AN EMPIRICAL STUDY OF CONCURRENT FEATURE USAGE IN GO

by

Dhana Srimanthini Tipirneni

December, 2022

Director of Thesis: Nasseh Tabrizi, PhD

Major Department: Computer Science

The Go language includes support for running functions or methods concurrently as *goroutines*, which are lightweight threads managed directly by the Go language runtime. Go is probably best known for the use of a channel-based, message-passing concurrency mechanism, based on Hoare's Communicating Sequential Processes (CSP), for inter-thread communication. However, Go also includes support for traditional concurrency features, such as mutexes and condition variables, that are commonly used in other languages. In this paper, we analyze the use of these traditional concurrency features, using a corpus of Go programs used in earlier work to study the use of message-passing concurrency features in Go. The goal of this work is to better support developers in using traditional concurrency features, or a combination of traditional and message-passing features, in Go.

AN EMPIRICAL STUDY OF CONCURRENT FEATURE USAGE IN GO

A Thesis

Presented to The Faculty of the Department of Computer Science

East Carolina University

In Partial Fulfillment of the Requirements for the Degree

Master of Science in Computer Science

by

Dhana Srimanthini Tipirneni

December, 2022

AN EMPIRICAL STUDY OF CONCURRENT FEATURE USAGE IN GO

by

Dhana Srimanthini Tipirneni

APPROVED BY:

DIRECTOR OF THESIS: _____

Nasseh Tabrizi, PhD

COMMITTEE MEMBER: _____

Rui Wu, PhD

COMMITTEE MEMBER: _____

Nic Herndon, PhD

COMMITTEE MEMBER: _____

Mark Hills, PhD

CHAIR OF THE DEPARTMENT

OF COMPUTER SCIENCE: _____ Venkat Gudivada, PhD

DEAN OF THE

GRADUATE SCHOOL: _____ Kathleen Cox, PhD

ACKNOWLEDGEMENTS

I could not have undertaken this journey without the immense support provided by Dr. Mark Hills, who constantly believed in me and patiently directed me through the research process. I greatly benefited from his knowledge, understanding, and patience. He provided timely and instructive comments and evaluations at every stage of my thesis process, allowing me to complete this project on schedule. I also want to thank Dr. Nasseh Tabrizi, who guided me with his vast experience and offered suggestions throughout my thesis. I'm grateful for my family, especially my brother. Their constant belief in me has kept my spirits and motivation high during this process. I would also like to thank my dog for all the entertainment and emotional support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

Chapter 1

Introduction

1.1 Background

Concurrency refers to a program's ability to be divided into segments that can run independently of one another. Concurrency is critical in today's programming. Websites must support a large number of concurrent users. Some of the processing for mobile apps must be done on servers (in the cloud). Background work which does not interrupt the user is required for graphical user interfaces [69]. Eclipse, for example, executes your Java code while you are still modifying it. Concurrent programming is commonly done using two models: shared memory and message passing. The message-passing and shared-memory models describe how concurrent modules communicate with one another. The common model is shared memory, but this leads to race conditions, so newer models, such as message passing models, are becoming more common. Concurrent modules are classified into two types: processes and threads. Concurrency allows programs to handle multiple tasks at the same time. However, writing concurrent applications is difficult. Dealing with constructs like threads and locks, as well as avoiding problems like race conditions and deadlocks [62], can be time-consuming, making concurrent systems difficult to create.

Go is a new programming language created by Robert Griesemer, Rob Pike, Ken Thompson, and others at Google. Go was released as open source in November 2009, was named "Language of the Year" in 2009 [8], and received the Bossie Award in 2010 for "best open source application development software" [81]. It has a garbage collector and is strongly

typed. Concurrent programming is well supported in Go.

Go is a statically typed imperative programming language focused on systems programming, but with a novel concurrency model. Its defining characteristics are lightweight threads (goroutines) and communication channels [46] . The communication channel synchronization method was inspired by theoretical models of concurrency such as Hoare's communicating sequential processes (CSP) and is reminiscent of Milner's calculus of communicating systems (CCS) [53] and π - calculus [55]. This emphasis on channel-based communication aids in the development of concurrent programs that are conceptually simpler and better suited to being automatically checked to ensure the absence of communication faults like deadlock and thread starvation [54]. The Go language and its related infrastructure, however, do not provide any means of detecting concurrency issues aside from a standard type system and a runtime global deadlock detector.

1.2 Motivation for this study

Popular programming languages such as Java and Python implement concurrency by using threads. On the other hand, Go has built-in concurrency constructs: goroutines and channels. Goroutines communicate with each other through channels. Communicating Sequential Processes [30] is used to describe how systems that feature multiple concurrent processes should interact with one another. Thus taking advantage of these features of Golang, our empirical study aims to get a better understanding of how traditional concurrency features of the Go language are applied in practice by studying publicly available Go projects from GitHub taken from a previous work[16] who analyzed the usage of message-passing primitives. These conclusions can be used to guide future research in the area of static or dynamic verification of concurrent programs. Our research will enable researchers and practitioners to make well-informed decisions on the scalability (towards larger programs) and applicability (towards a larger subset of Go) of their approaches. We created a matcher that analyzes Go codes, which we tested on 856 GitHub Go projects. This paper provides the findings of our

investigation, which is organized around two research questions arising from the analysis of concurrent programs.

1.3 Research Questions

In order to lay the groundwork for future work on program comprehension and transformation tools, this research aims to provide answers to the following research questions.

RQ1: How often do any traditional concurrency features appear in Go projects?

RQ2: In projects that make use of traditional concurrency features, what percent of files include at least one declaration or use of a traditional concurrency feature?

1.4 Contribution

There have been several studies on how concurrency is achieved in different languages, including Go, that is discussed in Chapter 3. However, to the best of our knowledge, prior studies haven't focused on analyzing traditional concurrency features across open projects. The results of this empirical study can be used as a foundation for developing a system that would help researchers to get a quick overview of the concurrent programs. We made use of a tool - "GoAst- Viewer" [86], available online, which helped in understanding the Abstract Syntax Tree as effortlessly and quickly as possible. Description of the data collected and the code designed to facilitate our research are publicly available on GitHub at https://github.com/PLSE-Lab/Go-Concurrency under an open-source license.

1.5 The structure of the thesis

The rest of the thesis is structured as follows: Chapter 2 gives background information which starts with the history of Golang and describes the key features. This chapter also gives an overview of the entire key concepts we use in the thesis focusing on Go's powerful and rich concurrency features along with the tool we used. Then, related work is described in Chapter

3. The current works about concurrency and Go language along with the various tools that are developed in the existing works are examined in brief. Further, in Chapter 4, we describe the techniques and processes used for extracting data from various open source projects, the features of the corpus, along with the flow of the execution. Chapter 5 presents the brief methodology, results of our work, discusses them in detail, and then answers the research questions that are posed in Chapter 1. This chapter also explains the internal and external threats to validity. Further, it consists of the potential applications that can be developed from the work. Finally, Chapter 6 presents our conclusions, and discusses possibilities for extending our work in the future.

Chapter 2

Overview

Chapter 2 explains the background of the essential concepts of the main topics such as the Go Language and concurrency that are used in this thesis.

2.1 The Go Programming Language

2.1.1 History of the Go Language

Developers, computer scientists, and engineers have always sought strategies to make their work easier to handle, which is now essential for programmers. This demand drives the continuous and expanding flow of new programming language development, some of which are intended to address a particular issue. Here, we have come across 'The Go Programming Language', and we investigate it in a particular domain to see what it offers and how it is used.

Contributing to open source projects is a typical recommendation for budding developers. Even though there might be a lot of people out there, it could be very intimidating to contribute to any of the crucial system tools that execute on Linux systems. Many of the essential Linux tools, such as TOP command, were created in C and have been around for a while. It is certainly difficult to work with mature C code, which has caused some developer groups to believe that there must be a simpler method to create and maintain tools on Linux. The computing industry has made enormous progress in the present era. This has enabled programmers to construct programming languages that are better suited for

particular tasks. Go Language was introduced in the year 2009 by three Google employees, Robert Griesemer (renowned for the Java Hotspot Virtual Machine), Rob Pike (known for Unix and Plan 9 at Bell Labs), and Ken Thompson(known for C and Unix at Bell Labs). The Google-born Go, an open-source programming language, that has recently attracted a lot of interest, is sometimes referred to as "the programming language of the Cloud" and is one of the programming languages with the greatest rate of growth in the software industry. Go is recognized for its channel-based concurrency mechanism and strong support for system programming. It is marketed as an "An open-source programming language that makes it easy to create reliable, simple and effective software" [25]. Go has developed into a crucial component of much contemporary software due to the rising prevalence of containerization technologies in the software industry these days.

Since May 2010, Go Language has been used in practice at Google for back-end infrastructure, such as creating programs for sophisticated environment administration. However, many corporations and start-ups are now using it to create powerful apps. Go is loosely based on the C programming language. It was created at Google with the objective of creating a language free of the complexity and ambiguity of the C++ language. Golang, like C, supports structs, pointers, type inheritance, and operator and method overloading. What distinguishes Go is that it was designed from the bottom up to give performance capable of competing with languages such as C/C++ while supporting a relatively basic syntax similar to dynamic languages such as JavaScript.

As in any other programming language, in Golang, big programs are built from a small number of core constructs. Variables are used to hold data values, while data types are employed depending on the data and program. Operators like addition and subtraction are used in Golang to chain simple expressions into larger, more sophisticated expressions. We will look into detailed syntax in the next section.

2.1.2 Why Go Language?

Golang is a growing programming language with in-demand abilities, therefore it makes sense to study Golang in 2022 to produce better software in less time. Golang is widely used at Google production and in many open-source applications. According to the popularity index, it is currently placed 13th, up from 65th two years ago [26]. It has entered into the list of Top 10 Programming language by IEEE Spectrum after Python, Java, C, C++, JavaScript, C#, and R and became the fourth most active language on GitHub in 2021 [6]. It is intended to simplify the software development process, particularly for complicated architecture and operations. That is why major tech companies and multinational corporations (MNCs) such as Google, Facebook, Docker, Apple, and others are embracing it.

As Moore's Law approaches its end of lifecycle, the necessity for concurrency grows, as does the need for a programming language that permits effective execution of concurrent systems. As a result, Go has recently become one of the most popular programming languages. This has made it quite popular among other companies dealing with enormous scaling issues. It's also one of the most popular programming languages in the previous ten years[57]. The language was created by and for those who build, read, debug, and maintain massive software systems. It has modern features from Java, such as garbage collection, and it also take full advantage of advanced multi-core computational capabilities including built-in concurrency functionality, which is similar to Java. Today, Go is designed for software development. It is simple, quick, easy to learn, and dependable. Go developers are among the highest-paid in the field, with an average annual compensation of $135,750 [26]. These benefits make learning Go Language worthwhile. Hence we have chosen to work on our thesis in Go language

2.1.3 Goals of Go Language

Today's software industry is dominated by two trends. First, the number of CPU cores is increasing. Second, high-performance software is developed in a language like C++, which

is a very sophisticated language with precise specifications that can span over a thousand pages[36].

Golang was created in an environment keeping the requirements in mind. The fundamental goal of the Go developers (all from Google) was to combine the simplicity of programming provided by an interpreted, dynamically-typed language (such as Python and Ruby) with the efficiency and safety provided by a statically-typed(such as C++), compiled language. However, it must be able to address the major problems while dealing with the company's complex systems. Golang's second goal is to support the recent trend of distributed and multicore computing by providing networking, concurrency, and parallelism support. The third goal is to reduce the time it takes to compile software in order to overcome the lengthy test-develop cycles experienced by Google's C++ developers.

Furthermore, when it comes to executing code, Golang aims to match the speed of C/C++. Unlike C++, which provides only the most basic memory management and asks the developer to handle it in his code, Golang is a garbage collected language, while C/C++ require manual memory allocation and deallocation. The inventors prioritized simplicity in order to attain these objectives. As a result, the syntax of the Go language is inspired by the C family. Other input, such as Pascal, can be seen in the declarations and packages. Newsqueak and Limbo [65] were other inspirations, as they contributed to the concurrency component of Go. These merits make the Go language a combination of the best features of each language.

2.1.4 Characteristics of Go Language

Open-Source, Simplicity, and Consistency : The primary feature of the Golang programming language is that it is open-source. That implies anyone can download and experiment with the code to incorporate better codes and fix associated errors. In comparison to other languages, Go is a fairly basic and consistent language [67]. The creators of Golang add only those aspects to the language that are relevant and do not complicate the language by

introducing other items.

Concurrency Support : It is one of the key features of the Go programming language. Unlike other programming languages, Golang provides simpler and more trackable concurrency settings. This allows app developers to execute requests more quickly, free up allotted resources and networks early, and much more. Channel-based concurrency, based on Hoare's Communicating Sequential Processes, is the main fundamental unit in the Go Language for achieving concurrency in a developing project. Goroutines are lightweight and simple to employ in the program.

Testing Capabilities : Along with writing app code, the Go language allows you to construct unit tests. It also helps you understand code coverage and benchmark tests, as well as write example code to create code documentation. Go includes a method for testing the packages you create. You can test your code written in "*_test.go" files with just the "go test" command.

Garbage Collection : The programming language also has amazing trash collection capabilities. This eliminates the need for developers to worry about freeing up pointers or dealing with the consequences of a dangling pointer [19].

Go has implicit interfaces: The implementation of implicit interfaces (technically known as structural typing) allows Go to enforce type-safety and decoupling while retaining much of the flexibility found in dynamic languages. Explicit interfaces, on the other hand, link the client and implementation together, making it significantly more difficult to replace a dependency in Java, for example, than in Go.

Static Typing : Go is a statically typed programming language that uses a mechanism to compile code accurately while handling type conversions and compatibility levels. This liberates developers from the difficulties involved with dynamically typed languages[67].

Handling errors: Errors in Go are handled differently than in other languages. In summary, Go handles errors by returning a value of type error as the function's final return value.

Object-Oriented Nature : Although Go does not contain the concept of classes and objects, structs are often utilized as a replacement for classes in Golang. In short, Go is not an Object-Oriented language, although it has the impression of one [19].

Powerful Standard Library and Tool Set : This programming language also has a large standard library. These libraries include a plethora of components that allow developers to avoid relying on third-party packages. It also provides a broader set of tools to make the development process more efficient. This includes Gofmt, Gorun, Goget, and Godoc.

2.1.5 Key Concepts in Go Language

In this section we will into the basic structure and key concepts that are involved in the Go Programming Language.

Program Structure and Syntax

A Go program is made up of the following components.

- Package Declaration
- Comments
- Import Packages
- Variables
- Functions
- Statements and Expressions

Golang has a syntax similar to C and constructs such as channels and goroutines. However, it is clean and mimics how simple it is to develop Python code. When compared to other programming languages, Golang's simplicity makes writing programs quite simple [14]. We will use an example to demonstrate some of the constructs and observe the language's features.

Listing 2.1: Hello World Program

```
package main
import "fmt"
func main () {
        fmt.Printf( "hello,world\n" )
}
```

Golang arranges programs into packages and includes import lines at the top of each one. Furthermore, Golang identifies unused imports as errors; for example, importing the OS package in the hello world program would be an error. By limiting imports, the program becomes extremely efficient.

Channels

A channel is a programming concept in Go that allows us to transport data across executing threads in our program, typically from distinct goroutines [50]. You can transmit values into channels through one goroutine and receive them from another. Channels are pipes that connect many goroutines.

Go Routines

A goroutine is a lightweight thread that the Go runtime manages. Goroutines are functions or processes that run alongside other functions or procedures. Goroutines can be compared to lightweight threads. When compared to a thread, the cost of starting a Goroutine is negligible. To initialize a goroutine, around 2kB of stack space is required. A typical thread, on the other hand, can consume up to 1MB of memory, therefore establishing a thousand goroutines requires far fewer resources than creating a thousand threads [77]. As a result, many Goroutines can operate concurrently in Go applications. It should be noted that concurrent execution may or may not be parallel. Every Go program contains at least one goroutine: the main goroutine. The go keywords are used to launch a goroutine.

Concurrency has existed for a long time in the form of Threads, which are now used

in practically all applications. Some of the advantages of Goroutines over threads are as follows:

- On a normal system, you can execute more goroutines than threads.
- Goroutines feature segmented stacks that can be expanded.
- Goroutines are faster to start than threads.
- Goroutines have primitives built in to allow them to communicate safely with one another (channels).
- When sharing data structures, goroutines allow you to avoid using mutex locking.
- Instead of a 1:1 mapping, goroutines are multiplexed onto a small number of OS threads.
- You can construct massively concurrent servers without resorting to event-driven programming.

Message Passing

Message passing is a technique in computer science for invoking activity (i.e., running a program) on a computer. The invoking software delivers a message to a process (which could be an actor or an object) and then relies on that process and its infrastructure required to select and execute some code. Message passing varies from traditional programming in that a subroutine, process, or function is called directly by name. Message passing is essential in some concurrent and object-oriented programming frameworks [63]. Message passing is common in today's computer applications. It is used to allow the objects that comprise a program to communicate with one another, as well as to allow objects and systems running on other computers (e.g., the Internet) to communicate.

Message passing is a programming technique that is widely used. This is the preferred concurrency mechanism for Go. In terms of message passing, we may understand that we generate messages that are received by the consumer, and the message is built and consumed only once [83]. When you insert a message into a channel, the action is halted until the

consumer reads it. Channel is similar to a pipeline that goroutines use to interact with one another; it aids in solving the problem of resource sharing among goroutines.

Communicating Sequential Process

C. A. R. Hoare created Communicating Sequential Processes (CSP) [30] in 1978. Hoare proposes the concept of inter-process communication via channels. CSP's channels are unbuffered, which means that a process waits until another process is ready to input or output on a channel. CSP channels are also used as guardians for process synchronization: a process listens on many channels for input and proceeds once a signal arrives on a channel. CSP's approach to concurrency and inter-process communication influenced several other languages, including Occam [9] and Erlang [79].

CSP pioneered the concept of inter-process communication channels (not in the original paper but in a later book on CSP, also by Hoare [30]). Channels enable CSP-style message transfer techniques [24] for inter-goroutine communication. Although traditional shared memory concurrency is also supported, channel-based techniques are strongly advised instead. It was thought that message-passing communication may make concurrent programming easier and more dependable.

Why build concurrency on the ideas of CSP? [23]

Concurrency and multi-threaded programs have earned a bad reputation over time. This is due to a combination of sophisticated architectures like threads and an overemphasis on low-level features like condition variables, mutexes, and memory barriers. Even if there are still mutexes and the like behind the covers, higher-level interfaces allow for much easier programming.

CSP has one of the most successful methods for providing high-level syntactic support for concurrency. CSP gave rise to two well-known languages: Occam and Erlang. The concurrency primitives in Go are descended from a distinct branch of the family tree, the key contribution of which is the powerful concept of channels as first-class objects [3] meaning

they can be send as values on channels, allowing them to be shared between goroutines. Experience with previous languages has demonstrated that the CSP style works well in a procedural language architecture.

2.1.6 Comparison to other Languages

The concurrency mechanism of Go is probably the most noticeable difference between it and many other programming languages. Traditional general-purpose programming languages allow you to run many functions concurrently by using threads given and scheduled by the operating system (or a rather abstract idea of "workers" based on OS threads) [34].

- Those threads typically have a memory stack of a few megabytes, which means you can't spawn too many of them; for example, 1000 threads each consuming 1 MB of memory would require at least GB of memory.
- Acquiring a thread from the OS isn't cheap, so take as many as you need during the program's initialization step and reuse them.
- Context changes on OS threads are not inexpensive. Most registers and caches will need to be replaced.

Because to the restrictions of OS threads, designing non-blocking concurrent systems is excessively difficult. Go has an entirely separate concurrency mechanism that is related to Erlang in certain ways. Go allows you to run functions in distinct "goroutines" [32].

- Goroutines in Go feature a flexible stack with at least 2kb of memory that expands as needed. This means that instead of thousands of threads, you can create millions of them.
- In the built-in run-time, goroutines are multiplexed over an OS thread pool, achieving 99.9% CPU usage.
- They are scheduled cooperatively, which means that when they are stalled, they restore control to the built-in user-space scheduler (by, for example, a database operation)
- Context switches in goroutines are particularly inexpensive because fewer registers must be swapped out.
- Because everything is done at the application level and does not involve a system-space round-trip, creating and suspending goroutines is incredibly cheap.

Goroutines make it easy to write non-blocking, massively concurrent programming. When developing a server, for example, you may handle each connection in its own goroutine because they're so inexpensive, and you won't need to utilize complex IO multiplexing mechanisms like epoll or similar because Go provides this for you internally [2]. Writing blocking Go code is perfectly acceptable because a goroutine will be instantly swapped out for another when it becomes stalled without blocking the CPU. There is no async/await, promises, callbacks, thread pools, or tasks; simply blocking code.

Unlike most modern programming languages, Go is notorious for the capabilities it lacks. It's a minimalistic language focusing on simplicity, which distinguishes it from others. In comparison to traditional programming languages such as Java/C/C++, because it is new, Go does not need to carry many legacies and does not need to be backward-compatible with programs developed in, say, 1999 [38]. It's also one of the few compiled languages that can be compiled as quickly as some interpreters. It does not optimize the code very effectively, achieving a balance between optimization and compilation speed. It's also one of the few constructed languages with a built-in runtime that houses the garbage collector, scheduler, and so on.

There are, of course, others.

- There are numerous syntactic differences.
- Generics are built-in (Go has built-in generics on built-in primitives like maps and slices, but you are not (yet) allowed to create your own generic structures).
- Error handling is a little different. It is done by returning a value of type error.
- Dependency management is decentralized and unique.
- The language and its toolchain already include testing, benchmarking, and profiling tools.
- Gofmt generates documentation directly from the code, allowing you to keep coding style consistent throughout all Go projects.
- Go takes a distinct approach to OOP.

Most importantly what distinguishes Go is the combination of compilation to actual binary executables, the runtime, the concurrency model [75], the garbage collector, and the language's specific characteristics.

2.2 Traditional Concurrency Features

As Goroutines share the same address space[45], access to shared memory must be synchronized. The sync package provides the concurrency primitives that are useful for synchronizing low-level memory accesses [70]. The primitives that it contains are explained below:

Wait Group: A WaitGroup waits for the completion of a collection of goroutines. The main goroutine utilizes `Add` to determine how many goroutines to wait for. Then, once finished, each of the goroutines calls `Done`. `Wait` can be used to block the program until all goroutines have completed [51].

Mutex: A Mutex is a mutual exclusion lock that prohibits other processes from accessing a critical section of data while one is occupying it in order to prevent race conditions. A critical section can be a chunk of code that cannot be executed by multiple threads at the same time because it involves common resources [21]. There are two main methods for accessing mutex - `Lock()` acquires or holds the lock, `Unlock()` releases the lock.

RWMutex: A reader/writer mutual exclusion lock is an RWMutex. An arbitrary number of readers or a single writer can hold the lock. In other words, readers are not required to wait for one another. They merely need to wait for the writers holding the lock [7]. It is therefore preferable to use `RWMutex` for data that is mostly read, as it saves time compared to `Mutex`.

Cond: The `Cond` condition variables can be used to coordinate resource-sharing goroutines. It can be used to alert goroutines that are blocked by a mutex when the state of shared resources changes. Each Cond contains a lock (usually a Mutex or RWMutex) that must be held when altering the condition and using the Wait method. One scenario in which the `cond` is required is when one process is getting data and other processes must wait for

this process to receive data before reading the correct data [28]. Only one process can wait and read the data if we only use a channel or mutex. There is no mechanism to notify other processes that the data has been read. Thus, we can sync.Cond to coordinate shared resources.

Once: Once ensures that only one execution will be carried out even among several goroutines. Unlike other primitives, `Once` only has a single method that is `Do(f func())` which calls the function f only once [21]. If Do is called multiple times, only the first call will invoke the function f.

Locker: The type `Locker` represents an object that can be locked and unlocked. This is a generalization of the `Mutex` and `RWMutex` types, used in some instances where the type of `Mutex` is not important (for instance, in a `Cond` variable).

Map: Map is similar to the standard `map[any]any`, but it is acceptable to use by many goroutines concurrently without additional locking or coordination. Loads, stores, and deletes happen in real time. Map is a specialized type. We need it mainly to achieve improved type safety and to make it easier to maintain other invariants with the map content, most code should instead use a plain Go map with separate locking or coordination.

The Map type is designed for two typical scenarios. The first one is when an entry for a specific key is only ever written once but read numerous times, as in growing caches. The second one is when more than one goroutine reads, writes, and overwrites entries for disjoint sets of keys [7]. The use of a sync.Map in these two circumstances, when combined with a separate Mutex or RWMutex, map may dramatically minimize lock contention. The zero Map is completely empty and ready for use. A Map must not be duplicated after the first time it is used.

Pool: Pool is a scalable temporary object pool that is also concurrency safe. Any saved value in the pool can be erased at any time without a notification. Furthermore, when there is a heavy load, the object pool can be dynamically increased, and when it is not used or the concurrency is low, the object pool shrinks. The fundamental concept is object reusing

to avoid repeated creation and destruction, which will decrease the performance.

The pool's goal is to cache allocated but unused items for eventual reuse, reducing the garbage collector's workload. That is, it facilitates the creation of efficient, thread-safe free lists. It is not, however, appropriate for all free lists. A Pool should be used to handle a group of temporary things that are discreetly shared among and potentially reused by concurrent independent clients of a package [70]. Pooling allows you to distribute the expense of allocation overhead across multiple clients. It is also important to note that Pool has a performance cost. Using sync.Pool instead of basic initialization is substantially slower. A Pool must also not be cloned after its initial use.

2.3 Channels

In the go programming language, a channel is a lock-free communication method used by a goroutine to communicate with another goroutine. Or to put it another way, a channel is a procedure that allows data to be sent from one goroutine to another [13]. The following figure illustrates how the default bidirectional channel allows goroutines to send and receive data across the same channel:

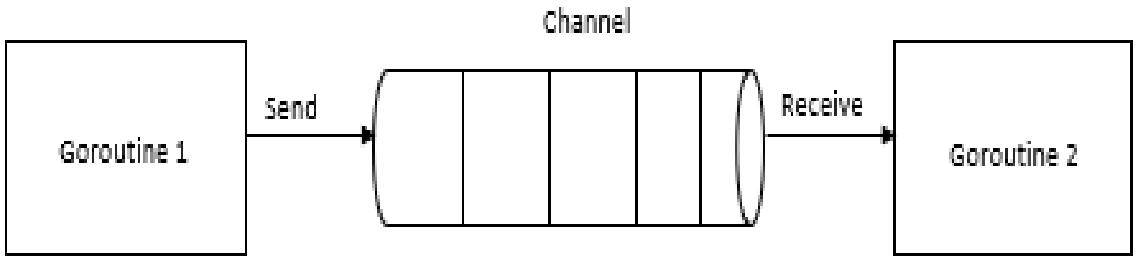

Figure 2.1: Channel In GoLang

Sending and receiving are the two main channel activities in the Go programming language. Together, these two actions are referred to as communication. The "← " operator's direction also determines whether the data is sent or received. Send and receive operations

in a channel are by default blocked until the opposite side is not ready [20]. It enables goroutines to synchronize with one another without using conditional variables or explicit locks.

Send Operation: Using a channel, the send operation is used to send data from one goroutine to another. Because they are copied, values like int, float64, and bool can be sent via a channel safely and easily with no chance of an unintentional concurrent access. In a similar manner, strings are likewise secure to transfer due to their immutability [17]. However, sending pointers or references across a channel is not secure because the value of pointers or references may change simultaneously by the sending and receiving goroutines, with unpredictable results. Therefore, one must make sure that any pointers or references they use in the channel can only access one goroutine at a time.

Mychannel ← element

The above statement indicates that the data(element) send to the channel(Mychannel) with the help of a ← operator.

Receive Operation: Receive operation: The data sent by the send operator is received using the receive operation.

element := ← Mychannel

The above statement indicates that the element receives data from the channel that is "Mychannel". If the outcome of the received statement is not going to be used, the statement is still valid. A receive statement might also be written as: ← Mychannel.

A channel is bidirectional by default, but you can also establish a channel that is unidirectional [50]. A unidirectional channel is one that only has one direction of data flow, either receiving or sending. The make() function can also be used to construct a unidirectional channel, as demonstrated below:

// Only to receive data c1:= make(← chan bool)

// Only to send data c2:= make(chan ← bool)

A bidirectional channel can be converted in the Go programming language into a unidi-

rectional channel, or to put it another way, you can change a bidirectional channel into a receive-only or send-only channel, but not the other way around.

Based on how they communicate data, channels can be divided into two categories:

Unbuffered Channels: There is no ability to store any value in an unbuffered channel before it is received. Both sending and receiving goroutine must be ready simultaneously in this type of channel before any send or receive action can be completed. The channel makes the goroutine that executes its corresponding transmit or receive operation first wait if the two goroutines aren't ready at the same time [22]. Synchronous communication between goroutines is accomplished using it. The interaction between the sends and receives on the channel depends on synchronization. Without the other, one cannot take place. An unbuffered channel ensures that a transaction between two goroutines is executed at the same time as the transmit and receive operations.

Buffered Channels: One or more values can be held in a buffered channel until they are received. These channels don't require goroutines to be prepared to send and receive data at the same time. Additionally, there are many circumstances under which a send or receive does block. Only in the event that the channel is empty would a receive will be blocked [20]. Only when there is no buffer available to place the value being transmitted will a send get blocked. Asynchronous communication is carried out through a buffered channel.

2.4 Select statement

The select statement is used to choose from multiple send/receive channel operations. Until one of the send/receive processes is complete, the select statement is inactive. When multiple operations are ready, one is picked at random. The syntax resembles a switch statement without input parameters, with the exception that each of the case statements will be a channel operation [66]. A case statement refers to communication, i.e., sent or receive operations on the channel.

Following are the key points about a select statement:

- In some circumstances, the select statement waits on starting until the communication (send or receive operation) begins.
- If a select statement doesn't include any case statements, it will wait forever.
- If none of the other cases are prepared for execution, the default case is carried out. Since operations are blocking by default, it stops the select from blocking the main goroutine.
- The blocking of the select statement means when there is no case statement is ready and the select statement does not contain any default statement, then the select statement blocked until at least one case statement or communication can proceed.
- If more than one case is prepared to move on, the select statement allows for the random selection of one of them.

2.5 Go Runtime Scheduler

Its responsibility is to spread executable Goroutines (G) among several worker OS Threads (M) that are executed on a single or multiple processors (P). Multiple threads are being handled by processors. Goroutines are handled by threads in groups. The number of processors is determined by the number of CPU cores; they are hardware dependent.

G = Goroutine

M = OS Thread

P = Processor

A goroutine is added to the list of runnable goroutines of the current processor when it is generated or when an existing goroutine is made runnable. The processor tries to pop a goroutine from its list of runnable goroutines when it has finished running a goroutine [13]. The processor picks a random processor and tries to take half of the runnable goroutines if the list is empty.

2.6 Abstract Syntax Tree

An Abstract Syntax Tree, or AST, is a tree representation of a computer program's source code that conveys the structure of the source code. Each node in the tree corresponds to a

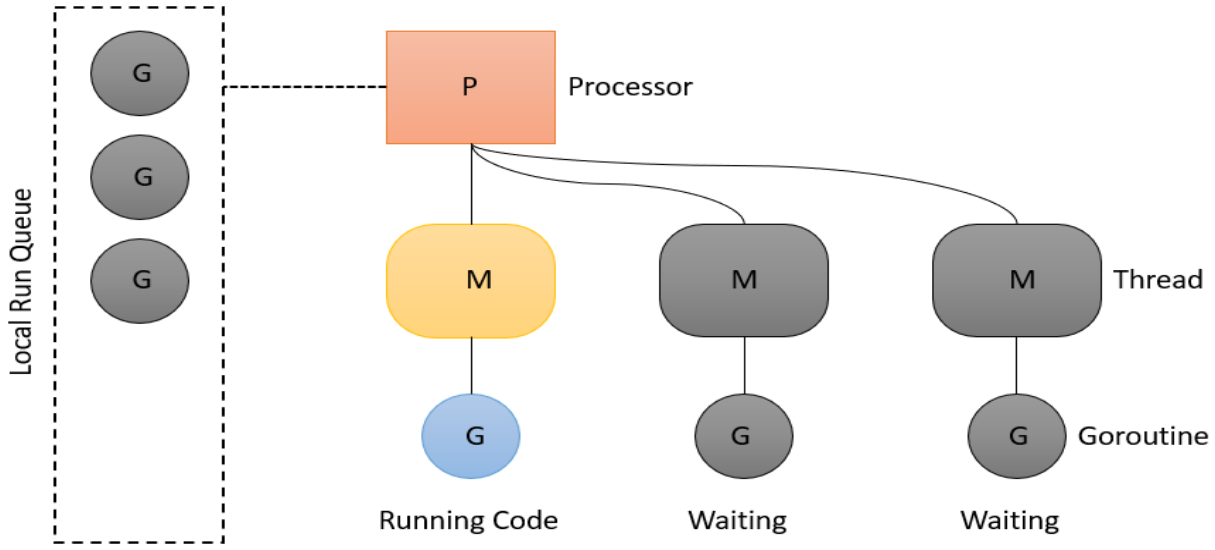

Figure 2.2: Go Runtime Scheduler Layout

construct found in the source code. Only the structural and content-related details of the source code are maintained during conversion to its AST, and all further details are deleted. [80] The following information is retained and is critical to the AST's mission:

- Variable types and variable declaration locations.
- The arrangement and formulation of executable statements.
- Left and right components of Binary operations.
- Identifiers and the values allocated to them.

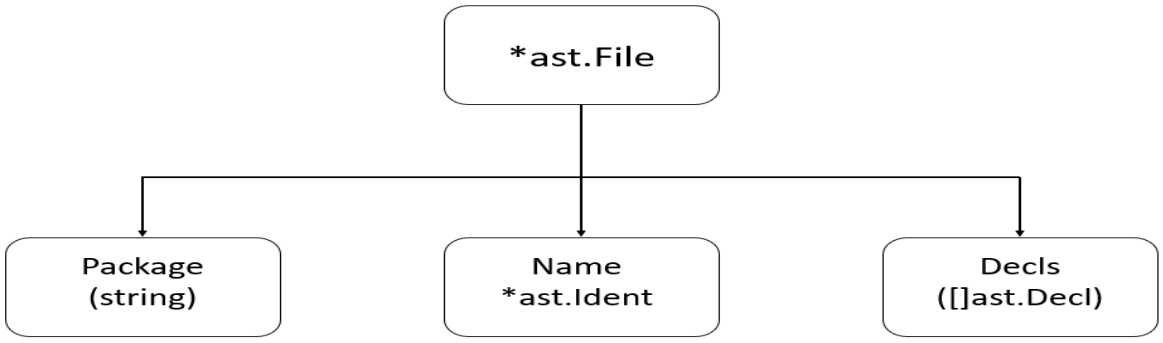

Figure 2.3: Basic Go AST structure of a single file

The Abstract Syntax Tree is constructed from a list of tokens generated during the Lexical Analysis phase of source code transformation (abstract syntax tree or even more strict concrete syntax tree) [44]. In general, converting source code to a tree structure consists of two steps:

- Lexical Analysis (done by a lexer, sometimes called a scanner)
- Syntax Analysis (done by a parser)

Lexical Analysis converts the source code given as input into a collection of lexical tokens. Lexical tokens, which include identifiers, keywords, comments, and literals, are the program's fundamental building blocks. During the lexical analysis phase, it is unknown how they fit together, that is, if they constitute a function or a variable declaration, or even if they do so correctly.

The lexical tokens are converted into the syntax tree by the parser during the Syntax Analysis phase. The language grammar defines the rules for how the programming language describes some abstractions syntactically [87]. If the lexical tokens' placement obeys those constraints, the parser attempts to combine them into the syntax tree (either abstract or concrete, also known as a parse tree). Some lexical token placement errors may occur, which is why the compiler raises syntax errors.

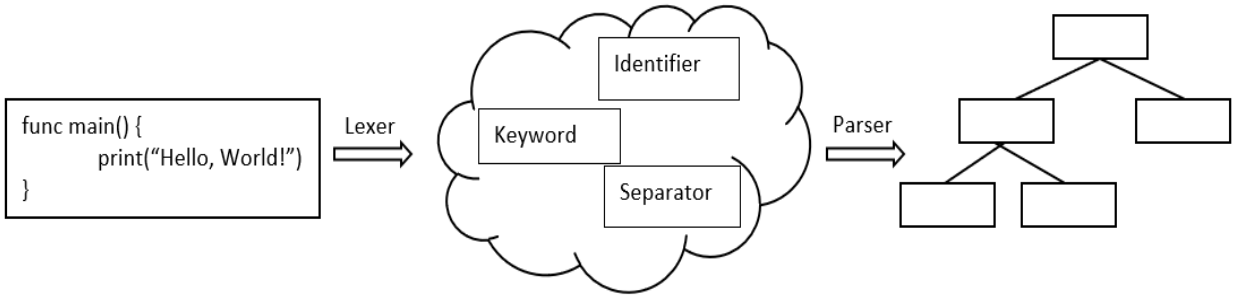

Figure 2.4: Syntax Tree generation overview

The Go programming language has several node types that describe Abstract Syntax Trees [44]. Every node holds information about the positions of lexical tokens in the source

code (this respects the position of braces and parentheses, which introduces some less abstract details to the AST structure). There are four types of nodes, which are distinguished by the following interfaces:

- 'Decl' is used for declaration nodes, such as FuncDecl for function declarations,
- 'Stmt' for statement nodes, e.g. AssignStmt for assignments,
- 'Spec' for specification nodes, e.g. ValueSpec for constant or variable declaration specifications,
- 'Expr' for expression nodes, e.g. BinaryExpr for binary expressions (like addition),
- 'Node' is a generic interface that every node in an AST implements.

Abstract Syntax Tree allows you to do complex source code inspections and modifications. Below are some of the examples of AST usage:

- Parsing the source code
- Printing an AST
- Traversing an AST
- Manipulating an AST
- Decorating an AST

In this thesis, we took advantage of the Abstract Syntax Tree of [86], which provided an interactive way of visualizing the declarations.

Chapter 3

Related Work

In Section 3.1 of this chapter, we first provide information about the work that has already been done in relation to the history of the Go language.In Section 3.2, we describe existing work regarding concurrency in Go and its features, and in Section 3.3, we explore existing work on tools and methodologies that have been developed and are currently in use to evaluate Go language code.

3.1 Go Language Usage and Popularity

The Go programming language was created in late 2007 as a solution to some of the issues Google was experiencing with the development of its software infrastructure [59]. It is so difficult to distinguish today's computing landscape from the one that existed when C++, Java, and Python were created. Go was created and developed to increase productivity in difficult settings [47]. Go is becoming increasingly popular for developing applications in production contexts. These Go programs include everything from command-line tools and libraries [15] to system software such as databases, container systems, and block-chain systems [39] . Along with its more well-known features like built-in parallelism and garbage collection, Go's design takes into account strict dependency management, the flexibility of the software architecture as systems expand, and robustness across component borders [58].

Go, as defined by Google, is expressive, brief, logical, and effective. Its innovative type system allows for flexible and modular program creation, while its concurrency techniques

make it simple to write programs that make full use of multi-core and networked devices. This language can also be compared to C [73]. Go feels as if Python served as some of the design inspiration for C when it was created in the current era. It aspires to be quicker to develop in than C after a user gets used to it.

3.2 Concurrency in Go

Concurrency, in computer science, is the capacity of separate elements or units of a program, method, or problem to be run out of order or in partial order without impacting the outcome [82]. This enables concurrent units to be executed in parallel, which can considerably enhance total execution speed in multi-processor and multi-core systems. Concurrency, in more technical terms, refers to the ability of a problem, algorithm, or program to be decomposed into partially-ordered components or order-independent units of computing [43]. Golang is an open-source programming language distributed under the BSD license. Golang was influenced by Erlang's concurrency primitive, which was built on Tony Hoare's concept of communicating sequential processes (CSP) [46].

Some previous attempts have dealt with channel-based concurrency and heap manipulating programs, but these are not immediately applicable to Go. Villard et al. [78] propose a sophisticated contract technique for specifying protocol requirements for channels. Their terminology for channel specification is more expressive than the one described in other papers. Their contracts are finite state machines, which means they can go through numerous phases. Their channels, however, are always shared between two peers, whereas Go enables more complex concurrency patterns in which both channel endpoints are shared among an infinite number of peers. Actris [29] , uses concurrent separation logic to make heap access and is built on the Iris framework. Actris could go beyond two entities, but it needs a memory model which is contradictory with Go's memory model to do so. Actris handles channel endpoint sharing using Iris' ghost locks, which, to our understanding, necessitates sequentialization of sends and dually receives, which Go's memory model does not ensure.

Concurrency is one of the main features of the Go programming language [18]. Concurrency in Go is well supported via goroutines and channels. From the experiment done by [76], it is shown that Go surpassed Java in terms of concurrency and compile-time performance. Go code also demonstrates how simple concurrent programming is. Go concurrency features include FIFO Queues, goroutines, and Select constructs [40]. [60] suggested how to check the program implemented by Go. The validation tool is a Communicating Sequential Processes(CSP) or Failures Divergence Refinement (FDR) refinement checker. An algorithm converts a Go program into CSP at first, and CSP writes its specification manually. The Go program is then tested to see if it fits to its specification by having both of them checked by FDR.

Despite Go's initial stages of development, it has obtained notable interest from the research community. The work of [56] is the first to address static verification of Go programs. In their work, they extract communicating finite state machines from source code and employ multiparty session types [31] and their connections to communicating automata [12], [42] to check for liveness in Go. However, neither asynchrony nor dynamic goroutine spawning can be supported by their work; rather, all goroutines must be run before any communication can occur. This greatly restricts the usefulness of their research. In addition, numerous Go features, such as phi instructions and uninitialized channels, are not covered by their analysis, which causes crashes [41].

3.3 Tools for analysis

Go is a modern programming language for concurrent systems. Meeting the challenge of multi-core concurrent programming is one of its objectives. Multi-thread programming is the primary solution to multi-core parallel programming [1]. However, building concurrent and parallel applications using the multi-threading architecture is very challenging. To illustrate the simplicity of multi-core parallel programming in Go and the effectiveness of parallel Go code, [74] offers two multi-core parallel Go programs in this paper together

with their runtime scheduler on an 8-core microprocessor were seen as highest than other methods. [85] presented Gobra, the very first modular verifier for Go that allows you to reason about a critical component of the language: the mix of heap-manipulating constructs and channel-based concurrency. Gobra is also the first validator to handle Go's interfaces and structural sub-typing. In future work, they plan on broadening the traits that Gobra can verify, focusing on liveness and hyper-properties. In addition, they are using Gobra to validate the implementation of a full-fledged network router [84]. Gobra's verification logic and Viper encoding were influenced by a number of other Viper-based verifiers, including Nagini [33] for Python, Prusti [4] for Rust, and VerCors [5] for Java. None of these verifiers handle the Go-specific capabilities supported by Gobra.

The focus on channel-based communication aids in the development of concurrent programs that are conceptually simpler and better suited to automatic verification to ensure the absence of communication errors like thread starvation and deadlock. However, the Go language and its related tooling do not provide any means to identify concurrent issues beyond a fairly conventional type system and a runtime global deadlock detector. Recently, several research teams have sought to close this gap by creating a variety of theories and tools designed to help programmers detect synchronization bugs in Go applications, either statically (at compile time) or dynamically (at runtime).

First, a tool to statically identify global deadlock in Go programs was put out by Ng and Yoshida [56] using choreographic synthesis [42]. Later, to identify global deadlocks, Stadtmuller et al. [68] introduced another static verification approach based on forkable regular expressions. Two more sophisticated static verification frameworks were suggested by Lange et al. [40], [41], which mimic Go programs with behavioural types [35] through their SSA intermediate representations. Through the use of exhaustive model checking in [41] and bounded executions in [40], different safety and liveness properties can be tested on behavioral types. For finding global deadlocks in a small subset of Go programs (without recursion), Midtgaard et al. [52] suggested a static verification approach based on abstract

interpretation. The dynamic verification of Go programs was covered by Sulzmann and Stadtmuller [[71], [72]].In [71], they offered a trace-based methodology to analyze Go programs that only use synchronous channels. In [72], they proposed a better method that supports asynchronous channels and relies on vector clocks. Both pieces of work demand that the code be instrumented before analysis. Unsurprisingly, only a portion of the Go language is supported by the static approaches discussed above. For instance, none of the frameworks for static verification in [41], [52], [56], or [68] can evaluate programs that create additional threads inside of a for loop. For these programs, the work in [40] only offers an inaccurate approximation. Additionally, only short Go programs or programs that use a limited number of message forwarding primitives have been used to demonstrate these approaches. Dynamic verification techniques might support a wider range of the language's capabilities because doing so only necessitates more instrumentation. However, intensive message passing primitive usage has an effect on them as well. For programs with high levels of concurrency, Sulzmann and Stadtmuller claim up to a 41% tracing overhead [72].

Chapter 4

Corpus

To investigate the use of concurrency features in Go code, we first needed to identify a corpus that could be analyzed. In this section, we describe the GitHub projects that we have gathered and the approach we have used to collect data. We have followed the same approach in collecting data that [16] followed in their paper. The corpus contains 856 systems that contain 500,322 Go source code files, with 132,746,554 lines of source code.

The main objective of our project is to find how developers and programmers use traditional concurrency features of Go language in their projects. For this we have selected a wide range of projects that [16] have used in their work.

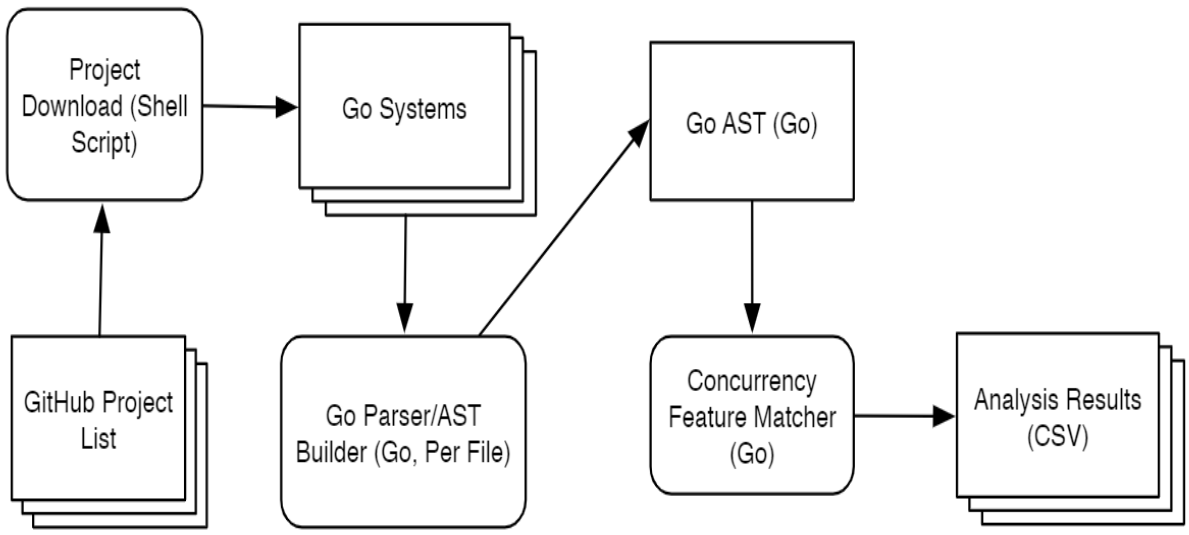

Figure 4.1: Overview of the model

Figure 4.1 gives an overview of the systems retrieval process. First, we have taken 856 systems on GitHub from [16],who used the criteria to select the projects that had the highest number of stars. So they are considered as the most popular Go projects, based on their stars. The number of stars generally indicates how many people appreciate it or are interested in that particular project. For all the systems that are considered, the highest starred repository is `go` under GitHub username `golang`. Summary of highest and lowest stars can be found in Table 4.1.

GitHub-Username/ Repo-Name	Stars	Topics
golang/go	104969	go,golang,language,programming-language
kubernetes/kubernetes	93083	cncf,containers,go,kubernetes
avelino/awesome-go	89771	awesome,awesome-list,go,golang,golang-library, hacktoberfest
moby/moby	64345	containers,docker,go
gin-gonic/gin	63774	framework,gin,go,middleware,performance, router,server

Table 4.1: Number of stars

We then executed a Shell Script code with the `git clone` command to retrieve the current version of each system from GitHub locally. The detailed code is given below in Listing 4.1. The project file containing the repositories is from the repository provided by the other repository. We are only retrieving the current state of the system, not any of the history. A `txt` file holds all the names of 856 projects, that feeds into the script and is used to download the projects.

Listing 4.1: Script to retrieve Files

```
#!/bin/bash

if [ ! -d logs ] d- directory, ! does not exist.
then
```

```
  mkdir logs
fi end of if

while read p; do
  git clone --depth 1 git@github.com:$p.git
done<projects.txt
```

Now that we have all the data to analyze, in the next step, the matcher code written in Go using the help of Abstract Syntax Tree(AST), will analyze all the systems to compute our main metrics based on the number of occurrences of traditional concurrency-related features. There are a total of 6 features that we are looking at. They are listed below:

1. Waitgroups(Declaration, Done, Add, Wait)
2. Cond (Declaration, Lock, Unlock, Wait, Signal, Broadcast, New)
3. Once (Declaration, Do)
4. Mutex (Declaration, Lock, Unlock)
5. RWMutex (Declaration, Lock, Unlock)
6. Locker (Declaration, Lock, Unlock)

The analyzer can be run on either individual files or on an entire directory. To run it on an individual file, we should use the `--filePath` command line argument. To run the analyzer on a directory containing .go files, you should use the `--dirPath` command line argument. Results of the analysis will be stored in the CSV file given using the `--output` command line argument. The analyzer develops the CSV file that contains the number of occurrences of traditional concurrency features for each file present in the system. Then we modified the spreadsheet and calculated the count for each system. From these results we analyzed the usage of features and are reported in the next Chapter.

Chapter 5

Discussion

This chapter evaluates the experimental implementation to determine the usage of concurrent features across open-source GitHub projects. It outlines the pertinent research questions relevant to a software developer, outlines the methodology by which the features are analyzed, and then discusses the results. It also describes how our work can be helpful in advanced research and while creating tools.

5.1 Methodology

The main goal of the analysis is to gain a better understanding of how the concurrent features in the Go language are spread across the openly available code repositories from GitHub. The first step is selecting the corpus as described in Chapter 4. We acquired the corpus from an existing study that worked on message-passing concurrency capabilities in Go to implement our thesis. The corpus contains 856 publicly available GitHub Go projects chosen based on the number of stars they have. Secondly, we developed a set of concurrency features to investigate. Then, to match the characteristics and determine the number of occurrences, a matcher code is created. Lastly, the results are saved in a CSV file, and the data is examined to provide insight into how different features are used. A detailed methodology can be seen in Figure 4.1 in Chapter 4.

5.2 Results

Out of the many concurrency features available in the Go language for our analysis, we have considered 6 main features. They are `Waitgroup`, `Cond`, `Once`, `Mutex`, `RWMutex`, and `Locker`. Along with these features, declarations, `Lock`, `Unlock`, `Wait`, `Do`, `Signal`, and `Broadcast` calls of these features are also counted. Some calls are categorized as `unknown`. These calls may be completely unrelated to concurrency. For instance, a function named `Do` called on a custom type, would be categorized as `unknownDo`, as would a call on a `Once` value if the analysis cannot determine a `Once` value is the target. Similarly a simple `Add` function to add two numbers, when called, is categorized as `unknownAdd`. `unknownAdd` which is the number of uncategorized calls to Add is the most occurred feature in the selected systems with an average of 2,564 occurrences per systems and a total of 248,385 occurrences in all the systems. The next ones are `unknownSignal` and `unknownDo` calls. Listing 5.1 is code taken from [48] files. Here they used `signal` function that will catch the exceptions thrown by other files. This type of call will be categorized as `unknownSignal`, since we are only searching for conditional signal usage in our work.

Listing 5.1: Code from system name "Sia"

```
mmapChan := make(chan os.Signal, 1)
   signal.Notify(mmapChan, syscall.SIGBUS)
   go func() {
      <-mmapChan
      fmt.Println("A fatal I/O exception (SIGBUS) has occurred.")
      fmt.Println("Please check your disk for errors.")
      os.Exit(1)
   }()
```

When it comes to the features we are looking at, the most used feature is mutexUnlock which is the number of calls to `Unlock` a `Mutex`. The average occurrences are 128.6 per

systems and the total occurrences are 67,295 for all the systems. It is followed by mutex lock with an average by system of 120.38 and a total across all the systems of 61,405. Mutex, or mutual exclusion, enables programmers to build code that is protected from race conditions. The sync package in Go implements Mutex, which includes functions for locking and unlocking processes.

- `Mutex.Lock()`- locks the mutex so that only one goroutine can access the data. This blocks other goroutines and they have to wait until the mutex is released.
- `Mutex.Unlock()`- unlocks the mutex and exposes it to other goroutines that would like to access the data.

Deadlock is a condition when the execution of the program becomes stopped while waiting for a resource to be released. Deadlock is a behavior that might result from failing to invoke the Unlock function. It must be made sure that the code always invokes the Unlock method to avoid Deadlocks. Locks and unlocks should always be balanced. But the count of total Unlock calls for Mutex, RWMutex, Condition and Locker features are 98.477. Whereas the Lock calls on the previously mentioned features are 91.428. Unlock calls are greater than Lock calls in every feature. This is mostly because the systems are using conditional statements for Unlock, requiring them to write the `.Unlock` call a couple of times, while `.Lock` is called a single time. This can be explained with a sample code in Listing 5.2 taken from [11] from their GitHub repository. You can see how they used `mu.Unlock` two times in the if...else statement. This can be a good reason why the unlock calls count are high. Another reason would be errors or inappropriate calls. A thread attempting to unlock an unlocked mutex will return with an error.

Listing 5.2: Code from system name "vuvuzela"

```
func (gc *GuiClient) deactivateConvo(convo *Conversation) bool {
   gc.mu.Lock()
   if gc.active[convo] {
      delete(gc.active, convo)
```

```
        gc.mu.Unlock()
    } else {
        gc.mu.Unlock()
        return false
    }
    return true
}
```

Mutex can work in combination with WaitGroup, another class included in the sync package. A WaitGroup is used to wait for multiple goroutines to finish. The total count of WaitGroup features present is 38.732 and that of Mutex is 148.668. From this we can know that only 26% files have used Mutex and WaitGroups in combination.

RWMutex is followed by Mutex lock and unlock calls in highest number of occurrences. A RWMutex is a reader/writer mutual exclusion lock. The lock can be held by an arbitrary number of readers or a single writer. The zero value for a RWMutex is an unlocked mutex. By using a sync.RWMutex, program becomes more efficient. Total RWMutex calls are 72,370 that is 8.1% of total. So around 70 systems are trying to access data by using read/write mutex.

Once is an object that will perform exactly one action. Once is declared around 9,900 times and onceDo is called around 10.355 times. The most number of times they both are used is 19 times in the same file [49]. The total `cond` feature occurrences are around 8,000 making it the second least feature used after Locker. The least used feature is Locker. A Locker is a generalization of Mutex and RWMutex, which means people are generally using those call directly. 540 occurrences are recorded which include three calls of Declarations, Unlock and Lock, with `LockerLock` (that is the number of calls to Lock on a Locker) and `LockerUnlock` (that is the number of calls to Unlock on a Locker) being least used feature calls in all the systems that are considered with an average per systems usage of 0.24 each.

Table 5.1 presents an overview on all the statistics of results.

Trait	Count
Total number of projects analyzed	856
Total number of Go language Files	500322
Total lines of code	132746554
Total Number of Occurrences of all features	883376
Total Number of Occurrences of traditional concurrency features	288559
Highest number of occurrences	Mutex Unlock 67295
Lowest number of occurrences	Locker Lock 159

Table 5.1: Statistics of Results

5.3 Research Questions

To understand more about how the concurrency features in Go language are used, we set out to answer the following research questions:

RQ1: How often do any traditional concurrency features appear in Go projects?

The corpus contains a total of 856 Go projects available on GitHub that are taken from previous work. Out of all the systems, the total number of occurrences of the concurrency features we considered is 288,559, excluding all the unknown features and their calls. 649 systems, that is 76.99% of all the systems, used at least one concurrency primitive, whereas 194 systems did not use any concurrency primitive at all.

	Feature Name	Count of Occurrences	Projects	Proportion
1	WaitGroup	38,732	486	57.65%
2	Cond	8,042	188	22.30%
3	Once	20,261	367	43.53%
4	Mutex	148,668	538	63.81%
5	RWMutex	72,370	430	51.00%
6	Locker	540	93	11.03%
7	Unknowns	594,817	194	23.00%

Table 5.2: Projects using traditional features

Table 5.2 gives a count of how many times the traditional concurrency features occurred in the Go projects. It also gives an insight into the total number of projects using the

features and what percent of these files are, out of the total files. We can observe that Mutex, RWMutex and and WaitGroup appear in more than 50% of the projects.

The top three projects that used the most concurrency primitives are *fabric, autoscaler, and rancher*. These three projects combined have 37.943 features of concurrency that accounted for 13.15% of the total features. The fabric repository alone used 14.844 features, while the other two repositories have 12.077 and 11.022 features respectively. The fabric repository has 4.391 files, while autoscaler has 24.038 files and rancher has 3,049 files, making fabric([37]) and rancher([10]) the projects that could be quickly viewed to understand the concurrency primitives. These two repositories also have the highest number of RWMutex features used accounting for a combined 20.15% of total usage of RWMutexes. Out of all the systems, autoscaler([64]) has the highest total usage of mutexes with 7,696 occurrences which is 5.17% of total Mutexes. The autoscaler repository even has the highest number of features of once and cond with 5.45% and 6.00% of total feature occurrences. The cockroach([61]) repository has the highest number of WaitGroup features(1,272) and also has the highest number of the least used feature locker accounting for 7.66% of the total lock features.

RQ2: In projects that make use of traditional concurrency features, what percent of files includes at least one declaration or use of a traditional concurrency feature?

There are a total of 500.322 Go language files in the projects that we selected. Out of these, only 31,295 files used at least one traditional concurrency feature, which is only 6.26% of the total files. The remaining 468.940 files haven't used any traditional feature. The highest number of features that gen.go file [27] has is 813. But the same file is pulled from one system five times. This can be a disadvantage of our work since the proportions of the results can be biased because of redundant code files. The gen.go file only has Mutex features present in it with 89 MutexDecls, and 362 each of MutexLock and MutexUnlock, making the file highest in the total Mutex features of all the files.

Figure 5.1 plots the % of the distribution of the feature declarations across the total

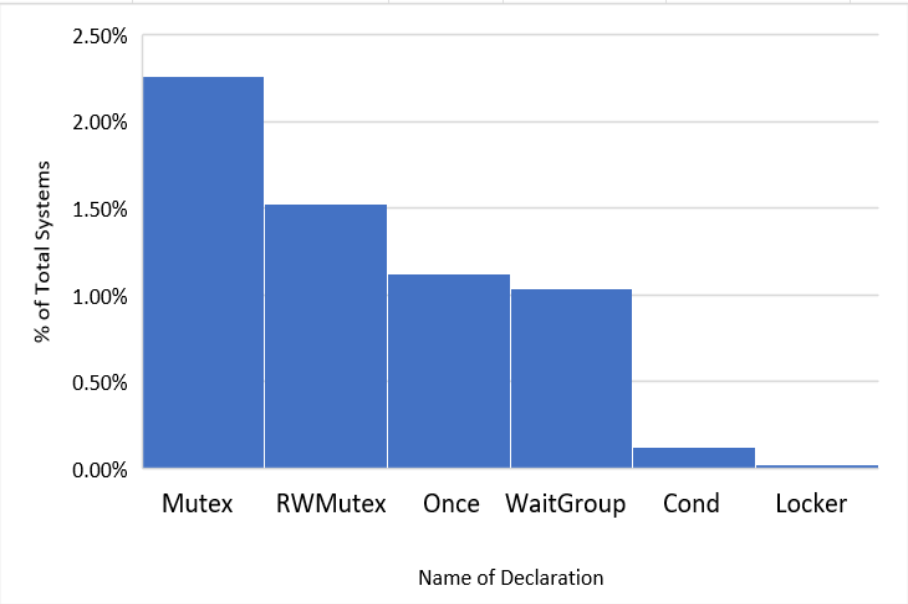

Figure 5.1: Percent of Systems Including Declaration Type

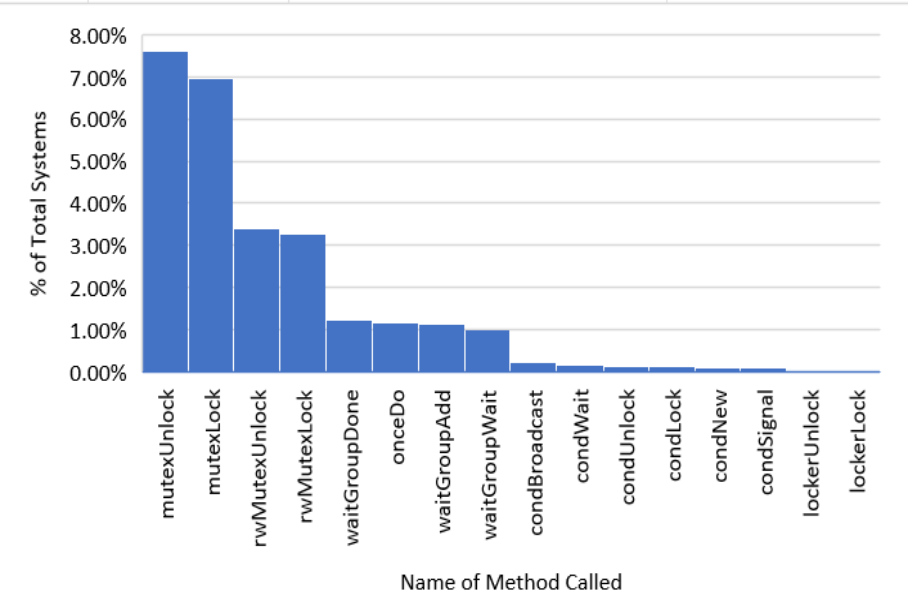

Figure 5.2: Percent of Systems Including Method Call

number of systems and Figure 5.2 plots the % of the distribution of the feature call for all the systems that are considered. They are arranged from highest to lowest total number of occurrences.

5.4 Threats to Validity

The main threat to validity is that the corpus of systems used in the evaluation may not be representative of other systems, written in Go, that were not included in the analysis. This is mitigated by the variety of systems used, which covers a number of different domains, with code written by different development teams. The same corpus was also used in a prior analysis, in that case focused on message passing concurrency [16].

Another potential threat to validity is that the analysis may both over- and under-count the occurrence of concurrency features in the analyzed code. The analysis occurs on individual files, using facts computed about that file in isolation. The linking of definitions of concurrency constructs with their uses in a file is based just on the defined name, which could be a simple name (e.g., c for a condition variable) or a field in a structure (e.g., item.Selected). Calls to methods such as verb or Do that cannot be linked to a name are categorized as `Unknown` calls of that type. Some of these calls may be calls to the concurrency methods of that name, but others could be calls to unrelated methods. In the former case, we may be under-counting. Over-counting would only occur if the same name was used in a different definition, with a type that happened to have methods with the same names as those being matched by our tool (e.g., both a Mutex and another type have a Lock method, and the same name is used in different functions, with these different types, and with calls to Lock). We believe this would be unlikely, but a stronger use-def analysis would help to reduce the likelihood of this occurring.

5.5 Potential Applications

Our work analyzes how the concurrency features are used by finding the number of occurrences of traditional concurrency primitives. This can benefit the developers in a couple of ways. The immediate benefit would be by looking at our results they can make sure that the particular concurrency primitive is still in use and what traditional feature is mostly

used and least used. These findings can be used to guide future study into static or dynamic verification of concurrent programs. Our research will enable researchers and practitioners to make well-informed decisions on the scalability (towards larger programs) and applicability (towards a larger subset of Go) of their approaches. If a developer want to develop any tool, for example, developing a tool that resolves bugs in a particular feature, they can use this work to know what feature is mostly used in Go languages and develop accordingly.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Go's approach for simplicity and encouragement for good programming practices makes it very useful for large-scale software development. Concurrency in Golang is the ability for functions to run independently of each other. The built-in concurrency system revolves around the "goroutines". They serve as lightweight threads managed by the Go runtime, and starting a large number of them is inexpensive (on a scale of thousands). Channel is a built-in type that allows safe communication and synchronization between goroutines. Apart from these, there are also traditional primitives of concurrency like WaitGroups, Mutex, Conditional variables, and many more. Goroutines and Channels are mostly used by Go language programmers, but what about traditional features? To get an overview of the question, this master's thesis is set out to find out how developers are using the traditional features of concurrency in the Go language. This is done by analyzing the total number of occurrences of each feature in Go language files.

To implement our thesis, first, we have gathered the corpus from an existing paper that worked on message-passing concurrency features in Go. The corpus is above 800 openly available GitHub projects in the Go language selected based on the number of stars they have. Then we came up with a set of concurrency features to analyze. A matcher code is developed to match the features and also know the number of occurrences. The results are stored in a CSV file and data is analyzed to get an insight into the usage of different

features. `mutexUnlock` is the most used traditional feature with a 7.62% of total instances. The least used feature is Locker, and the least used call in Locker is `Lockerlock` with just 159 occurrences which are 0.017% of the total. Other related statistics such as the mean and median of the occurrences are explained in the Discussion chapter. We believe this information lays a solid foundation for future empirical studies, as well as for the construction of tools for understanding and renovating concurrency features in the Go language.

6.2 Future Work

As part of future research, it would be interesting to extend these experiments to a wide range of features, providing additional context for the current results. One such thing would be to find if there are any patterns of the features occurrences. If one program is using a feature, say Mutex, is it using another concurrency feature, such as RWMutex or Conditions or Once statement? It would be a nice thing to analyze what features co-occur in that same file and which ones don't. Another area for future work is to be able to convert code that uses Mutexes and turn them into Channels. We can also study the possibility of integrating our work with error detection tools to analyze if there are any errors in the systems considered. Another area for future work is to expand our corpus to recently added open-source projects as well. It would also be interesting to compare the finding of the previous research with our analysis, where they analyzed message-passing concurrency primitives. It gives us better insights into how the primitives they analyzed and the primitives our work analyzed are similar or different. Lastly, another interesting comparison would be between the findings of this work and with the use of similar features in other programming languages, for example, comparing the use of mutexes in Go with the use of mutexes in the pthreads library in C or C++

BIBLIOGRAPHY

[1] Akhter, S., and Roberts, J. *Multi-core Programming: Increasing Performance Through Software Multi-threading*. Books by engineers, for engineers. Intel Press, 2006.

[2] Alomari, Z., Halimi, O., Sivaprasad, K., and Pandit, C. Comparative Studies of Six Programming Languages. *IEEE* (04 2015).

[3] Ankur Anand. Communicating sequential processes(CSP) for Go developer in a nutshell. https://levelup.gitconnected.com/communicating-sequential-processes-csp-for-go-developer-in-a-nutshell-866795eb879d. [Online; accessed 21-September-2021].

[4] Astrauskas, V., Müller, P., Poli, F., and Summers, A. J. Leveraging Rust Types for Modular Specification and Verification. *Proc. ACM Program. Lang. 3*, OOPSLA (oct 2019).

[5] Blom, S., and Huisman, M. The VerCors Tool for Verification of Concurrent Programs. In *FM 2014: Formal Methods* (Cham, 2014), C. Jones, P. Pihlajasaari, and J. Sun, Eds., Springer International Publishing, pp. 127–131.

[6] CASS, S. Top Programming Languages 2021 Python dominates as the de facto platform for new technologies. https://spectrum.ieee.org/top-programming-languages-2021, 2021. [Online; accessed 14-November-2022].

[7] Chabbi, M., and Ramanathan, M. K. A Study of Real-World Data Races in Golang. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (New York, NY, USA, 2022), PLDI 2022, Association for Computing Machinery, p. 474–489.

[8] Contributors, T. TIOBE programming community index. https://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2010. Accessed: 22-September-2022.

[9] Corp, I. *Occam Programming Manual*. Prentice Hall Trade, 1984.

[10] Damle, P. Rancher. https://github.com/rancher/rancher, 2022.

[11] David Lazar, N. Z. Vuvuzela. https://github.com/vuvuzela/vuvuzela/blob/master/cmd/vuvuzela-client/gui.go, 2018.

[12] Deniélou, P.-M., and Yoshida, N. Multiparty Session Types Meet Communicating Automata. In *Programming Languages and Systems* (Berlin, Heidelberg, 2012), H. Seidl, Ed., Springer Berlin Heidelberg, pp. 194–213.

[13] Deshpande, N. A., and Weiss, N. Go Language. In *Analysis of the Go runtime scheduler* (2012).

[14] Developers, G. The Go Programming Language Specification. https://go.dev/ref/spec. [Online; accessed 11-June-2022].

[15] dfawley. A high performance, open-source universal RPC framework. https://github.com/grpc/grpc-go.

[16] Dilley, N., and Lange, J. An Empirical Study of Messaging Passing Concurrency in Go Projects. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2019), pp. 377–387.

[17] Dilley, N., and Lange, J. Bounded verification of message-passing concurrency in Go using Promela and Spin. In *Proceedings of the 12th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2020, Dublin, Ireland, 26th April 2020* (2020), S. Balzer and L. Padovani, Eds., vol. 314 of *EPTCS*, pp. 34–45.

[18] Doxsey, C. Concurrency. https://www.golang-book.com/books/intro/10, 2021. [Online; accessed 25-April-2022].

[19] Ellis, S. 10 Features of Go that Set it Apart From Other Languages. https://itnext.io/10-features-of-go-that-set-it-apart-from-other-languages-89337e5ee551. [Online; accessed 10-June-2022].

[20] Fang, Z., Luo, M., Anwar, F. M., Zhuang, H., and Gupta, R. K. Go-realtime: a lightweight framework for multiprocessor real-time system in user space. *SIGBED Rev. 14*, 4 (2017), 46–52.

[21] Gabet, J., and Yoshida, N. Static Race Detection and Mutex Safety and Liveness for Go Programs (extended version), 2020.

[22] Giunti, M. GoPi: Compiling Linear and Static Channels in Go. In *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings* (2020), S. Bliudze and L. Bocchi, Eds., vol. 12134 of *Lecture Notes in Computer Science*, Springer, pp. 137–152.

[23] Go Language Contributors. Frequently Asked Questions (FAQ). "https://go.dev/doc/faq#csp", 2021. [Online; accessed 15-September-2022].

[24] Go Language Contributors. The Go Programming Language Specification. https://go.dev/ref/spec, 2021. [Online; accessed 15-September-2022].

[25] GoLanguage. The Go programming language. https://go.dev/, 2018. [Online; accessed 22-March-2022].

[26] Gupta, V. Top 7 Reasons to Learn Golang. geeksforgeeks.org/top-7-reasons-to-learn-golang/, 2020. [Online; accessed 3-July-2022].

[27] Hance, S. k8s-cloud-provider. https://github.com/GoogleCloudPlatform/k8s-cloud-provider/blob/master/pkg/cloud/gen.go, 2022.

[28] Hawthorne, S. A Language Comparison Between Parallel Programming Features of Go and C, 2020.

[29] Hinrichsen, J. K., Bengtson, J., and Krebbers, R. Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic. *ArXiv abs/2010.15030* (2022).

[30] Hoare, C. A. R. Communicating Sequential Processes. *Commun. ACM 21*, 8 (aug 1978), 666–677.

[31] Honda, K., Yoshida, N., and Carbone, M. Multiparty Asynchronous Session Types. *Journal of the ACM 63*, 1 (2016), 1.

[32] Horntvedt, R., and Åkesson, T. IEEE. In *Java, Python and Javascript, a comparison* (2018).

[33] *Tools and Algorithms for the Construction and Analysis of Systems: 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part I* (Berlin, Heidelberg, 2019), Springer-Verlag.

[34] Hundt, R. Loop Recognition in C++/Java/Go/Scala. In *Proceedings of Scala Days 2011* (2011).

[35] Hüttel, H., Lanese, I., Vasconcelos, V. T., Caires, L., Carbone, M., Deniélou, P., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H. T., and Zavattaro, G. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv. 49*, 1 (2016), 3:1–3:36.

[36] ISOcpp. The Standard. https://isocpp.org/std/the-standard, 2020. [Online; accessed 10-July-2022].

[37] James Taylor, M. S. Hyperledger Fabric. https://github.com/hyperledger/fabric, 2022.

[38] K P, N. R., .Y, G., .D, S., and Rajesh, M. Comparison of Programming Languages: Review. *IEEE 9* (07 2018), 113–122.

[39] Lange, F. Official Go implementation of the Ethereum protocol. . https://github.com/ethereum/go-ethereum/releases.

[40] Lange, J., Ng, N., Toninho, B., and Yoshida, N. Fencing off go: Liveness and safety for channel-based programming. *ACM SIGPLAN Notices 52*, 1 (2017), 748–761.

[41] Lange, J., Ng, N., Toninho, B., and Yoshida, N. A Static Verification Framework for Message Passing in Go Using Behavioural Types. In *Proceedings of the 40th International Conference on Software Engineering* (New York, NY, USA, 2018), ICSE '18, Association for Computing Machinery, p. 1137–1148.

[42] Lange, J., Tuosto, E., and Yoshida, N. From communicating machines to graphical choreographies. In *42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2015), ACM, pp. 221–232.

[43] Levin, R. *Biography*. Association for Computing Machinery, New York, NY, USA, 2019, p. 131–170.

[44] Liu, X., and Gontey, P. Program Translation by Manipulating Abstract Syntax Trees. In *Proceedings of the C++ Workshop. Santa Fe, NM, USA, November 1987* (1987), USENIX Association, pp. 345–360.

[45] Liu, Z., Zhu, S., Qin, B., Chen, H., and Song, L. Automatically Detecting and Fixing Concurrency Bugs in Go Software Systems. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2021), ASPLOS '21, Association for Computing Machinery, p. 616–629.

[46] LLC, M. The Go Programming Language, Frequently Asked Questions(FAQ). https://go.dev/doc/faq, 2017. [Online; accessed 30-March-2022].

[47] Lu, D., Wu, J., Sheng, Y., Liu, P., and Yang, M. Analysis of the Popularity of Programming Languages in Open Source Software Communities. In *2020 International Conference on Big Data and Social Sciences (ICBDSS)* (2020), pp. 111–114.

[48] Luke Champine, D. V. Capricorn. https://github.com/NebulousLabs/Sia/blob/master/cmd/siad/daemon.go, 2018.

[49] Martin Holst Swende, G. B. Ethereum. https://github.com/ethereum/go-ethereum/blob/master/eth/protocols/snap/sync_test.go, 2022.

[50] McGranaghan, M., and Bendersky, E. Go by Example: Channels. https://gobyexample.com/channels. [Online; accessed 13-June-2022].

[51] McGranaghan, M., and Bendersky, E. Go by Example: WaitGroups. https://gobyexample.com/waitgroups, 2021. [Online; accessed 10-September-2022].

[52] Midtgaard, J., Nielson, F., and Nielson, H. R. Process-Local Static Analysis of Synchronous Processes. In *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings* (2018), A. Podelski, Ed., vol. 11002 of *Lecture Notes in Computer Science*, Springer, pp. 284–305.

[53] Milner, R. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[54] Milner, R. Bigraphical Reactive Systems. In *CONCUR 2001 - Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20-25, 2001, Proceedings* (2001), K. G. Larsen and M. Nielsen, Eds., vol. 2154 of *Lecture Notes in Computer Science*, Springer, pp. 16–35.

[55] Milner, R., Parrow, J., and Walker, D. A Calculus of Mobile Processes, I. *Inf. Comput. 100*, 1 (1992), 1–40.

[56] Ng, N., and Yoshida, N. Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis. In *25th International Conference on Compiler Construction* (2016), ACM, pp. 174–184.

[57] Paul, J. Is Golang worth learning in 2022? Why should you learn Go Programming Language? https://medium.com/javarevisited/what-is-go-or-golang-programming-language-why-learn-go-in-2020-1cbf0afc71db, 2022. [Online; accessed 3-July-2022].

[58] Pike, R. The go programming language. *Talk given at Google's Tech Talks 14* (2009).

[59] Pike, R. Go at Google. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity* (New York, NY, USA, 2012), SPLASH '12, Association for Computing Machinery, p. 5–6.

[60] Prasertsang, A., and Pradubsuwun, D. Formal verification of concurrency in Go. In *2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)* (2016), pp. 1–4.

[61] Qazi, F. CockroachDB. https://github.com/cockroachdb/cockroach, 2022.

[62] Reppy, J. H. CML: A Higher Concurrent Language. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1991), PLDI '91, Association for Computing Machinery, p. 293–305.

[63] Robert, O., Dan, H., and Jeri, E. *The Essential client/server survival guide*. New York : Wiley Computer Publishing, 1996.

[64] Robot, K. P. Kubernetes Autoscaler. https://github.com/kubernetes/autoscaler, 2022.

[65] Rybacka, K. The Go programming language—everything you should know. https://codilime.com/blog/go-programming-language-everything-you-should-know/, 2021. [Online; accessed 10-July-2022].

[66] Saini, A. Select Statement in Go Language. https://www.geeksforgeeks.org/select-statement-in-go-language/?ref=lbp, 2019. [Online; accessed 25-September-2022].

[67] Sharma, A. A Mini-Guide on Go Programming Language. https://appinventiv.com/blog/mini-guide-to-go-programming-language/. [Online; accessed 10-June-2022].

[68] Stadtmüller, K., Sulzmann, M., and Thiemann, P. Static Trace-Based Deadlock Analysis for Synchronous Mini-Go, 2016.

[69] Stenman, E. book. In *Efficient Implementation of Concurrent Programming Languages* (2002).

[70] Stinson, D. L. Deep Learning with Go, 2020.

[71] Sulzmann, M., and Stadtmüller, K. Trace-Based Run-Time Analysis of Message-Passing Go Programs. In *Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings* (2017), O. Strichman and R. Tzoref-Brill, Eds., vol. 10629 of *Lecture Notes in Computer Science*, Springer, pp. 83–98.

[72] Sulzmann, M., and Stadtmüller, K. Two-Phase Dynamic Analysis of Message-Passing Go Programs Based on Vector Clocks. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018* (2018), D. Sabel and P. Thiemann, Eds., ACM, pp. 22:1–22:13.

[73] Suresh, D. Introduction to the Go Programming Language. *J. Comput. Sci. Coll. 29*, 2 (dec 2013), 113–114.

[74] Tang, P. Multi-Core Parallel Programming in Go, 01 2010.

[75] Togashi, N., and Klyuev, V. Concurrency in Go and Java: Performance analysis. In *2014 4th IEEE International Conference on Information Science and Technology* (2014), pp. 213–216.

[76] Togashi, N., and Klyuev, V. Concurrency in Go and Java: Performance analysis. In *2014 4th IEEE International Conference on Information Science and Technology* (2014), pp. 213–216.

[77] Tu, T., Liu, X., Song, L., and Zhang, Y. Understanding Real-World Concurrency Bugs in Go. In *Go* (New York, NY, USA, 2019), ASPLOS '19, Association for Computing Machinery, p. 865–878.

[78] Villard, J., Lozes, É., and Calcagno, C. Proving Copyless Message Passing. In *Programming Languages and Systems* (Berlin, Heidelberg, 2009), Z. Hu, Ed., Springer Berlin Heidelberg, pp. 194–209.

[79] Virding, R., Wikström, C., Williams, M., and Armstrong, J. *Concurrent Programming in ERLANG (2nd Ed.)*. Prentice Hall International (UK) Ltd., GBR, 1996.

[80] Wang, Y., and Li, H. Code Completion by Modeling Flattened Abstract Syntax Trees as Graphs. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021* (2021), AAAI Press, pp. 14015–14023.

[81] Wayner, P. Bossie Awards 2010: The best open source software of the year. https://www.infoworld.com/article/2625971/bossie-awards-2010--the-best-open-source-software-of-the-year.html. Accessed: 22-September-2022.

[82] Wikipedia contributors. Concurrency (computer science) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Concurrency_(computer_science)&oldid=1089344057, 2022. [Online; accessed 5-July-2022].

[83] Wikipedia contributors. Message passing — Wikipedia, The Free Encyclopedia, 2022. [Online; accessed 20-July-2022].

[84] Wolf, F. A., Arquint, L., Clochard, M., Oortwijn, W., Pereira, J. C., and Müller, P. Gobra: Modular Specification and Verification of Go Programs. In *Computer Aided Verification* (Cham, 2021), A. Silva and K. R. M. Leino, Eds., Springer International Publishing, pp. 367–379.

[85] Wolf, F. A., Arquint, L., Clochard, M., Oortwijn, W., Pereira, J. C., and Müller, P. Gobra: Modular Specification and Verification of Go Programs (extended version), 2021.

[86] yuroyoro. GoAst Viewer. http://goast.yuroyoro.net/, 2019. [Online; accessed 10-May-2022].

[87] Ziobrowski, A. Introduction to Abstract Syntax Trees in Go. https://tech.ingrid.com/introduction-ast-golang/, 2021. [Online; accessed 30-September-2022].