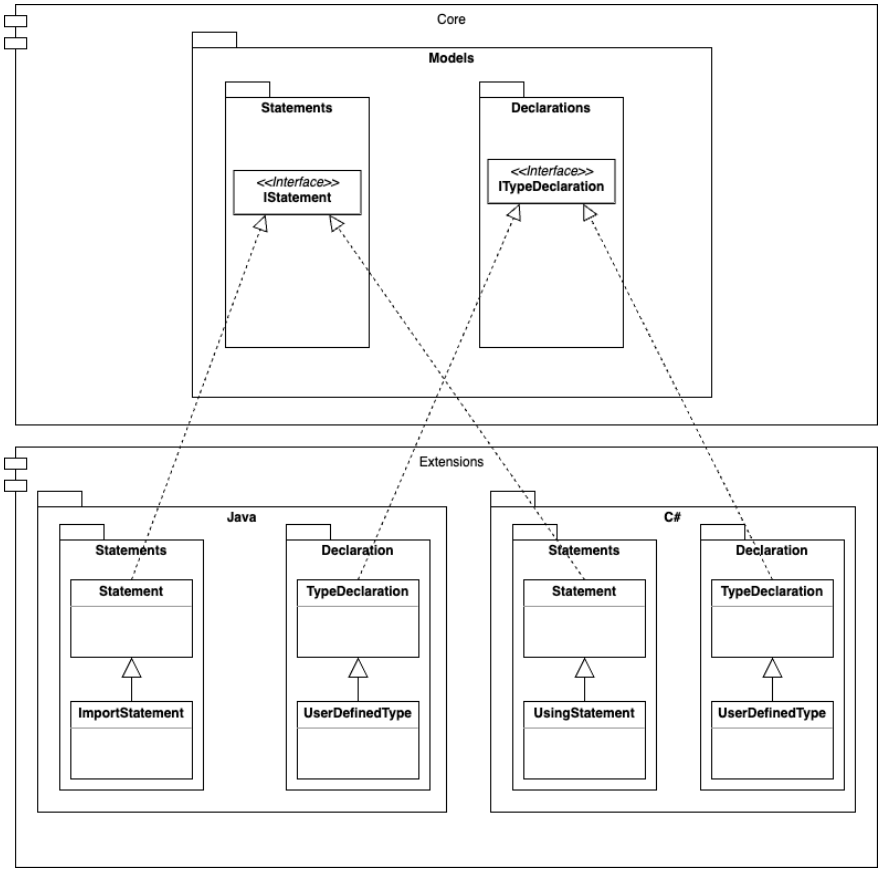

Figure 2.4: Model packaging and inheritance in Walker's version.

2.5.1 Porting the extensions of Singh's version atop Walker's version

Enabling data persistence is essential for Walker's version. To achieve this, the approach suggested by Singh (2021) needs to be ported; this involves annotating each class and its parents with special annotations to allow for serialization and deserialization of all program classes, which will be applied to all model classes in the Core and Extensions projects.

2.5.2 Porting the extensions of Afzal's version atop Walker's version

In the software architecture of Afzal's version, all the classes used to represent ASTs are stored inside the *Core* project and there is a single model representing all languages. The Model directory contains other directories such as Declaration, Expression, Statement, etc., and inside each of these directories, there are classes that represent various kinds of that entity. For instance, inside the Statement directory, we have classes such as AssertStatement, BlockStatement, etc. In the case of statements, we have three kinds of classes: (1) those that are shared between all languages, residing in the root directory of Statement; (2) those that are only related to procedural languages including Java, C#, and Python, reside inside the GPP directory; and (3) those that are related to the Transact-SQL language that reside in the TSql directory; this division exists for directories other than statements as well. Classes in the GPP and TSql directories extend classes in the root of the directory. In addition, in some cases, a class in TSql extends a class in GPP while we expect these directories not to be dependent on each other.

Inside the GPP classes, the codebase has to support four languages, C#, Java, C, and Python. The fact that all these languages use the same classes makes the classes much more complex; remembering that some of these classes are extended by T-SQL makes this structure yet more complex.

Problems with the architecture of Afzal's version

We identify various issues within the ModCP codebase of Afzal's version that have hindered development, maintainability, and traceability, leading to the consideration of alternative methods for orchestrating diverse languages and aligning the codebase with fundamental principles of software engineering.

- **Unused methods and attributes for some languages.**

  Each language possesses its own special concepts that may or may not be present in other languages. When multiple languages use the same model, those concepts must be defined in all of them. As an example, the method shown in Listing 2.5 is used by all languages in Afzal's version. This method is defined inside the Method class in the language model. At line 2, the code checks whether this instance is a constructor or not, has a parent super class, and is static or not; all of these concepts are defined

```
1    public CallSite GetCallSiteBySelf() {
2        if (!method.IsConstructor() && !hasSuper && !method.IsStatic()) {
3            if (method.Language == "tsql" || method.Language == "python") {
4                poly = false;
5            }
6            else {
7                poly = true;
8            }
9        }
10   }
11
```

Figure 2.5: An example of special cases for handling Python and T-SQL.

in an object-oriented language such as Java, but when they are used to represent a non-object-oriented language like T-SQL, they become irrelevant values. Therefore, despite the fact that T-SQL does not support the concepts of constructors, polymorphism, and static methods, its model still implements the IsConstructor() and IsStatic() methods, and the fields hasSuper and poly.

- **Inheritance problems.**

  In 30+ instances within the ModCP codebase, it is observed that a class in the T-SQL language extends a class from the GPP directory. For example, the class BatchDeclaration from T-SQL extends the class ProcedureDeclaration in GPP. When extending an existing class, it is crucial to evaluate whether an "IS-A" relationship exists between the two classes and to ensure adherence to the Liskov Substitution Principle (LSP) (Noback, 2018), asserting that objects of a superclass should be able to be replaced with objects of its subclasses without disrupting the functionality of the application. This principle necessitates that the objects of subclasses exhibit the same behavior as the objects of the superclass. In the context of TSql, this extension does not conform to this principle, as it is not a specialized version of C#, Java, or Python, and therefore does not align with the fundamental principles of software engineering.

  Figure 2.6 shows another example of these complex inheritance relations between classes. In this example, we have a class named Declaration in the GPP directory and TSQLDeclaration in the T−SQL directory. The classes in these directories do not follow any particular rule for inheritance. In GPP, some classes like TypeDeclaration extend the general Declaration class suggesting that Declaration must be the parent of all declarations. Conversely, we can see that ProcedureDeclaration does not extend Declaration. Similarly, in the T−SQL directory, there is a class called TSQLDeclaration that presumably should be the parent of all classes in that directory. While DatabaseDeclaration follows this rule, BatchDeclaration breaks it by extending ProcedureDeclaration from GPP.

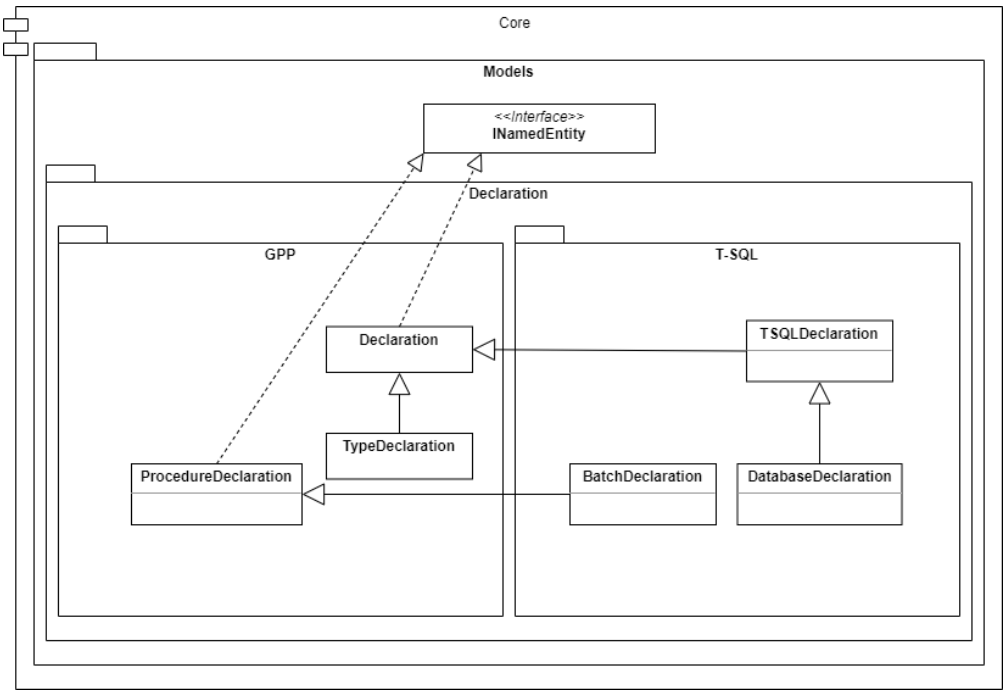

Figure 2.6: An example of complex inheritance relationships.

- **Adding a new language to ModCP.**

  Given the details above about how models are orchestrated inside ModCP, design decisions about adding a new language would not be deterministic. There can be multiple possible ways to add a new language.

  1. **Add the new language in GPP**.

     To add a new language to the GPP directory, we would have to deal with more complex conditional statements, like the one in Listing 2.5. As a result, when we add support for more languages in our codebase, the number of these complex conditional statements increases; as they are redundant and repeated in different classes, the developer should be careful to update all of them if needed while adding a new language, which makes maintenance more time-consuming.

  2. **Completely isolate the newly added language**.

     It is possible to fully isolate our language model from other language models, thereby minimizing the impact of languages on one another. However, this approach necessitates the duplication and independent implementation of all functionality present in ModCP, as it currently only supports GPP and TSql. This would require the creation of distinct versions of ModCP for each added language, resulting in an inefficient approach with regard to development speed, code reusability, and maintainability.

3. **(Hybrid) Create a new directory and extend GPP.**

   This methodology resembles the approach employed in the case of the TSql directories. While this approach facilitates development by enabling the utilization of GPP classes at will and the incorporation of personalized logic as needed, it also presents the drawback of making tracing and debugging significantly more challenging, as it becomes difficult to determine the origin of functionality, whether it be from GPP or TSql. Additionally, as the number of languages developed utilizing this methodology increases, the associated maintenance costs will also experience a rapid increase.

2.6 Summary

In this chapter, we discussed the subject of software development analysis tools and the two prevalent models for representing language models, namely the universal-model and the multiple-models approaches, providing examples of their usage and highlighting their advantages and disadvantages. We also provided an overview of the evolution of ModCP, tracing its development from its inception to the current version. The architecture of the ModCP codebase was outlined and examples of its shortcomings and the resulting challenges for code maintenance and development were provided.

Additionally, the topic of embedded code, where code written in one language is integrated with code written in another language, was examined. Additionally, we mentioned notable related works in the field of embedded language detection, including Guesslang, Linguist, and the Unix *file* command, and identified their limitations, which motivated the development of a new embedded language detection system within ModCP.

Chapter 3

Approach

This chapter presents a comprehensive description of the implementation process of Soltanpour's version of ModCP. We outline the integration of support for previously supported languages and also the addition of Crystal Reports, a new language not previously supported in earlier versions of ModCP. Our implementation also includes the application of persistence, as previously described in Singh (2021), to the refactored architecture of ModCP (Walker's version). Furthermore, we discuss the method employed for the detection of embedded languages and describe the evolution of the implementation process, from the initial brute force approach to the implementation of a more accurate and efficient keyword detection approach.

Specifically, I have implemented Crystal Report AST, builder, evaluation, and tests alongside C# AST, builder, and tests. In addition, I have implemented T-SQL AST, builder, and tests. I also have ported the functionality of serialization in the current version and I have implemented the embedded language brute force and keyword detection approaches. In terms of git stats, I have made 152 commits on ModCP with 3900 files and 1,540,450 line modifications.

3.1 ModCP Extension

In Walker's version of ModCP, language-related models moved to the *Extensions* project, leaving the *Core* project to contain only the abstract model; a model to support Java is already implemented in the *Extensions* project.

3.1.1 Implement C#

Figure 3.1 shows the directory hierarchy of C# in the *Extensions* project. The C# directory contains two main directories, Parser and Model. Parser contains the CSharpLexer and CSharpParser classes which are auto-

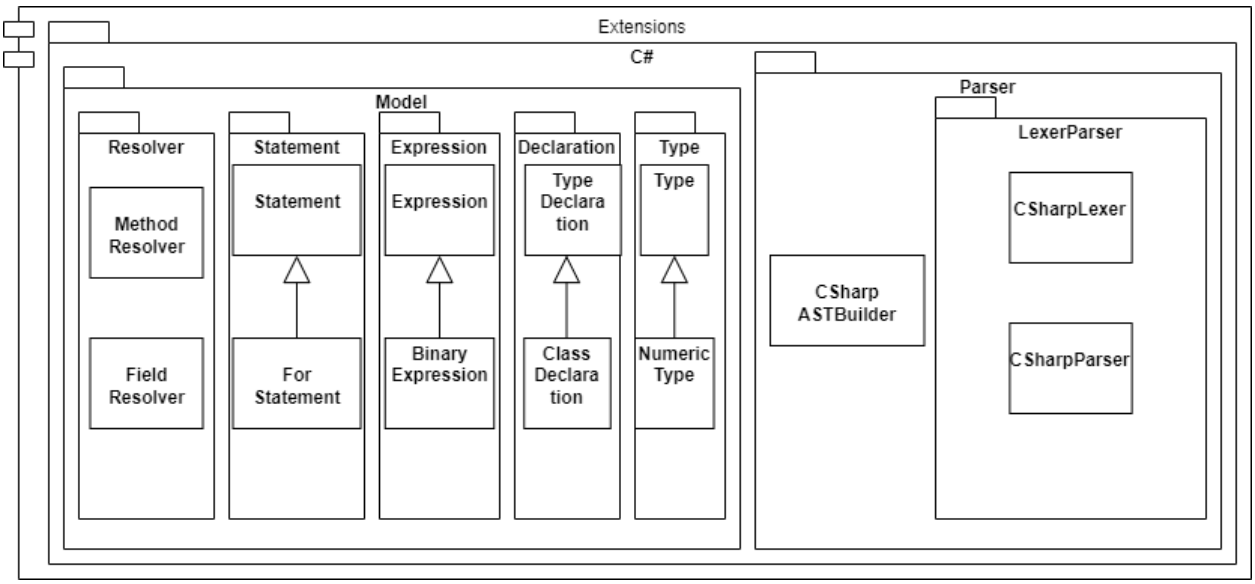

Figure 3.1: C# Directories in Soltanpour's version.

generated based on the given ANTLR 4 grammar. The CSharpASTBuilder class is responsible to translate files processed by the lexer and parser to the AST models in the Model directory.

The first step to implement the support for C# is to create a parser for C# code. We use GitHub (2021) to get an ANTLR 4 grammar for C#. ANTLR supports generating lexers and parsers for C# code; since ModCP is written in C# as well, this simplifies the integration of the parser into ModCP. Using the pre-build commands feature of Visual Studio, we generate the C# parser from the given grammar each time the *Extensions* project is built. The next step is to develop a model to represent the abstract syntax tree (AST) of parsed C# code. Since Java syntax is similar to C# in most cases, we leverage the experience expressed in the Java models in constructing the equivalent C# ones.

The Resolver directory contains some helper classes and methods; their main responsibility is to get a type, field, or method name and try to find its definition, return type, and input arguments. The classes within this directory mainly inherit from AbstractDiagramVisitor from the *Core* project.

The statement directory models statements such as **for**, **if**, **while**, block, etc. in the AST. There is a class in this directory named Statement that serves as the base class for all other statement classes, extending the IStatement interface from the *Core* project.

The expression directory models various expressions that may appear in the code, with "binary expression" being one of the most common. Likewise, the directory includes a root class named Expression that serves as the parent class for other expression classes and extends the IExpression interface from the *Core* project.

The declaration directory is used to model the declaration of methods, classes, fields, anonymous classes, functions, or other types. The main class in this directory is TypeDeclaration, which serves as the base class

and extends the ITypeDeclaration class from the *Core* project.

The operator directory is responsible to reflect various types of operators that are defined in the language, such as Binary operators.

Lastly, the type directory contains definitions for the various types that may exist in the language, ranging from primitive and built-in types like Integer and Boolean to user-defined types such as classes. The main class in this directory is simply called Type and extends the IType interface from the *Core* project.

To integrate the parser generated from the grammar into the extended core model for C#, we create a set of ASTBuilder classes. Each builder will contain an instance of the generated lexer and parser, and its role will be to use the parser on given code, then to traverse the parser's output in order to construct the abstract syntax tree (AST) based on the defined model. It is crucial to update the grammar during this step to ensure all desired information is extracted and properly transferred to the model.

After completing the implementation of the models, builder, and parser, we should write both unit tests and integration tests to validate the correctness of each component individually and also in combination with each other. For guidance on how to set up these tests and what functionality to test, we took inspiration from the Java tests.

Some of the most important functionalities that we want to test include:

- Control Flow Graph (CFG) Tests: To confirm the accuracy of the control flow graph, tests are performed on various types of statements such as **for**, **while**, **if**, **switch**, etc. These tests verify that the graph is generated correctly for all potential combinations.
- Anonymous Class Test: The aim of this testing is to confirm that ModCP correctly identifies inner and anonymous classes and has the capability to resolve their types.
- Call Graph Test: This testing validates if the call graph between functions and classes has been accurately detected and created.
- Call Site Test: Each statement in the model has a collection of call sites, representing the list of functions invoked within that statement. Call site tests evaluate the accuracy of this list.
- Expression Test: This testing compares the program against complex expressions to evaluate parsing accuracy.
- Parse Test: This testing evaluates the parser for corner cases such as classes with the same name.
- Program Dependence Graph (PDG) Test: This testing validates the integration and collaboration of all components from parsing to graph creation to ensure the main flow of the program, which is the creation of a program dependence graph, is executed correctly.

In total, we created 139 automated test cases for C# and made sure all of them pass.

Main challenges in C# development

The implementation of the parser for C# posed several challenges, including differences in naming conventions and language features compared to Java, such as the use of "string" instead of "String" and the existence of "partial classes". Additionally, there were discrepancies in the behaviour of methods, such as "substring(Integer, Integer)" between the two languages. When using Java's "substring(start, end)" function, it retrieves a substring that begins at the START index and concludes at one index prior to the specified END. Conversely, in C#, you can achieve the same outcome by invoking the "substring(start, start − end)" method.

These challenges required extensive debugging and tracing. Furthermore, it took additional time to become familiar with the code and develop and write tests for C#. Lastly, there were also issues with the proper versioning of ANTLR and setting up the pre-build command to generate the C# output from ANTLR.

3.1.2 Implement T-SQL

Following the successful implementation of C#, we added support for Transact-SQL to ModCP. This language has been prioritized as it holds significance for our industrial partner.

The steps followed for T-SQL are similar to the ones we followed for C#. We utilized the ANTLR grammar available from GitHub (2021) and added a new command to our pre-build process to generate the T-SQL parser files using ANTLR.

The next step is to generate the AST models to represent T-SQL codes. Building on the work of Afzal (2020), much of the T-SQL directory could be reused, but significant modifications were necessary to adapt the models for use with ModCP. The structure of the directories for the Model directory remained consistent with C#, although the specific classes within them were designed specifically for T-SQL and were not present in other languages. Examples of these language-specific classes include BatchDeclaration, ProcedureDeclaration, SelectStatement, and InsertStatement.

Once the model and parser are implemented, the next step is to write the ASTBuilder classes that iterate through the parsed data and instantiate the models to construct the abstract syntax tree (AST). The general behaviour of the builder classes is similar to what we developed for other languages and it was not a significant challenge during the development process.

Lastly, we wrote tests to examine the functionality of the T-SQL parser, the tested functionalities being

mostly what was previously covered by the C# tests. We wrote 40 tests that successfully ran on T-SQL.

Main challenges in T-SQL development

Finding adequate examples for testing purposes proved to be a challenge, as a limited amount of examples were available for the desired scenarios. To address this, we created custom T-SQL code that, while not typical, was necessary for testing various components of ModCP. Some test cases were also eliminated as they were not relevant to the context of T-SQL.

3.1.3 Data Contract implementation

Following the refactoring conducted by Prof. Walker, it was necessary to implement the solution proposed by Singh (2021) to enable data persistence in the updated version of ModCP.

To add the ability to serialize and deserialize all classes in the program, we needed to annotate each class with the annotations specified in Listing 3.1. By adding these, we mark a class as one that can be serialized by DataContract. If the class has a parent class, we should add the same annotations to the parent class as well, and if it does not have any parent classes, we have to add the type of the class to our serialize and deserialize functions like what is done in Listing 3.2. If we add annotations to two classes Parent and Child in which Parent is the parent of Child, in addition to adding the code from Listing 3.1 to both classes, we should add that in Listing 3.3 to the class Parent as well. In this case, there is no need to add Listing 3.2 for the Child class.

For each field of the class that we are going to serialize, we should add annotations to the field, as exemplified in Listing 3.4. Note that the *Order* number should be unique for each field within a given class.

```
1 [System.Serializable]
2 [System.Runtime.Serialization.DataContract(IsReference = true)]
```

Listing 3.1: Annotations for serializing a class.

```
1 DataContractSerializer serializer =
2     new DataContractSerializer(typeof(ModularProgramDependenceGraphProject),
3     new System.Type[] { typeof(Namespace.ClassA), typeof(Namespace.ClassB)} );
```

Listing 3.2: Introducing a serializable class to Data Contract.

```
1 [System.Runtime.Serialization.KnownType(typeof(Child))]
2 public class Parent {}
```

Listing 3.3: Additional annotatations for the parent class.

```
1 [System.Runtime.Serialization.DataMember(Order = 2)]
2 public int Line { get; private set; }
```

Listing 3.4: Additional annotatations for the parent class.

Usually, we increment the number for each field, starting from 1.

DataContract in the *Extensions* Project

To begin the process of persisting models in the *Extensions* project, we add DataContract serialization annotations to each model and field, making them serializable. If an annotation is forgotten for a class, the code will throw an exception while it tries to serialize or deserialize a class with that type expressing that the serialization of this class is not supported. It was crucial to exclude auto-generated Lexer and Parser classes from the annotations, as they are regenerated every time the program is built, which would delete any manually added annotations. To be more specific, each time that we use Visual Studio to compile the project, If there is no change in the whole *Extensions* project and the projects it depends on, they won't be regenerated. But if there are any changes in *Extensions* project or any project it depends on, all parser classes will be regenerated.

Additionally, since the data stored in these classes is not useful, there is no need to persist it.

Data Contract in the *Core* Project

When we annotate a class as serializable, we have to annotate its parent classes as well. As a result, we reach the models in the *Core* project and we have to annotate those basic models. We have followed the same process for all other languages to make sure all of the models can be serialized properly.

Testing Data Contract Implementation

In order to verify the success of the annotation process, we implemented a function that serializes and deserializes a given model. This function was added to various tests in each language immediately after the creation of the model and prior to processing it. By doing this, we were able to identify any classes or fields that may have been overlooked during the annotation process. Additionally, it helped to ensure that the models remained unchanged before and after serialization.

3.1.4 Implement Crystal Reports

Implementation

Prof. Walker developed the ANTLR grammar for Crystal Reports, on the basis of user documentation for Crystal Reports and a large body of examples provided by Find it EZ; this served as the foundation for our project. Since Crystal Reports has a distinct structure from other supported languages, we were unable to reuse much of the existing code. The Crystal Reports files consist of nested key–value pairs, and our grammar is able to parse the values. Therefore, we incorporated pre-existing Java code, also developed by Prof. Walker, to parse the files to a key–value dictionary. We translated this code to C# and integrated it into our Crystal Reports builder. With this implementation, we can utilize our ANTLR grammar to parse each value.

Our development process involved creating models and builder classes concurrently. We iteratively refined the builder classes, extending the necessary models from the *Core* project whenever a new model was required. Through this iterative process, we successfully implemented the models and types needed to accurately represent the Crystal Reports codes.

Testing

Once we completed an initial version of the Crystal Reports parser, we began writing tests to identify and address potential bugs that might arise during the development process. Similar to our approach with previous languages, we created tests for the control-flow graphs, call graphs, program dependence graphs (PDGs), PDG slicer, resolvers, and parser, all of which were covered for Crystal Reports. A significant portion of the development and debugging process took place during the testing process.

Evaluation

Alongside our effort to ensure the correct functioning of ModCP with Crystal Reports through testing, we devised evaluation scenarios and tests to gauge the recall and precision of the ModCP change propagation detection in detecting dependencies in this new language. We calculated the recall and precision for each test case scenario, which will be discussed in further detail in Chapter 4.

Main challenges of Crystal Reports development

The implementation of support for Crystal Reports was the most challenging task among the languages covered in this thesis. Initially, we had to familiarize ourselves with the language's syntax, usage, and coding

conventions since it was a new and unfamiliar language to us. Crystal Reports posed unique parsing challenges compared to the other languages because it employed a key–value structure in each file, necessitating a different parsing procedure. Furthermore, the development of models and builders, in this case, was distinct from that of the other languages, rendering any prior knowledge or previous examples obsolete. Finally, we faced difficulties in finding reliable, publicly accessible sources of Crystal Reports code, compelling us to rely solely on examples from our industrial partner.

Upon completion of the language extension for ModCP, a simplified version of the model is depicted in Figure 3.2. This diagram serves as an illustrative example of how classes from various languages now extend the base model in the *Core* project while remaining isolated from each other in the *Extensions* project. It should be noted that this structural diagram only represents a sample of the new model architecture and does not include all of the structural details.

3.2 Embedded language detection

In this part, we address the issue of identifying embedded language strings. Specifically, we aim to develop a system that can take string input and determine the programming language in which the input is written.

3.2.1 Brute force approach

One potential solution is a brute force approach, which involves implementing the grammar for each language that we want to support. When given a string input, we would then run all of the generated parsers against the input to check for matches without errors. If a match is found, we can determine the language, but if no match is found, we conclude that the input is not written in a language currently supported by the system.

Complications in this situation include the need to choose an order in which to try the parsers and the question of whether a single positive result will suffice or if all languages for which the string is a valid code snippet should be reported. In our SDAT context, a single positive result will be needed so the process can end as soon as one is found.

Advantages

This approach guarantees 100% precision, meaning there will be no false positives.

Disadvantages

The drawback is that the process of running the given string against each grammar is expensive and not easily scalable as the number of supported languages increases.

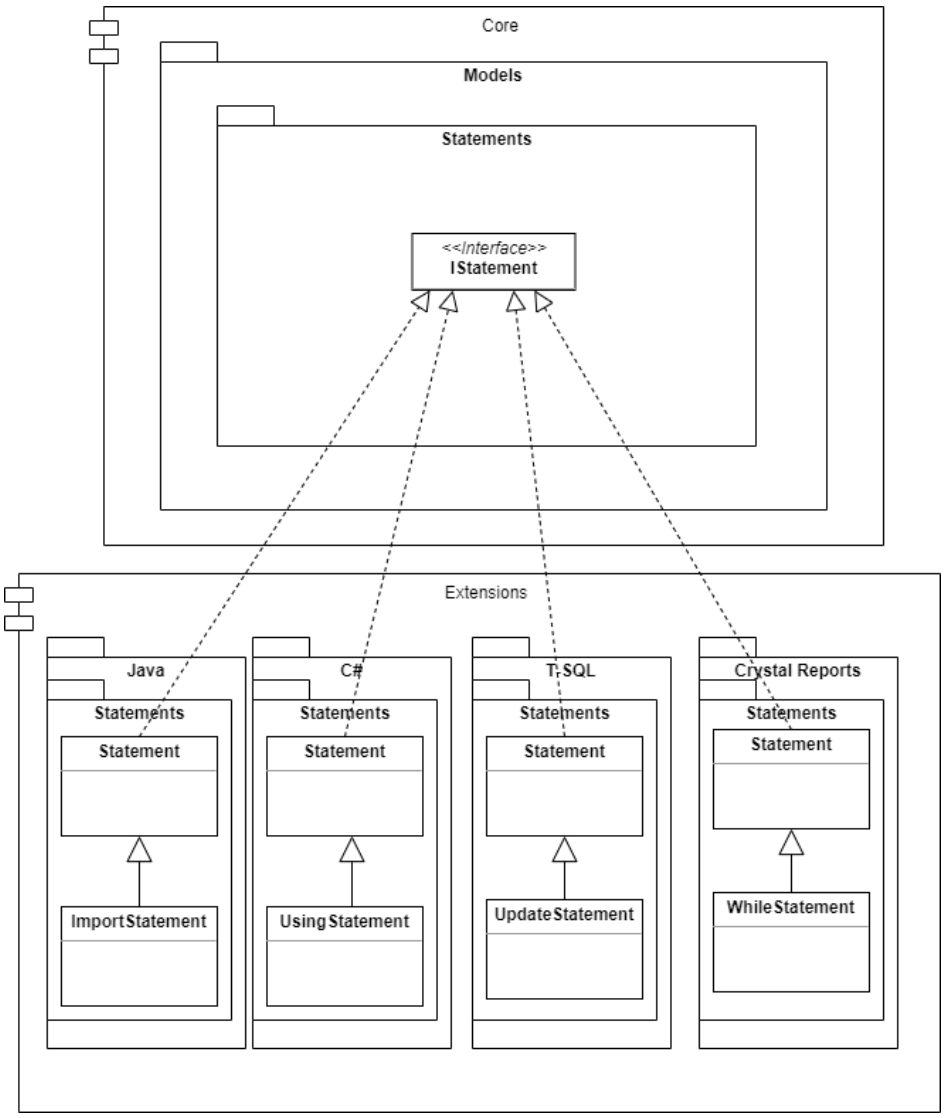

Figure 3.2: Project structure after Soltanpour's version.

3.2.2 Keyword detection approach

In this approach, we define a general interface named Detector; this interface provides the two methods intersectTokens(..) and parse(..) as seen in Listing 3.5.

```
1 int intersectTokens(string text, HashSet<string> tokens);
2 void parse(string text);
```

Listing 3.5: Methods of Detector

We create a class for each supported language in our system that implements an interface. The class contains a set of language-specific keywords that can distinguish that language from others. When the

Algorithm 1: Brute force approach.

Input : A text to detect for which to determine the programming language
Output: The programming language name or null

```
detectors ← [CSharp, Java, TSql, CrystalReports];
foreach detector ∈ detectors do
    try:
        detector.parse(text);
        return detector;
    catch ParseException:
        continue;
return Null;
```

intersectTokens(..) method is called, the tokens set is intersected with the keywords set, and the number of matching keywords found in the input is returned; the most promising candidate is tried, as explained in the next subsection. The parse(..) method runs the given text through the language-specific parser. If the parsing fails, an exception is thrown; otherwise, the code is considered successfully parsed.

3.2.3 Algorithm

Algorithm 2: Keyword detection approach.

Input : A text for which to detect the programming language
Output : The programming language name or null

```
detectorToKeywordsCount ← emptyMap()
detectors ← [CSharp, Java, TSql, CrystalReports]
tokens ← tokenize(text)
foreach detector ∈ detectors do
    detectorToKeywordsCount[detector] ← detector.intersectTokens(tokens)
sortedDetectors ← sortDescendingByCount(detectorToKeywordsCount)
foreach detector ∈ sortedDetectors do
    try do
        detector.parse(text)
        return detector
    catch ParseException do
        continue
return null
```

Above, we present Algorithm 2 for language detection. We start by counting the number of keywords found for each detector in the detectorToKeywordsCount map. Next, we sort the map in descending order based on each detector's value. This allows us to determine the most likely language of the input text. We can efficiently identify the correct parser for the given language by performing this intersection and sorting step. It is important to note that this intersection and sorting step is relatively quick compared to running

the parser against the language.

The purpose of the tokenize(..) method is to split a given text into words by white space, trim them of any additional pre-pending or post-pending whitespace, and return them as a set. This method is language-independent and is intended for use with all supported languages; thus, its performance is high-speed.

3.2.4 Refining the algorithm

Two steps were taken to improve the performance of our algorithm. First, we observed that some words commonly used in comments, such as is, as, and **do** in C#, are also keywords in programming languages. Since we aim to keep the tokenization step fast and minimal and do not use grammars and regular expressions to parse the text, we cannot differentiate comments from other code parts. Therefore, we removed very common keywords from the list of C# keywords to reduce the number of Java code snippets wrongly detected as C# due to their comments. As a second step to improve our algorithm, we have established a threshold for the number of keywords present in the text to filter out weak guesses. Detectors with a keyword count below the threshold are removed before the sorting stage. While this helps quickly identify non-code snippets, there is a risk of missing actual code snippets that use only a few keywords. We have used an empirical approach to determine the best value for this threshold, and we figured out that in our test cases, the best value would be zero to reduce the risk of eliminating the code snippets with a few keywords. Algorithm 3 shows the threshold feature added to the existing keyword detection algorithm. At line 6, we remove those parsers that have fewer keywords in common with the input threshold value. In our empirical study, we used a threshold value of 5. We also could use some approaches such as using regular expressions to discard comments, but we chose not to do this to keep our performance high.

3.2.5 Comparison against brute force

As previously discussed, our approach involves prioritizing the guesses made by the brute force method to achieve better performance without sacrificing accuracy. This approach has the advantage of requiring only a relatively small implementation cost for each supported language. We will provide a detailed comparison of this approach with both the brute force method, Guesslang (Somda, 2021), and the Unix *file* command (Darwin, 2011).

3.3 Summary

In this chapter, we delved into the details of how we extended the functionality of ModCP to support multiple programming languages. We began by discussing the implementation of models, builders, and parser classes

Algorithm 3: Keyword detection approach with threshold.

Input : A text for which to detect the programming language and a threshold value
Output : The programming language name or null

```
1  detectorToKeywordsCount ← emptyMap()
2  detectors ← [CSharp, Java, TSql, CrystalReports]
3  tokens ← tokenize(text)
4  foreach detector ∈ detectors do
5      detectorToKeywordsCount[detector] ← detector.intersectTokens(tokens)
6  filteredDetectors ← detectorToKewordsCount.filter(count >= threshold)
7  sortedDetectors ← sortDescendingByCount(filteredDetectors)
8  foreach detector ∈ sortedDetectors do
9      try do
10         detector.parse(text)
11         return detector
12     catch ParseException do
13         continue
14 return null
```

for the C# language, one of the most widely used programming languages in the industry. We described the most important classes in the Models directory and explained the tests that we performed to ensure the correctness of our implementation.

After successfully implementing the C# language support in ModCP, we extended its functionality to support T-SQL language as well. We followed the same steps as before and implemented the necessary models, builders, parsers, and tests to support T-SQL. By doing so, we were able to expand the scope of ModCP to include support for database programming languages.

Furthermore, we implemented the work done by Singh (2021) to add persistence to ModCP. We integrated their work into our newly implemented models and classes to allow ModCP to persist data.

We discussed the implementation of Crystal Reports support in ModCP. We explained how this was the most challenging language to implement among all of the supported languages, due to the lack of rich online resources and references, combined with the ambiguity of the language. However, we were able to overcome these challenges and successfully add Crystal Reports support to ModCP.

The rest of this chapter details our approach to identifying the programming language of a given string. Initially, we implemented a brute force method to identify bottlenecks and explore potential solutions. We discovered that prioritizing candidates based on a pre-determined order could enhance performance while preserving accuracy. To this end, we developed a keyword detection approach that involved counting the number of language-specific keywords present in the given text, sorting the languages in descending order based on the number of identified keywords, and then executing the parser in this order. Our accuracy was

further improved by excluding common keywords used in comments and setting a threshold for the number of keywords required for identification.

Chapter 4

Evaluation

In this chapter, we evaluate our extension of ModCP and our language identifier tool. Regarding the ModCP extension, we ask one research question (RQ1) and find its answer by extracting data from code metrics in the different developed versions. The research question is:

- RQ1: Is our approach able to simplify the codebase and decrease its complexity when compared to Afzal's version?

Regarding embedded language detection, we have asked three research questions (RQ2–4) to compare our approach with the brute force approach, Guesslang, and the Unix *file* command, in terms of performance, accuracy, and scalability. These research questions are:

- RQ2: How accurate is our approach compared to the brute force approach, Guesslang, and the Unix *file* command?
- RQ3: How does the performance of our approach compare to the brute force approach, Guesslang, and the Unix *file* command?
- RQ4: How scalable is our approach in terms of performance for supporting new languages?

4.1 ModCP efficient extension

In this section, we compare our approach for extending ModCP with the approach taken by Afzal (2020). Our approach involved extending ModCP to support C#, T-SQL, and Crystal Report languages. To evaluate our approach, we compare various code measurements arising from the codebases for two approaches in order to assess the resulting code complexity and maintainability.

Table 4.1 includes general metrics obtained from Men's version, Afzal's version, Walker's version, and Soltanpour's version for the *Core* project. It is evident that in Afzal's version, there are ten times more classes than in Soltanpour's version with double the number of methods per class, which supports the hypothesis that the addition of new languages requires the inclusion of new methods and classes in the existing project for Afzal's approach.

Metric	Men's version	Walker's version	Afzal's version	Soltanpour's version
Number of classes	23	100	1444	102
Number of interfaces	4	38	28	37
Mean methods per class	5.052	4.113	8.249	4.066
Mean lines per class	83.39	7.30	195.0	129.1
Mean executable lines per class	27.26	125.8	53.07	34.47
Mean lines per method	6.218	15.671	7.827	15.29
Mean executable lines per method	2.479	5.038	3.258	4.879

Table 4.1: General metrics for the *Core* project in each version.

4.1.1 RQ1: Is our approach able to simplify the codebase and decrease its complexity when compared to Afzal's version?

To answer this, we measure some code quality metrics defined below for our work result and that of Afzal (2020); these include:

- Cyclomatic complexity: The cyclomatic complexity (CC) metric proposed by McCabe (1976) measures the number of linearly independent paths through a piece of code. The basic idea behind cyclomatic complexity is that the more decision points there are in a program, the more complex it is likely to be. Decision points can include things like **if**-statements, loops, and **switch**-statements. A higher cyclomatic complexity score indicates that a program has more decision points, and is therefore likely to be more complex and harder to maintain (Ebert et al., 2016).

- Maintainability index: The MI score is calculated based on a formula that takes into account several factors, including the Halstead volume (Hariprasad et al., 2017), McCabe's cyclomatic complexity (McCabe, 1976), and the lines of code in the system. These factors are combined to produce a single score between 0 and 100, where higher scores indicate better maintainability (Welker, 2001).

- Depth of inheritance: In the context of object-oriented programming, the depth of inheritance metric refers to the maximum distance from a class node to the root of the tree, which is measured by counting

the number of ancestor classes. A high depth of inheritance can indicate a complex and tightly coupled class hierarchy that may be difficult to modify and maintain over time. On the other hand, a shallow inheritance tree can indicate a more modular and flexible design (Chidamber and Kemerer, 1994; Shaheen and du Bousquet, 2008).

- Class coupling: This is a metric that evaluates the level of interconnection between classes in a software system. This metric measures the number of classes that are coupled to a particular class, which indicates how many other classes rely on that class. In the field of software engineering, it is commonly agreed that excessive coupling can be detrimental to the structure of a system, leading to increased complexity (Hitz and Montazeri, 1995; Briand et al., 1997; Harrison et al., 1998; Aggarwal et al., 2006).

- Lines of code: The lines of source code (LOC) metric is a simple measure of the size of a software program that counts the total number of physical lines of code in the program. This metric is commonly used to provide a rough estimate of the development, testing, and maintenance effort required for the software. However, it is important to note that LOC is not a reliable indicator of software quality, efficiency, or maintainability, as it disregards the structure, readability, and complexity of the code. Additionally, factors such as formatting, comments, and blank lines can significantly affect the LOC count, although they do not contribute to the program's functionality.

- Lines of executable code: This is a measure of software program size that counts only the number of lines of code that are possibly executed at runtime. Unlike *lines of code*, which may be influenced by factors such as comments, formatting, and blank lines, lines of executable code only take into account the code that is executed (Bhatt et al., 2012).

We collect measurements of the metrics for both versions, presenting them in Tables 4.2 and 4.3. Table 4.2 compares the measurements of the classes in the *Core* project between Soltanpour's version and Afzal's version, while Table 4.3 compares the language-specific models, which in Soltanpour's version are located in the *Extensions* project, but were located inside the Models directory in the *Core* project of Afzal's version.

Table 4.2 shows that there is a slight increase in the maintainability index and reduction in depth of inheritance, and we were able to significantly reduce the cyclomatic complexity, class coupling, lines of code, and lines of executable code. This reduction in complexity makes the *Core* project easier to maintain and less complex, as evidenced by the 45.64% reduction in cyclomatic complexity and 42.70% reduction in class coupling. The reduction in lines of code and executable code by 31.8% and 45.80%, respectively, further highlights the significant reduction in complexity within the *Core* project. As the *Core* project is *the* foundational component of ModCP and serves as a dependency for other projects, reducing its complexity

Metric	Afzal's version	Soltanpour's version	Change (A → S)
Maintainability index	84.77	84.86	+0.1%
Cyclomatic complexity	25.13	13.66	-45.6%
Depth of inheritance	1.58	1.30	-17.7%
Class coupling	12.74	7.30	-42.7%
Mean lines of code per source file	141.63	96.60	-31.8%
Mean lines of executable code per source file	45.63	24.73	-45.8%

Table 4.2: Metrics for the *Core* project in each version.

Metric	Afzal's version	Soltanpour's version	Change (A → S)
Maintainability index	80.68	81.06	+0.5%
Cyclomatic complexity	20.11	19.66	-2.2%
Depth of inheritance	2.20	2.11	-4.1%
Class coupling	10.22	12.94	+26.6%
Mean lines of code per source file	91.47	92.01	+0.6%
Mean lines of executable code per source file	31.69	33.66	+6.2%

Table 4.3: Metrics for the Model directory in each version.

is crucial in simplifying the entire system and moving complexity to non-fundamental components.

Upon reviewing Table 4.3, we can observe that the language-specific models in both versions are quite similar, with only minor increases for metrics such as class coupling and lines of code in Soltanpour's version than Afzal's version. Notably, the cyclomatic complexity has been slightly reduced. It is important to note that the language-specific models in the *Extensions* project were extracted from the *Core* project and may have a higher level of complexity, but their overall code complexity is equivalent to the models in the *Core* project of Afzal's version. This indicates that we were able to reduce the functionality of the *Core* project by extracting a portion of it and incorporating it into the *Extensions* project with the same level of code complexity.

Evaluate the correctness of Crystal Reports implementation in ModCP

In order to ensure an accurate representation of Crystal Reports in ModCP and to verify its ability to correctly identify change propagations, a series of test cases were designed. The test cases were based on and inspired by the real-world examples shared by our industrial partner. In each test case, we have two versions of code, the first version is called a baseline version, and the other version is called an updated version. The difference is that the updated version has been refactored in some lines compared to the baseline version to do the same functionality with a different approach. The refactoring may involve adding or removing a variable, changing a variable name, changing some code that has no side effect on other lines, changing

string literals, changing functions that are circularly connected to each other, and changing **for** and **while** loops. The aim is to evaluate ModCP's ability to predict necessary changes, identify unnecessary changes, and detect missed changes. ModCP states changes in terms of lines of code; for example, it states that if you make a change at line number 10, you should probably change lines 11, 14, and 15 as well. The recall and precision results for each test case are presented in Table 4.4. We have designed 7 test cases to cover different aspects of changes.

1. Variable Name: The point of the test is to check if a variable name is changed, then all of the direct and indirect references to that variable should be potentially updated.

2. Function Call: In this test case, a function is calling another function. The callee is returning a string literal; this string literal is changed and we expect that the line in the caller function which is using the return value be marked as a potential change.

3. Function Call V2: This test case involves the same change as the previous one. The difference is that in this case, we have added some lines of code to the called function that do not contribute to the return value, but ModCP still detects them as potential candidates for being updated which is not expected.

4. Chain: In this test case, we want to check if ModCP detecting a change in function D, whose return value is being used by a function C, whose return value is used by a function B whose return value is used by a function A. All lines in functions A, B, and C that are directly or indirectly using the return value of function D are expected to be marked as potential changes.

5. Circular: This test case defines three functions that call each other circularly. We expect that if we make a change in one of the functions, both other functions should be marked as potential changes regarding this update.

6. For: This test case changes the value of a variable name that is defined outside of a **for**-loop but used and re-assigned inside the **for**-loop body. We expect ModCP to identify all potential changes inside the loop body that might be affected by this update.

7. While: This test case makes a change in the value assigned to a variable inside a **while**-loop body. We expect ModCP to identify all potential changes inside the loop body that might be affected by this update.

Recall and precision are two important metrics that are used to evaluate the performance of algorithms. Recall is the proportion of actual positive cases that are correctly identified as positive by the algorithm. It

is a desirable feature when the goal is to minimize the number of false negatives or instances where relevant items are not identified. Precision, on the other hand, refers to the proportion of predicted positive cases that are actually positive. These metrics are widely used in machine learning, data mining, and information retrieval. In the context of computational linguistics and machine translation, recall has been shown to be of major importance as it reflects how many of the relevant cases have been identified. Therefore, both recall and precision are important measures to consider in different contexts. The formulae for calculating recall and precision (relative to a specific context) are shown in Equations 4.1 and 4.2, respectively (Powers, 2020), where TP is the set of true positive occurrences, FN the set of false negatives, and FP the set of false positives. In this context, true positives are the number of potential lines of code to be changed that are detected by ModCP and are actually changed in the updated version. False Positive means the number of potential lines of code to be changed that are detected by ModCP but not changed in the updated version. True negative means that a line of code is not considered to be a potential change and it is not changed in the updated version. False negative means a line of code is considered not to be a potential change, but it had to be changed in the updated version of the code.

$$\text{recall} = \frac{|TP|}{|TP| + |FN|} \tag{4.1}$$

$$\text{precision} = \frac{|TP|}{|TP| + |FP|} \tag{4.2}$$

This evaluation process was conducted to ensure the reliability and effectiveness of our approach for representing Crystal Reports in ModCP.

As depicted in Table 4.4, ModCP always attempts to predict all possible changes, resulting in a recall rate of 100% for Crystal Reports. This means that there are no false negatives in our results. However, it may predict more changes than necessary, ensuring that no changes are missed. This cautious approach implemented by ModCP has been effective for Crystal Reports, as demonstrated by the evaluation results.

Test Case Name	Recall	Precision
Variable Name	100%	50.0%
Function Call	100%	75.0%
Function Call V2	100%	15.3%
Chain	100%	60.0%
Circular	100%	71.4%
For	100%	100.0%
While	100%	70.0%

Table 4.4: Recall and precision of ModCP change propagation identification in Crystal Reports.

4.2 Embedded Language Identification

In this section, we present a comparison between our approach and other methods, namely the brute force method, Guesslang (Somda, 2021), and the Unix *file* command (Darwin, 2011).

4.2.1 Evaluation Procedure

To evaluate our approach against other methods, we need to gather a collection of code snippets written in different languages supported by both Guesslang and our keyword detection approach. Java and C# are the only languages supported by Guesslang, brute-force, and keyword detection while *file* does not support C# but supports Java. All of these approaches take a string as the input parameter and determine its programming language. Therefore, we want to gather samples of files written in the supported languages and use the content of those files as the input for our language detection tools.

Therefore, we need to first gather a collection of code snippets written in Java, and C#. After collecting suitable samples (we refer to these samples as test cases), the next step is to use the content of our test case files as the input of different tools. While running the tools against the given test cases, we will record some data related to their performance and accuracy that makes us enable to compare the approaches in terms of performance, accuracy, and scalability.

4.2.2 Collecting Test Cases

For this purpose, we select 10 popular repositories (based on GitHub stars) per language to use their files' contents as sample data for evaluation; the Java repositories are RxJava (Christensen, 2022), ElasticSearch (Elastic.Co, 2022), Retrofit (Square, 2022b), OkHttp (Square, 2022a), Spring Boot (Spring, 2022a), Guava (Google, 2022), MPAndroidChart (Jahoda, 2021), Glide (BumpTech, 2022), Spring Framework (Spring, 2022b), and ButterKnife (Wharton, 2020), and the C# repositories are Shadowsocks Windows (Shadowsocks, 2021), Powershell (Powershell, 2022), CodeHub App (CodeHubApp, 2020), ASP Net Core (DotNet, 2022), Wox Launcher (Launcher, 2022), dnSpy (dnSpy, 2022), v2rayN (2dust, 2022), eShopOnContainers (.NET Foundation, 2022), WaveFunctionCollapse (Gumin, 2022), and ShareX (ShareX, 2022).

After cloning the aforementioned repositories, in the course of our testing, we discovered that the grammar we used for C# could only handle features up to version 6, which was not sufficient for the repositories under consideration. To address this limitation, we included the ModCP files as a project written in C# 6 to ensure that our test cases use the supported version of C#. Additionally, we made some modifications to our grammar to accommodate the basic features found in the more recent versions such as the ability to parse arrow functions in constructors introduced in C# version 7.

To conclude, we have three sources in our test cases: Java codes extracted from the aforementioned GitHub repositories, C# codes extracted from the aforementioned GitHub repositories, and C# version 6 codes extracted from the ModCP codebase. In the table, we refer to them as *C# Repositories*, *Java Repositories*, and *C# ModCP* respectively.

For testing the detectors in Java and C#, we gathered files with *.java* and *.cs* extensions from the sources. We then developed a Python script and randomly selected 100 files from each source as our test cases. To select random cases, we used Python 3's built-in random library which generates pseudo-random numbers (Python Software Foundation, 2008). Additionally, we selected 50 files per source that have the largest number of characters to evaluate the detectors' performance on longer inputs. We use the content of these files as the code snippet inputs to *file*, Guesslang, brute force, and keyword detection approaches.

4.2.3 Order of parsers in brute force

Each time that we have run the brute force algorithm, we used a random ordering of different language parsers. To make our results more reliable and prone to best-case or worst-case scenarios, for each test case, we have run it for 10 times, and each time the order was random, and we picked the median number.

4.2.4 RQ2: How accurate is our approach compared to the brute force, Guesslang, and *file* approaches?

To answer this question, we run our test cases against each method. The results of our first attempt are included in Table 4.5. Regarding Equation 4.1, TP in this context contains the test cases that their language is guessed correctly, and FN contains test cases that their language was not guessed correctly. In this context, precision is not relevant since we do not have any false positives; therefore, we are just measuring recall in our calculations.

As the Unix *file* command does not support C# files, its recall in all of the tables in this research question is represented with 0%, but it supports Java files, and its recall for Java is compared to other approaches.

Test case	Keyword Detection (With Validation)	Guesslang	Brute Force	*file*
C# Repositories	32%	90%	32%	0%
Java Repositories	100%	92%	100%	87%

Table 4.5: Recall of each strategy (initial attempt).

The errors encountered in the brute force method are due to some code snippets that are valid but not supported by our grammar. This highlights a limitation of the brute force approach: it requires a fully functional grammar for each version of each programming language, making it challenging to maintain and

scale; however, in our context, we ultimately need to obtain abstract syntax trees for the code snippets, which would require such grammars anyways.

The errors that happen in the keyword detection approach have the same cause. On our next try, we tried to fix errors in our grammar to support more features of higher versions of C# such as supporting the syntax of anonymous functions, and we removed the test cases that are not compatible with the brute force grammar. We believe that it is fair to remove some of the test cases since we want to test our system against what it actually supports, and brute force must have the ability to parse all of the given inputs. Table 4.6 shows the results after this improvement.

Test case	Keyword Detection (With Validation)	Guesslang	Brute Force	*file*
C# Repositories	100%	91%	100%	0%

Table 4.6: Recall after C# grammar improvement.

We significantly increased the accuracy of our C# detector from 32% to 75%, but it still falls short of the ideal accuracy achieved in Java. To address the issue of grammar incompatibility, we added more samples of C# version 6 from the ModCP project. Our initial attempt on these C# version 6 files resulted in improved accuracy, as shown in Table 4.7. These results indicate that our approach works well when the grammar is properly aligned with the input provided. After this, we decided to keep these samples as part of our next measurements that aim to assess other aspects of accuracy and performance.

Test case	Keyword Detection (With Validation)	Guesslang	Brute Force	*file*
C# ModCP	100%	96%	100%	0%

Table 4.7: Recall on ModCP C# files.

As outlined in Chapter 3 and the keyword detection algorithm, our algorithm consists of two parts: the first part involves predicting the most likely programming language based on the input string, while the second part involves validating the prediction by running the parsers against the input. In contrast, Guesslang and *file* solely focus on the prediction aspect and do not perform any validation. As a result, we deemed it appropriate to compare the accuracy of our prediction component with that of Guesslang and *file* to gauge the effectiveness of our algorithm's prediction capabilities without validation. Table 4.8 presents the preliminary results of this comparison.

We encountered a peculiar scenario wherein the accuracy of C# identification was remarkably high, while that of Java was surprisingly low. Upon further investigation, we determined that C# encompasses almost all of the keywords utilized in Java and has some keywords, such as *is*, *as*, and *in*, that are commonly present in comments. Consequently, our keyword detection approach perceives these keywords in comments

Test case	Keyword Detection(Without Validation)	Guesslang	*file*
C# ModCP	100%	96%	0%
C# Repositories	99%	91%	0%
Java Repositories	51%	92%	87%

Table 4.8: Recall of guesses without validation.

as language-specific keywords, leading to a higher rate of C# identification and a lower rate of Java identification. As a result, the number of identified C# codes has increased, while the number of identified Java codes has decreased.

To address this problem, we opted to exclude the generic keywords from the list of C# keywords, i.e., treat them as stop words (Fox, 1989), and re-run our tests. The outcomes of these tests are presented in Table 4.9.

Test case	Keyword Detection (Without stop words and validation)	Guesslang	*file*
C# ModCP	96%	96%	0%
C# Repositories	79%	91%	0%
Java Repositories	96%	92%	87%

Table 4.9: Recall of guesses without validation, but after stop word removal.

As evident from the results, this modification resulted in a substantial improvement in the accuracy of identifying Java codes. It has decreased the accuracy in C# but the tradeoff is more even now between different languages.

Conclusion

We can conclude our final accuracy numbers in Table 4.10. In the KD (With validation) row, each guess is validated by running against the guessed language grammar.

Test case	KD (Without validation)	KD (With validation)	Guesslang	Brute Force	*file*
C# ModCP	96%	100%	96%	100%	0%
C# Repositories	79%	100%	91%	100%	0%
Java Repositories	96%	100%	92%	100%	87%

Table 4.10: Final recall of guesses (Stop words are ignored in keyword detection columns).

Based on our analysis, we can deduce if our approach outperforms the other approaches in terms of performance, it would become a viable alternative with the same accuracy but better performance, making it a favourable option.

4.2.5 RQ3: How does the performance of our approach compare to the brute force, Guesslang, and *file* approaches?

To measure the performance, we used the same samples as the previous analysis. In addition, for each project, we selected the top 50 files with the most characters to see how our approach performs in the worst-case scenario. In order to mitigate the impact of job scheduling and prioritization by the operating system, we conducted a minimum of 10 test runs for each case and selected the median result to minimize undesired variability.

Test case	Keyword Detection (With validation)	Guesslang	Brute Force	*file*
C# ModCP	0.039	2.965	0.082	N/A
C# Repositories	0.047	2.621	0.085	N/A
Java Repositories	2.287	2.872	2.333	0.003

Table 4.11: Mean time to detect a string's language (in seconds).

Table 4.11 provides us with the mean time required to detect a string's language, for each of the three approaches. Notably, we can observe that Guesslang takes a considerable amount of time to detect the programming language of a single string and is more suited for manual queries rather than scalable automated ones. Furthermore, we can see that the keyword detection approach has reduced the time by 50% compared to the brute force method.

These results unexpectedly reveal that the processing time for Java repositories in both the keyword-detection and brute-force approaches is significantly longer than that of other programming languages. After further investigation, we determined that the root cause of this problem is the grammar we used for Java, which is not optimized for efficiency. To mitigate this issue, we replaced the grammar with an open-source grammar obtained from GitHub (2021), resulting in a significant improvement in our algorithm's performance shown in Table 4.12.

Test case	Keyword Detection (With validation)	Guesslang	Brute Force	*file*
Java Repositories	0.008	2.872	0.009	0.003

Table 4.12: Performance on Java (in seconds) after updating the grammar.

The subsequent evaluation involves measuring the performance of the keyword detection approach against other methods for files with the maximum number of characters to assess their performance under worst-case conditions. To achieve this, we selected the top 50 files with the most characters from both Java Repositories and ModCP. The outcomes of running all three approaches on these files are demonstrated in Table 4.13.

As evident from the results in Table 4.13, our keyword detection approach shows significant improvement

Test case	Keyword Detection (With validation)	Guesslang	Brute Force	*file*
C# ModCP - Top 50 Chars	0.555	2.965	0.645	N/A
Java Repositories - Top 50 Chars	0.298	3.081	0.301	0.005

Table 4.13: Performance on top 50 files with the most characters, in seconds.

in performance compared to other approaches. Specifically, it outperforms Guesslang by almost 82% and the brute-force approach by almost 15% for C# language. In the case of Java, our method performs almost 90% better than Guesslang. In this case, keyword detection and brute force have nearly identical performances. In this table, the *file* command is showing the best performance, but we should note that in this table, keyword detection and brute force are validating the guess but Guesslang and *file* are just making a guess without validation. We will conduct another experiment to just compare the guessing in the next part.

To evaluate the impact of the validation part on the performance of the keyword detection approach, we have conducted a similar experiment as in the accuracy analysis by removing the validation step. The objective is to compare the performance of the guessing section of our approach to Guesslang. The results are shown in Table 4.14 for both the general sets and the sets restricted to the top 50 samples with the most characters.

Test case	Keyword Detection (Without validation)	Guesslang	*file*
Java Repositories	4.865×10^{-6}	2.872	3.04×10^{-3}
C# Repositories	3.190×10^{-6}	2.621	N/A
C# ModCP	8.401×10^{-6}	2.965	N/A
Java Repositories - Top 50	1.147×10^{-4}	3.081	5.18×10^{-3}
C# ModCP - Top 50	9.487×10^{-5}	3.058	N/A

Table 4.14: Performance on Java after updating the grammar, in seconds, excluding validation.

The results presented in Table 4.14 demonstrate that the keyword detection approach significantly outperforms Guesslang in terms of performance. Specifically, keyword detection is approximately 32,000 times faster than Guesslang. In the case of C# repositories, the keyword detection approach is performing nearly 600 times better than *file*, and in the case of C# ModCP top 50 files, it performs better than *file* by 45 times. This vast difference can be attributed to the simplicity of keyword detection, which only involves a tokenizer and a few set-intersection operations, as opposed to the complex, deep learning algorithms used by Guesslang and pattern-matching used by *file*.

Conclusion

To address the research question, we performed measurements on regular and worst-case samples using the keyword-detection approach, which prioritizes language detection to improve brute-force performance. We

found that the keyword-detection approach significantly outperformed Guesslang in just the guessing section and performed at least 45 times better than the *file* command again in the guessing section. Considering its superior performance and comparable accuracy, we conclude that the keyword-detection approach offers the best option for the accuracy–performance trade-off.

4.2.6 RQ4: How scalable is our approach in terms of performance for supporting new languages?

To investigate the answer to this research question, we collected test cases for Crystal Reports and T-SQL, in addition to the existing ones for Java and C#. Find it EZ, our industrial partner, provided a rich resource for these languages, which facilitated the sample-gathering process, as there were not many open-source samples available. For our research, we conducted a study to examine the impact of integrating a new language into our keyword detection approach, with the goal of determining the additional cost of incorporating a new language into the overall system's performance. This analysis will provide insight into the performance of our approach as we aim to expand it to support a larger number of languages.

In our current study, we have implemented a keyword detection approach that supports four languages. To evaluate the impact of adding a new language to the system's performance, we conduct a series of experiments. Each experiment involves generating a new permutation of the four languages and testing the system's performance by adding languages in the order specified by the permutation. We begin with the first language in the permutation and test it against all samples for all four languages. We then add the next language in the permutation and repeat the tests. This process continues until all four languages have been added, and we move on to the next permutation to repeat the experiment. Figure 4.1 shows an example in which the permutation is Crystal Reports, T-SQL, Java, and C#.

Since there are $4! = 24$ permutations of these languages, we have 24 versions of diagrams similar to Figure 4.1. To determine the impact of adding a new language on our approach latency, we performed a linear regression. To do this, we used Python to write a script using the *scikit-learn* library (Buitinck et al., 2013; Pedregosa et al., 2011) that provided us with an API to calculate the linear regression for the given data. After calculation, we drew the diagram using the *matplotlib* library (Hunter, 2007) in Python.

The equation of the fitted line obtained from the data is $y = 2883x + 18323$, indicating that adding a new language only results in an increase of 2,883 nanoseconds, while the constant 18,323 nanoseconds can mainly be attributed to the cost of tokenization.

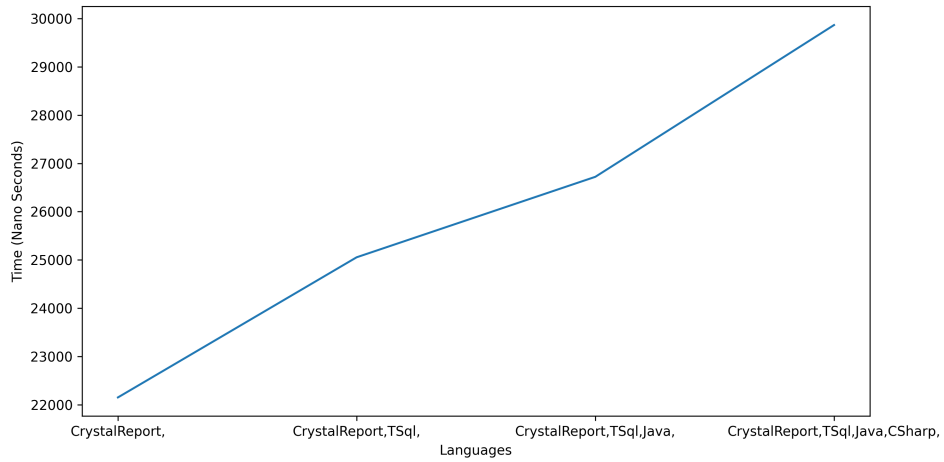

Figure 4.1: An example of adding new languages.

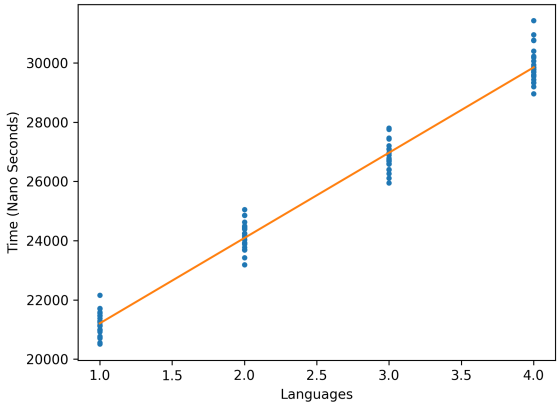

Figure 4.2: Regression over all 24 sets of data.

Conclusion

Drawing upon the measurements in this section, we infer that the cost of integrating a new language into our keyword detection approach is negligible and that the approach scales easily to accommodate additional languages. Essentially, the integration of a new language merely involves an additional *intersect* operation between two sets, which is highly efficient and justifies the approach's scalability.

4.3 Summary

We conducted an evaluation of our approach for extending ModCP in comparison to Afzal's version. The main objective of our evaluation was to investigate how our approach could enhance code complexity. We have figured out that our development approach will increase the maintainability index by 0.1% and reduces

complexities such as cyclomatic complexity by 45% and depth of inheritance by 17% and the number of lines of code by 30–40% in the *Core* project. We then assessed our approach for detecting embedded code snippets using keyword detection, comparing it to the brute-force method, the Guesslang tool, and the Unix *file* command in terms of accuracy and performance. We evaluated both the guess and evaluate components, as well as just the guessing component. Our results indicate that the guessing component of our approach has nearly identical recall compared to the Guesslang and *file* but with better performance. Therefore, our approach is highly efficient and accurate for detecting code snippets. We also examined the scalability of the guessing section and determined that adding a new language has a minimal impact (2.883 nanoseconds added for a new language) on the program's performance.

Chapter 5

Discussion

In this chapter we discuss the limitations of our work and potential threats to its validity, and we propose ideas for future work.

5.1 Limitations

5.1.1 ModCP extension

One potential limitation of the model proposed in the *Extensions* project is that any language incorporated must be able to conform to the predefined classes in the *Core* project, such as Statement and Expression. This limitation could pose a challenge when attempting to implement a language that cannot be adequately represented by the *Core* project's concepts, as is the case with the Prolog programming language. Prolog relies on formal logic and symbolic computation, utilizing logical rules and facts rather than explicit statements or expressions to define program behaviour (Colmerauer, 1990). While T-SQL and Crystal Reports are syntactically different languages than Java and C#, the concepts used in those languages were ultimately able to be mapped to the model. However, if the concepts of a new language are significantly different from those defined in the *Core* project, the model may not be able to represent it effectively.

In the *Extensions* project, the code for each language is isolated from others, which creates the challenge of not being able to reuse code across languages. This results in duplication of code, particularly for similar use cases in languages such as C# and Java. While duplicate code may not require frequent modification (Hotta et al., 2012), it can complicate the maintenance and evolution of the code (Ducasse et al., 1999). Although we have pursued the approach with the idea of complete independence of the languages supported, there is also the possibility of grouping related languages so as to provide a type hierarchy for some of the

common constructs therein; whether this will be an improvement or would lead to a more confusing structure overall is an open question.

5.1.2 Language Identification

Our keyword-detection approach has limitations that we discuss here.

The first problem is that, unlike Guesslang which can be trained on many languages and then detect all of them, we need development effort to add each new language. That is why our approach supports four languages at this stage while Guesslang supports 54 programming languages. If we want to add a new language with a validation part as well, we should implement a correct ANTLR grammar for that language and this can increase development costs. If we just want the guessing part, we just need to gather keywords of that language and create a new Detector class for that language.

Another issue is that Guesslang can be trained on multiple versions of a programming language, and slight syntactical differences among these versions do not affect its precision much. However, in our validation process, we must ensure that our grammar supports all versions of the language or utilize different grammars for each version. Alternatively, we can create a grammar that encompasses the fundamental principles of the language, which are consistent across all versions. Nonetheless, this method demands significantly more maintenance than Guesslang.

Another limitation is that in the guessing phase, we do not parse the provided text, which causes the keyword detection approach to consider keywords used in the comments. If a foreign language keyword is often repeated in the comments, it can mislead the keyword detection approach. Similarly, this scenario can occur with Guesslang as well, where if we embed a code in another language in the comments section of the input text, Guesslang's output may be inaccurate.

An additional constraint of our approach is its inability to identify strings that are produced during runtime. Since we examine the codebase in a static manner and do not monitor the code during execution, any code snippets created by concatenating various variables, for instance, would not be detected. This limitation is shared by both the brute force method and the Guesslang approach as well.

Since the brute force and the keyword detection approach with validation parse the input with the grammars, the input string should be in its final format to be able to be parsed by the grammar. But our approach without validation alongside Guesslang and *file*, is still able to make a guess about the programming language.

Another limitation of our approach is supporting languages that do not have any keywords. LISP is an example of a language that does not have any keywords (Steele, 1990). At this moment, our industrial

partner is not using languages of this kind, but in case its needed, we need to implement a new detector class for this language and process the input in a different way such as regular expressions.

5.2 Threats to Validity

There are several threats to the validity of our work for the extension of ModCP and also for identifying the embedded language. Based on Wright et al. (2010),Wohlin et al. (2012), and Siegmund et al. (2015), we have categorized threats to validity into three categories.

- **Internal validity threats**: This pertains to the degree of accuracy and scientific soundness of the conclusions drawn from our observations regarding the relationship between the independent and dependent variables. It concerns the validity of the relationship between these variables.
- **External validity threats**: This pertains to the validity issues that arise when attempting to generalize and replicate experimental results in different contexts or scenarios.
- **Construct validity threats**: A construct validity threat refers to a potential issue or challenge that could undermine the validity of the constructs being measured or manipulated in a research study. These threats arise when there are concerns regarding the accuracy, appropriateness, or representativeness of the operationalizations or measures used to assess the constructs of interest.

5.2.1 Language identification

We collected a dataset of samples to evaluate language identification methods using widely-used GitHub repositories. These repositories are typically subjected to extensive review by developers and adhere to best practices in coding syntax and commenting. However, if we had access to private or industrial code bases, we might encounter varying syntaxes and commenting conventions, which could require further adjustments to our keyword-based detection or grammar to enhance their precision. This can be categorized as an *external validity threat*. To fix this in the future, we should get access to some private industrial repositories and study their code to check if they use some different patterns in their programming style and update our approach accordingly.

In the evaluation chapter, the reported performance metrics for each approach may be influenced by various factors such as background processes running on the operating system during the execution of our jobs, or issues with job scheduling and priority on the OS. This can be categorized as an *internal validity threat*. To fix this, we have taken measures to minimize these effects by running each test case at least

ten times and selecting the median value of the resulting data, ensuring that we do not rely solely on the best-case or worst-case scenarios.

We support four languages in the current version of the keyword detection approach. In case we want to scale our approach and support multiple new languages, common keywords and comments in the given input can increase the chance of being wrongly detected and the accuracy numbers reported in the evaluation chapter may be subject to change. This can be categorized as an *external validity threat*. To fix this, we should try to accommodate the language-specific keywords that are not used in other languages and increase their weight in scoring the detector to decrease the effect of this threat as much as possible.

5.3 Future work

5.3.1 ModCP extension

Now that we have a system inside ModCP that can guess the embedded language, the next step is to use this system inside the parsing section of ModCP. So when ModCP faces a string literal, it passes the string literal to our keyword detector and if the keyword detector says that this is a programming language, it tries to parse it. After parsing, ModCP should determine the variables inside the embedding code that are being used by the embedded code and make relations between them,

ModCP architecture now has the ability to be extended and support new languages without making the code much more complicated. Therefore, we can implement new languages such as Python, JavaScript, MySQL, C, and C++.

5.3.2 Language detection

We can add support for more programming languages in the keyword-detection approach. The process of adding a new language is to gather all of its keywords, write a grammar for our desired version, then create a new class to guess based on the keywords and validate using the grammar.

We can add support for C# language to the Unix *file* command as it is an open-source library. Adding support for new languages makes this tool a more powerful tool and with its performance that is notably better than Guesslang, it can become one of the main options for this purpose.

An additional avenue of exploration in this field is the optimization of Guesslang's functionality. As outlined in the evaluation section, the primary issue facing Guesslang is its sub-optimal performance, rendering it unsuitable for real-time deployment. Should a viable solution be identified to resolve this issue and a usable API be developed for other tools, such as ModCP, the accuracy, and scalability of Guesslang in terms

of language support make it a viable replacement option.

5.4 Summary

This chapter provides an overview of the limitations of the research, including challenges posed by the use of disparate syntaxes, code duplication, the scope of the keyword detection approach, and the parsing of comments. The discussion also highlights potential threats to the validity of the study, particularly the reliance on the dataset gathered from popular open-source repositories that may be different from actual private industrial code bases. Furthermore, the chapter suggests future directions for research, such as integrating the keyword detector into the ModCP parsing section and exploring relationships between embedded and embedding code.

Chapter 6

Conclusion

Software maintenance is the process of modifying and updating systems to improve performance, fix bugs, and satisfy new requirements. Maintenance is a crucial part of a software's life cycle, and over time, it gets more expensive and complex as the code base grows. The manual maintenance of software by software developers can be prone to various types of errors; to mitigate this risk, software development and analysis tools (SDATs) can aid in semi-automating the process, thereby increasing the efficiency and accuracy of the maintenance process. Impact analysis involves identifying potential consequences of a modification, or estimating what parts of code need to be updated in reaction to a change. Change propagation (CP) is a software development technique that ensures the consistency and functionality of a system by updating program outputs or results when there is a modification in its inputs or dependencies.

ModCP is a static change propagation analysis tool. Initially, the tool was specifically designed for the analysis of Java source code. In the next step of the development of ModCP, the industrial partner of the ModCP project required an extension in order to accommodate new languages, focusing on common and popular languages such as C# and Python. This part of the development was done by Afzal (2020). In their implementation architecture, all of the models representing the ASTs were stored inside the Core project and there was a single model representing all languages. However, they also needed to implement a new model for cases that could not fit into the current universal model. For instance, they introduced a directory named *GPP* to represent Python, Java, C#, and T-SQL models, but because T-SQL has some different concepts, they had to make another directory for some special cases of T-SQL. Some classes in this directory extend each other and some extend classes in the *GPP* directory. This caused the code to be more complex and harder to maintain, resulting in issues like the following.

- Each language possesses its own special concepts that may or may not be present in other languages.

When multiple languages use the same model, those concepts must be defined in all of them.

- In a significant number of instances within the ModCP codebase, we observed that a class in a TSql directory extends a class from the GPP directory while T-SQL is not a special case of "GPP", so an inheritance relationship is wrong in this case.
- The current architecture makes the process of adding a new language ambiguous and costly. A developer does not know whether they have to implement a new directory or add the new language to existing classes, making the code base poorly extendable.

In addition, another language of particular significance for our industrial partner was Crystal Reports (SAP, 2023), a business intelligence application that enables the creation and generation of reports from various data sources. We wanted to implement this language besides other languages in the ModCP as well.

During the implementation of the support for Crystal Reports for ModCP, a significant number of instances were encountered where embedded SQL code was utilized in the form of strings and called upon other functions. To establish a relationship between code and embedded strings, it is necessary to determine whether a string in question is a piece of code or simply a regular string. To address this, an approach was designed and developed that takes a string as input and outputs a determination of whether it is written in a programming language or not.

There are multiple studies and development works about representing multiple languages using models. Some studies use a universal model to represent all languages and some use a model per group of languages or a model per language to do so. Professor Robert Walker has used the idea of a combination of these two models to represent languages. The idea is to create an abstract model inside the *Core* project and create a language-specific model for each language in the *Extensions* project. Each language in the *Extensions* project inherits the base abstract model and has nothing in common with other languages in the *Extensions* project. We implemented the C#, T-SQL, and Crystal Reports languages on this architecture and compared the results with the architecture developed by Afzal (2020) to see how these architectures will lead to increase code maintainability and reduce code complexity.

6.1 ModCP development

We developed our ModCP extension work by starting with the development of the support for the C# language. For each language, we have to write or find a grammar (lexer and parser), generate the C# code from the grammar, implement the models in the *Extensions* project, and create a builder to generate models by iterating on the generated lexer and parser. After doing these, we have to write tests to make sure that

each stage from parsing the language to creating the dependency graphs for the given code in that language is correctly executed. After C#, we did the implementation for T-SQL and Crystal Reports languages as well.

During the implementation of the mentioned languages, we have added the work done by Singh (2021) to our models. The responsibility of their work was to persist the models and graphs generated from the source code by using the DataContract API to avoid the cost of re-calculation.

6.2 Language identifier development

During the development of Crystal Reports, we encountered situations where SQL code was embedded within the report file, which made a connection with Crystal Reports variables. Similar occurrences were also found in Java or C# code, and HTML code with embedded JavaScript. Parsing the embedded code and finding its relation to the embedding code is a crucial feature for ModCP as a change propagation framework. In response to this, we developed a system that takes a string as input, detects the programming language of the code snippet, and validates the guess if required.

Our approach aims to improve the brute-force performance of traditional methods that run the given string against all supported grammars to determine the correct parser. Our solution involves designing a Detector class for each language, which contains the keywords specific to that language. By counting the number of keywords found in the input, we prioritize the order of parser execution against the given string. The brute force method can be used as one of our baseline approaches based on this.

There are limited implementations regarding this work available. The closest tools to our work are Guesslang and the Unix *file* command. Guesslang utilizes deep learning and TensorFlow to implement a linear classifier trained on an extensive dataset of publicly available source codes on the internet. The Unix *file* command is a command-line tool that detects the type of the given file and its programming languages if applicable using the patterns it finds in the file. We also used them as baseline approaches for guessing the input without validation.

6.3 Evaluating our implementation

In order to evaluate our implementation for extending ModCP and language identifier, we have designated research questions and found their answers by extracting data from code metrics, and performance and accuracy tests.

RQ1: *Is our approach able to simplify the codebase and decrease its complexity when compared to the*

baseline? We evaluated our codebase against the code developed by Afzal (2020), considering factors such as cyclomatic complexity, maintainability index, depth of inheritance, class coupling, and lines of code. Our approach has increased the maintainability index by 0.1% and reduced complexities such as cyclomatic complexity by 45% and depth of inheritance by 17% and the number of lines of code by 30–40% compared to Afzal's version. As a result of our analysis, we were able to reduce the complexity of our Core project and establish a new project called Extensions. This project now contains all language models, which are isolated from each other to improve the overall organization of our codebase.

RQ2: *How accurate is our approach compared to the brute force and the baseline approaches?* Regarding accuracy, our approach exhibits the same level of accuracy as the brute force method, achieving a 100% accuracy rate when utilizing both the guessing and validation components. On the other hand, Guesslang provides a 94% accuracy rate and *file* provides 87% accuracy on Java. By removing the validation component from our approach, the guessing component was able to achieve an average accuracy rate of approximately 90% with significantly better performance.

RQ3: *How is the performance of our approach compared to the brute force and the baseline approaches?* For the purpose of evaluating performance, we conducted tests on three approaches, both on typical use cases and on samples with the highest number of characters in the file. After making adjustments to different parts of our approach, we were able to achieve a performance that was almost 32,000 times better than that of Guesslang and at least 45 times better than *file* for the guessing component alone. Moreover, our approach demonstrated performance that was 10 times better than Guesslang for both guessing and validation and twice as fast as brute force.

RQ4: *How scalable is our approach in terms of performance for supporting new languages?* For this study, we assessed the scalability of our keyword detection approach in adding new languages. We added languages one by one for all possible permutations of supported languages to measure the time it takes to add a new language. The results showed that adding a new language only takes 2,883 nanoseconds, which is an extremely small amount of time. This indicates that our keyword detection approach can easily accommodate new languages without increasing the system's latency.

6.4 Future work

Regarding future work for the ModCP extension, one potential avenue is integrating an embedded language detection system into ModCP to enable the parsing of multi-language codes and relating their graphs to each other. Additionally, new languages like Python, JavaScript, MySQL, C, and C++ can be implemented into the new architecture of ModCP. The keyword detection approach can be improved by adding support

for more programming languages in the language detection system. Another area of exploration in this field is optimizing Guesslang's functionality. Currently, the sub-optimal performance of Guesslang makes it unsuitable for real-time deployment. Moreover, adding the support for more languages such as C# to *file* makes it a powerful tool with an acceptable performance that can be used in use cases such as guessing the embedded language.

Bibliography

2dust. 2022. v2rayN. https://github.com/2dust/v2rayN. [Accessed 29 June 2023].

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ [Accessed 29 June 2023].

Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2006. Adaptive Functional Programming. *ACM Transactions on Programming Languages and Systems* 28, 6 (Nov. 2006), 990–1034. https://doi.org/10.1145/1186632.1186634

Umut A. Acar, Guy E. Blelloch, and Jorge L. Vittes. 2005. An Experimental Analysis of Change Propagation in Dynamic Trees. In *Proceedings of the Workshop on Algorithm Engineering and Experiments and the Workshop on Analytic Algorithmics and Combinatorics*. 41–55.

Raheela Afzal. 2020. *Modular Dependency Analysis in Heterogeneous Software Systems*. Master's thesis. University of Calgary.

K. K. Aggarwal, Yogesh Singh, Arvinder Kaur, and Ruchika Malhotra. 2006. Empirical Study of Object-Oriented Metrics. *Journal of Object Technology* 5, 8 (2006), 149–173. http://www.jot.fm/issues/issue_2006_11/article5 [Accessed 29 June 2023].

Florian Angerer. 2014. Variability-Aware Change Impact Analysis of Multi-Language Product Lines. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden). 903–906. https://doi.org/10.1145/2642937.2653472

Trosky B. Callo Arias, Paris Avgeriou, and Pierre America. 2008. Analyzing the Actual Execution of a Large Software-Intensive System for Determining Dependencies. In *Proceedings of the Working Conference on Reverse Engineering*. 49–58. https://doi.org/10.1109/WCRE.2008.11

Ken Arnold, James Gosling, and David Holmes. 2005. *The Java Programming Language*. Addison Wesley Professional.

Robert Arnold and Shawn Bohner. 1996. *Software Change Impact Analysis*. IEEE Publications, Piscataway, New Jersey, USA.

Amir Aryani, Fabrizio Perin, Mircea Lungu, Abdun Mahmood, and Oscar Nierstrasz. 2011. Can We Predict Dependencies Using Domain information?. In *Proceedings of the Working Conference on Reverse Engineering*. 55–64. https://doi.org/10.1109/WCRE.2011.17

Sufyan Basri, Nazri Kama, Roslina Ibrahim, and Saifuladli Ismail. 2015. A Change Impact Analysis Tool for Software Development Phase. *International Journal of Software Engineering and Its Applications* 9 (Sept. 2015), 245–256. https://doi.org/10.14257/ijseia.2015.9.9.21

Keith H. Bennett and Václav T. Rajlich. 2000. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering*. 73–87.

Kaushal Bhatt, Vinit Tarey, Pushpraj Patel, Kaushal Bhatt Mits, and Datana Ujjain. 2012. Analysis of source lines of code (SLOC) metric. *International Journal of Emerging Technology and Advanced Engineering* 2, 5 (2012), 150–154.

Lionel Briand, Prem Devanbu, and Walcelio Melo. 1997. An Investigation into Coupling Measures for C++. In *Proceedings of the International Conference on Software Engineering*. 412–421. https://doi.org/10.1145/253228.253367

Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122.

BumpTech. 2022. Glide. https://github.com/bumptech/glide. [Accessed 29 June 2023].

Shyam R. Chidamber and Chris F. Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20, 6 (1994), 476–493. https://doi.org/10.1109/32.295895

Elliot J. Chikofsky and James H. Cross. 1990. Reverse engineering and design recovery: A taxonomy. *IEEE Software* 7, 1 (1990), 13–17. https://doi.org/10.1109/52.43044

Ben Christensen. 2022. RxJava. https://github.com/ReactiveX/RxJava. [Accessed 29 June 2023].

CodeHubApp. 2020. CodeHub. https://github.com/CodeHubApp/CodeHub. [Accessed 29 June 2023].

Alain Colmerauer. 1990. An Introduction to Prolog III. *Commun. ACM* 33, 7 (July 1990), 69–90. https://doi.org/10.1145/79204.79210

Krzysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming.* Addison Wesley, Boston, MA.

Neelam Dahiya, Sheifali Gupta, and Sartajvir Singh. 2022. A Review Paper on Machine Learning Applications, Advantages, and Techniques. *ECS Transactions* 107, 1 (April 2022), 6137–6150. https://doi.org/10.1149/10701.6137ecst

Ian Darwin. 2011. Fine Free File Command. https://www.darwinsys.com/file/. [Accessed 29 June 2023].

Sayed Dehaghani and Nafiseh Hajrahimi. 2013. Which Factors Affect Software Projects Maintenance Cost More? *Acta Informatica Medica* 21 (March 2013), 63–66. https://doi.org/10.5455/AIM.2012.21.63-66

Jamilah Din, Anas AL-Badareen, and Yusmadi Jusoh. 2012. Antipatterns detection approaches in object-oriented design: A literature review. In *Proceedings of the International Conference on Computing and Convergence Technology.* IEEE, 926–931.

dnSpy. 2022. dnSpy. https://github.com/dnSpy/dnSpy. [Accessed 29 June 2023].

DotNet. 2022. ASP Net Core. https://github.com/dotnet/aspnetcore. [Accessed 29 June 2023].

Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. 1999. A language independent approach for detecting duplicated code. In *Proceedings of the IEEE International Conference on Software Maintenance.* 109–118.

Christof Ebert, James Cain, Giuliano Antoniol, Steve Counsell, and Phillip Laplante. 2016. Cyclomatic complexity. *IEEE Software* 33, 6 (2016), 27–29. https://doi.org/10.1109/MS.2016.147

Elastic.Co. 2022. ElasticSearch. https://github.com/elastic/elasticsearch. [Accessed 29 June 2023].

Norman E. Fenton and Martin Neil. 2000. Software Metrics: Roadmap. In *Proceedings of the Conference on The Future of Software Engineering.* ACM, 357–370. https://doi.org/10.1145/336512.336588

File. 2011. File GitHub Repository. https://github.com/file/file. [Accessed 29 June 2023].

Martin Fowler. 2019. *Refactoring: Improving the design of existing code.* Addison-Wesley.

Christopher Fox. 1989. A Stop List for General Text. *SIGIR Forum* 24, 1–2 (Sept. 1989), 19–21. https://doi.org/10.1145/378881.378888

GitHub. 2021. Antlr-grammars-v4. https://github.com/antlr/grammars-v4. [Accessed 29 June 2023].

GitHub. 2022. Linguist. https://github.com/github/linguist. [Accessed 29 June 2023].

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning.* MIT press.

Google. 2022. Guava. https://github.com/google/guava. [Accessed 29 June 2023].

Maxim Gumin. 2022. WaveFunctionCollapse. https://github.com/mxgmn/WaveFunctionCollapse. [Accessed 29 June 2023].

Kevin Gurney. 1997. *An Introduction to Neural Networks.* CRC press.

T. Hariprasad, G. Vidhyagaran, K. Seenu, and Chandrasegar Thirumalai. 2017. Software complexity analysis using Halstead metrics. In *Proceedings of the International Conference on Trends in Electronics and Informatics*. 1109–1113. https://doi.org/10.1109/ICOEI.2017.8300883

Mark Harman. 2010. Why Source Code Analysis and Manipulation Will Always be Important. *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation*, 7–19. https://doi.org/10.1109/SCAM.2010.28

Rachel Harrison, Steve Counsell, and R. Nithi. 1998. Coupling metrics for object-oriented design. In *Proceedings of the International Software Metrics Symposium*. 150–157. https://doi.org/10.1109/METRIC.1998.731240

Martin Hitz and Behzad Montazeri. 1995. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of the International Symposium on Applied Corporate Computing*. 1–10.

Keisuke Hotta, Yui Sasaki, Yukiko Sano, Yoshiki Higo, and Shinji Kusumoto. 2012. An empirical study on the impact of duplicate code. *Advances in Software Engineering* 2012 (2012). https://doi.org/10.1155/2012/938296

J. D. Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering* 9, 3 (2007), 90–95. https://doi.org/10.1109/MCSE.2007.55

Philipp Jahoda. 2021. MPAndroidChart. https://github.com/PhilJay/MPAndroidChart. [Accessed 29 June 2023].

Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the International Conference on Software Engineering*. 672–681.

Anders Krogh. 2008. What are artificial neural networks? *Nature Biotechnology* 26, 2 (2008), 195–197.

Sébastien Lapierre, Bruno Laguë, and Charles Leduc. 2001. Datrix™ Source Code Model and Its Interchange Format: Lessons Learned and Considerations for Future Work. *SIGSOFT Softw. Eng. Notes* 26, 1 (jan 2001), 53–56. https://doi.org/10.1145/505894.505907 [Accessed 29 June 2023].

Wox Launcher. 2022. Wox. https://github.com/Wox-launcher/Wox. [Accessed 29 June 2023].

Jeannette Lawrence. 1993. *Introduction to Neural Networks*. California Scientific Software.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (May 2015), 436–444. https://doi.org/10.1038/nature14539

Andreas Ludwig and Dirk Heuzeroth. 2001. Metaprogramming in the Large. In *Generative and Component-Based Software Engineering*, Greg Butler and Stan Jarzabek (Eds.). Springer, 179–188.

Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

Hao Men. 2018. *Fast and Scalable Change Propagation through Context-Insensitive Slicing*. PhD dissertation. https://doi.org/10.11575/PRISM/34987 [Accessed 29 June 2023].

Microsoft. 2022a. C# Language Specification. https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/. [Accessed 29 March 2023].

Microsoft. 2022b. Transact-SQL Reference. https://docs.microsoft.com/en-us/sql/t-sql/language-reference?view=sql-server-ver15. [Accessed 29 March 2023].

Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, Laurence Duchien, and Alban Tiberghien. 2010. From a Domain Analysis to the Specification and Detection of Code and Design Smells. *Formal Aspects of Computing* 22, 3 (May 2010), 345–361. https://doi.org/10.1007/s00165-009-0115-x

Daniel L. Moise and Kenny Wong. 2005. Extracting and representing cross-language dependencies in diverse software systems. In *Proceedings of the Working Conference on Reverse Engineering*. 209–218. https://doi.org/10.1109/WCRE.2005.19

Berndt Müller, Joachim Reinhardt, and Michael T. Strickland. 1995. *Neural Networks: An Introduction*. Springer.

Zaigham Mushtaq and Ghulam Rasool. 2015. Multilingual source code analysis: State of the art and challenges. In *Proceedings of the International Conference on Open Source Systems & Technologies*. 170–175. https://doi.org/10.1109/ICOSST.2015.7396422

.NET Foundation. 2022. eShopOnContainers. https://github.com/dotnet-architecture/eShopOnContainers. [Accessed 29 June 2023].

Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gundefinedrba. 2005. The Story of Moose: An Agile Reengineering Environment. *SIGSOFT Software Engineering Notes* 30, 5 (Sept. 2005), 1–10. https://doi.org/10.1145/1095430.1081707

Matthias Noback. 2018. The Liskov Substitution Principle. In *Principles of Package Design: Creating Reusable Software Components*. Springer, 31–53. https://doi.org/10.1007/978-1-4842-4119-6_3

Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2 ed.). Pragmatic Bookshelf, Raleigh, NC. https://www.safaribooksonline.com/library/view/the-definitive-antlr/9781941222621/ [Accessed 29 June 2023].

Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

David M. W. Powers. 2020. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. https://doi.org/10.48550/ARXIV.2010.16061

Powershell. 2022. Powershell. https://github.com/PowerShell/PowerShell. [Accessed 29 June 2023].

Python Software Foundation. 2008. Python Documentation: random. https://docs.python.org/3/library/random.html. [Accessed 3 July 2023].

Python Software Foundation. 2022. Python Language Reference. https://docs.python.org/3/reference/index.html. [Accessed 29 March 2023].

Gopinath Rebala, Ajay Ravi, Sanjay Churiwala, Gopinath Rebala, Ajay Ravi, and Sanjay Churiwala. 2019. Machine learning definition and basics. In *An Introduction to Machine Learning*. Springer, 1–17. https://doi.org/10.1007/978-3-030-15729-6_1

SAP. 2023. SAP Crystal Reports: Business intelligence reporting tools. https://www.crystalreports.com/ [Accessed 29 June 2023].

Hagen Schink. 2013. sql-schema-comparer: Support of multi-language refactoring with relational databases. In *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation*. 173–178. https://doi.org/10.1109/SCAM.2013.6648199

Shadowsocks. 2021. Shadowsocks Windows. https://github.com/shadowsocks/shadowsocks-windows. [Accessed 29 June 2023].

Muhammad Rabee Shaheen and Lydie du Bousquet. 2008. Relation between Depth of Inheritance Tree and Number of Methods to Test. In *Proceedings of the International Conference on Software Testing, Verification, and Validation*. 161–170. https://doi.org/10.1109/ICST.2008.34

ShareX. 2022. ShareX. https://github.com/ShareX/ShareX. [Accessed 29 June 2023].

Janet Siegmund, Norbert Siegmund, and Sven Apel. 2015. Views on Internal and External Validity in Empirical Software Engineering. In *Proceedings of the IEEE/ACM IEEE International Conference on Software Engineering*. 9–19. https://doi.org/10.1109/icse.2015.24

Kanishka Singh. 2021. *Scalable Encoding of Modularized Dependency Graphs for Fast Analysis*. MSc thesis. University of Calgary. https://doi.org/10.11575/PRISM/10182 [Accessed 29 June 2023].

Pramod Singh, Avinash Manure, Pramod Singh, and Avinash Manure. 2020. Introduction to TensorFlow 2.0. In *Learn TensorFlow 2.0: Implement Machine Learning and Deep Learning Models with Python*. Springer, 1–24.

Yoeo Somda. 2021. Guesslang. https://github.com/yoeo/guesslang. [Accessed 29 June 2023].

Spring. 2022a. Spring Boot. https://github.com/spring-projects/spring-boot. [Accessed 29 June 2023].

Spring. 2022b. Spring Framework. https://github.com/spring-projects/spring-framework. [Accessed 29 June 2023].

Square. 2022a. OkHttp. https://github.com/square/okhttp. [Accessed 29 June 2023].

Square. 2022b. Retrofit. https://github.com/square/retrofit. [Accessed 29 June 2023].

Guy Steele. 1990. *Common LISP: the language*. Elsevier.

Dennis Strein, Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. 2007. An Extensible Meta-Model for Program Analysis. *IEEE Transactions on Software Engineering* 33, 9 (2007), 592–607. https://doi.org/10.1109/TSE.2007.70710

Xiaobing Sun, Bixin Li, Hareton Leung, Bin Li, and Junwu Zhu. 2015. Static change impact analysis techniques: A comparative study. *Journal of Systems and Software* 109 (2015), 137–149. https://doi.org/10.1016/j.jss.2015.07.047 [Accessed 29 June 2023].

Nikita Synytskyy, James R. Cordy, and Thomas R. Dean. 2003. Robust multilingual parsing using island grammars. In *Conference of the Centre for Advanced Studies on Collaborative Research*.

Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall, and Andy Zaidman. 2019. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering* 25, 2 (Nov. 2019), 1419–1457. https://doi.org/10.1007/s10664-019-09750-5

Mark Weiser. 1981. Program Slicing. In *Proceedings of the International Conference on Software Engineering*. 439—449.

Kurt D. Welker. 2001. The software maintainability index revisited. *CrossTalk* 14 (2001), 18–21.

Jake Wharton. 2020. ButterKnife. https://github.com/JakeWharton/butterknife. [Accessed 29 June 2023].

Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer. https://doi.org/10.1007/978-3-642-29044-2

Hyrum K. Wright, Miryung Kim, and Dewayne E. Perry. 2010. Validity concerns in software engineering research. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM. https://doi.org/10.1145/1882362.1882446

Guo-Xun Yuan, Chia-Hua Ho, and Chih-Jen Lin. 2012. Recent advances of large-scale linear classification. *Proceedings of the IEEE* 100, 9 (2012), 2584–2603.