BOUNDED VERIFICATION OF MESSAGE-PASSING CONCURRENCY IN GO

Thesis submitted
for the degree
of doctor of philosophy
at The University of Kent

By
Nicolas Dilley
School of Computing
University of Kent
April 2022

Abstract

Go is a programming language that has gained increased popularity due to its good support for system programming and its channel-based message passing concurrency mechanism. These features rendered Go the language of choice of many platform software developers. Go offers a wide range of primitives to coordinate lightweight threads, e.g., channels, waitgroups, and mutexes. Although, these concurrency primitives help mitigate data races, they introduce additional complications due to the complexity of reasoning about concurrency.

In this thesis, we first perform an empirical analysis on concurrent Go programs which analyses 125 Go projects from GitHub in order to understand how concurrency is used in publicly available code. Our results include the following findings: (1) concurrency primitives are used frequently and intensively, (2) most projects use synchronous communication channels over asynchronous ones, and (3) most Go projects use simple concurrent thread topologies, which are however currently not fully supported by existing static verification frameworks. To address these limitations, we propose a novel static checker for Go programs that relies on performing bounded model checking of their concurrent behaviours. In contrast to previous works, our approach deals with large codebases, supports programs that have statically unknown parameters and is extensible to additional concurrency primitives.

Our work includes an empirical analysis that studies the usage of concurrency

in Go projects, a detailed presentation of the extraction algorithm from Go programs to Promela models, an algorithm to automatically check programs with statically unknown parameters, and a large scale evaluation of our approach. The latter shows that our approach outperforms the state-of-the-art.

Acknowledgements

I dedicate this thesis to my father who always told me that; *when you start something, you should see it all the way through.*

But of course, starting something is hard enough, seeing it through requires help! I am deeply indebted to those who have helped me and want to offer a few words of specific thanks here:

First and foremost, I would like to give my deepest thanks to my supervisor, Julien Lange, without whom this thesis would have never been possible nor achievable. From the start, even though I doubted my abilities, you have helped me go far beyond what I think I could achieve. You have really been the best supervisor I could ask for!

Second, I want to thank David Perez-Castro, Mahammad Mousavi, Nicholas Ng, Stefan Marr, Bernardo Toninho, Laura Bocchi, Andy King and Alex Freitas for their comments on earlier version of this work: Your insights, wisdom and comments have been invaluable.

Third, this thesis would not have been possible without the support and love of my family and my precious friends, Jessica, Thaïs, Sebastien, James and Marco: You all have always believed in me and helped make these three years as enjoyable and as productive as possible.

Finally, I want to thank the University of Kent and my colleagues at the PLAS group: who have been of great support and always there when I needed them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, there has been growth in the popularity of programming languages that natively support higher-level concurrency abilities, such as Go or Rust. These concurrency abilities allow developers to divide a program into multiple independents parts, which can execute in parallel when multiple cores are available. However, developing concurrent software is particularly difficult because bugs are often non-trivial to detect since they do not occur in every execution (e.g., due to non-determinism or because they depend on the program's arguments) and some may not be easily observable (e.g., they might only affect the memory footprint of the program). The additional complexity that characterises concurrent software can lead to unexpected bugs which can, in turn, lead to disastrous consequences. Consider the example reported in Clearfield and Lofchie (2013), in which a technical failure, due to race conditions, affected the prices of the shares of Facebook on the NASDAQ exchange. This failure was caused by the unexpected huge enthusiasm of the initial public offering (IPO) of Facebook (80 million shares in 30 seconds according to Pepitone (2012)). As a result of this technical failure, NASDAQ was fined 10 million dollars.

To avoid such costly consequences, similarly to structural engineers and architects who rely on a diverse range of specialised tools to verify the integrity of their

constructions, we want to help developers by implementing software verification techniques and tools to prove the correctness of their programs before using them in production. Developing such tools would reassure developers that executing their concurrent software will not generate certain bugs, such as deadlocks. However, each additional thread in a program can drastically increase the number of possible interleaving of executions of all threads in the program, which as a result increases the complexity of implementing verification approaches that *scale* well and provide *appropriate feedback* on large distributed software.

To overcome these challenges, one of the key ideas is to find an appropriate *abstraction* that reduces the complexity of the program by keeping only the constructs that are relevant to the properties that need to be verified. The main challenge in choosing an appropriate abstraction is to strike a good balance between reducing the complexity of the program while keeping the abstraction precise enough to diminish the rate of false alarms/positives. Over the years, many techniques have been used to abstract programs such as types and separation logic.

The abstraction technique used in this thesis is a modelling technique called *behavioural types*. Behavioural types describe the dynamic behaviour of a program such as the interactions and creation of threads in a program. Various modelling approaches, called process calculi, have been developed to reason about the behaviours of programs such as CCS (Calculus of Communicating Systems by Milner (1989)) or CSP (Communicating Sequential Processes by Hoare (1978)). These modelling languages use channels and processes to model the interactions of various participants in an interactive system. Concurrency theory and type disciplines such as session types have used these languages to reason about various safety and liveness properties of distributed systems. According to one of Go designers, Pike (2012), the channel-based concurrency model of the programming language Go has been greatly inspired by such languages (notably CSP). This

makes Go a great candidate to apply theoretical approaches based on process calculi to real-world programs.

Go is an open-source programming language that was initiated by Google in 2009. It is renowned for its good support for system programming and its channel-based concurrency mechanism. It is advertised by Golang (2018) as "*an open source programming language that makes it easy to build simple, reliable, and efficient software*". These strengths have made it the language of choice for many platform software such as Docker and Kubernetes, which in turn are the most common software for containerisation management. With the growing popularity of containerisation technology in today's software industry Go has become a key element of many modern software.

The native inter-thread synchronisation mechanisms in Go differ from more traditional synchronisation mechanisms over shared memory by promoting the motto "*don't communicate by sharing memory, share memory by communicating*" (Pike (2015)), and encouraging communication via channels. This emphasis on channel-based communication helps to develop concurrent programs which are conceptually simpler and better suited to be automatically verified to guarantee the absence of communication bugs such as deadlock and thread starvation.

However, a recent survey of popular Go projects by Tu et al. (2019) has showed that message-passing-based software is as liable to bugs as other concurrent programming techniques such as lock-based techniques. They also showed that Go concurrency-related bugs are hard to detect and have a long lifetime. This is reflected in a recent survey amongst Go programmers (team (2017)) reporting that programmers often do not feel they are able to effectively repair bugs related to Go's concurrency features. It is thus essential to develop reliable *static* checkers that can rule out these bugs.

Concretely, message-passing concurrency bugs in Go fall into two categories:

(*i*) blocking bugs, where a goroutine is permanently waiting for a matching send/receive action and (*ii*) channel bugs, where a goroutine attempts to close or send to a channel that is already closed. However, beyond a rather standard type system and a runtime global deadlock detector, the Go language and its associated tooling do not offer any means to detect concurrency bugs.

To study how Go developers use concurrency in their programs, we have performed an empirical analysis that analyses 125 of the most popular Go Github projects. This allowed us to determine which concurrency features and patterns not supported by previous work (notably Lange et al. (2018); Gabet and Yoshida (2020); Liu et al. (2021)) were most commonly used in real-world applications.

Based on the result of the analysis, we implemented a practical tool based on a behavioural types approach first formalised in Lange et al. (2017) which expands the subset of Go supported and reduces the number of false alarms by increasing the preciseness of the model generated. This is achieved by adding supports for *communication parameters*, which are variables in the programs that affect the concurrent topology of the programs. In addition, to increase the scalability of previous approaches inspired by behavioural types, we partition the program into smaller concurrency independent programs and verify them separately.

To evaluate the applicability and the false alarm rate of our approach on real-world programs, we have devised a set of benchmarks aimed at reproducing current concurrency patterns used in real-world projects. We evaluate our approach on this set of benchmarks and compare the results with two of the more mature recent static checkers, GCatch and Godel2.

Overall, we have found that our approach finds the most bugs while returning very few false alarms compared to other tools. In addition, we have performed a large-scale evaluation of our approach on the same set of 125 Github projects used for the empirical analysis to test the scalability of our approach as well as manually checking the validity of the reported bugs.

1.1 Contributions

The main contributions of this thesis are :

- **An empirical analysis on the usage of concurrency in Go projects** structured around four research questions stemming from the point of view of the static verification of concurrent Go programs. The analysis is based on our paper (Dilley and Lange (2019)) and has been extended, in this thesis, to analyse additional concurrency primitives (mutexes and waitgroups) and has been used to survey 125 of the most popular Go projects. The tool used to perform our analysis is available at Dilley and Lange (2022b)

- **A novel static verification technique of concurrent Go programs**, based on our paper (Dilley and Lange (2020)) and further updated in our following paper (Dilley and Lange (2021a)) (Section III to V), which builds on the behavioural types approach first formalised in Lange et al. (2018), where models, that represent the concurrent behaviours (behavioural types), are extracted from Go programs. These models over-approximate their programs and can be verified using an off-the-shelf model checker. The implementation of our approach is available at Dilley and Lange (2022c)

- **The selection of two sets of buggy benchmarks**, which contain Go features that are known to cause problems to verification tools as well as benchmarks inspired by real-world programs. These two set of benchmarks are used to survey the functionalities supported by static and dynamic checkers and their applicability to real-world programs respectively.

- **A technique to automatically synthesise concurrent programs and automatically apply the resulting benchmarks on static checkers.** This technique was used to produce a set of 220 benchmarks and evaluate

our tool, GOMELA, against GCatch and Godel2 in Section 5.1. The tool is freely available at Dilley and Lange (2022a).

- **The evaluation of our approach on 125 Github projects** to determine the scalability of our approach on Go projects. The number of lines of these projects varies from 126 to 30 million lines of code.

1.2 Publications

This section lists, in chronological order, the three papers I published during my PhD thesis.

- ***Nicolas Dilley*** and Julien Lange. An Empirical Study of Messaging Passing Concurrency in Go Projects. Published at the 26th IEEE international conference on Software ANalysis, Evolution and Reengineering (SANER 2019). This paper describes the implementation and the results of an analysis of the usage of message-passing concurrency performed on 865 Go projects. This analysis has been extended to support additional concurrency primitives and used to answer the research questions introduced in Chapter 3.

- ***Nicolas Dilley*** and Julien Lange. Bounded verification of message-passing concurrency in Go using Promela and Spin. Published at the 12th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 2020). This paper presents the core ideas of our static approach which consists of extracting *parameterised* behavioural type models from Go programs. These models are then verified, using the model checker SPIN, for the absence of deadlocks and various channel safety errors. The implementation of our verification approach, described in Chapter 4, is based on the concepts first explored in

this paper.

- ***Nicolas Dilley*** and Julien Lange. Automated Verification of Go Programs via Bounded Model Checking. Published at the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE 2021). This paper extends the verification approach of our previous paper by supporting additional features of the Go language as well as adding support for two popular concurrency primitives. In addition, we evaluated and compared our approach against two other recent state-of-the-art static checkers as well as evaluating our approach on 99 Github projects. This paper forms the basis of Chapter 4 as well as the evaluation performed in Section 5.2. This paper received an *ACM SIGSOFT Distinguished Paper Award* as well as two badges, *Available* and *Reusable*, for the accompanying tool Gomela, developed to evaluate the verification approach on several buggy benchmarks and Github projects.

1.3 Thesis outline

The chapters of this thesis are organised as follows:

Chapter 2. Background. In this chapter, we provide a description of the Go language as well as a review of the literature enumerating previous work in the verification of Go programs.

Chapter 3. Empirical analysis of concurrency in Go projects. In this chapter, we present the analysis of concurrent Go programs that we performed on 125 Github projects. We describe our methodology, including our data selection and our Go program analyser, `GoSurvey`. We present the results of our analysis by answering four research questions. The first two questions discuss how often and

how intensively concurrency primitives are used in those projects respectively. The third question discusses how often asynchronous channels are used in Go programs. The last question explores how often complex concurrency patterns, known to cause problems to static checkers, are used in Go projects. In addition, we also discuss the limitations of our analysis, related work and give concluding remarks.

Chapter 4. Verifying Concurrent Go Programs. In this chapter, we present a novel static verification approach which verifies the absence of deadlocks and concurrency-related safety errors in concurrent Go programs. We give an overview of the modelling language Promela and the model checker SPIN. We then describe how we extract parameterised models from Go programs and the technique used to valuate and verify each resulting model. We then discuss the implementation of our approach into a tool called Gomela as well as explaining how we modeled specific Go constructs such as structures or anonymous functions.

Chapter 5. Empirical Evaluation of Gomela. In this chapter, we evaluate the applicability and scalability of our tool, Gomela. We first assess the concurrent functionalities supported by Gomela against two other recent static checkers, GCatch and Godel2. Secondly, we evaluate the applicability of Gomela on a set of benchmarks which are drawn from real-world Go programs and compare the results against GCatch and Godel2. Finally, we evaluate the scalability of Gomela by verifying 125 popular Go projects.

Chapter 6. Conclusions and Future Directions. In this chapter, we discuss the concluding remarks of this thesis as well as suggesting future works.

Chapter 2

Background

In this chapter, we introduce Go as well as giving a broad overview of the state of research on the verification of concurrent programs as a whole and on the most recent approaches targeted at Go programs.

2.1 Preliminaries

In this section, we review key aspects of the Go programming language.

2.1.1 Go programs and their properties

A Go program consists of a list of declarations of functions, structures (`struct`, on which one can define methods), and interfaces (i.e., sets of method signatures that can be implemented by structures). The special function `main()` is the entry point of the program. Go is known for its distinctive support for concurrent programming, advocating for message-passing instead of shared memory. Go natively supports channels (`chan`) over which lightweight threads (a.k.a. goroutines) coordinate their tasks by sending and receiving messages. The standard library also offers two popular concurrency primitives: waitgroups and mutexes.

	$f(\overline{a})$	Call f with arguments $\overline{a}$
	`go` $f(\overline{a})$	Spawn f with arguments $\overline{a}$
	`if` e `then` $\overline{s_1}$ `else` $\overline{s_2}$	Conditional
	`for i` $:= e_1; e_2; r\ \{\overline{s}\}$	For loop
	`for i,x` $:=$ `range` $l\ \{\overline{s}\}$	Iteration over collection l
	$ch :=$ `make(chan T,` e`)`	Declare a chan. with capacity e
○●	$ch \leftarrow\ e$	Send e over ch
○	$\leftarrow ch$	Receive on channel ch
●	`close(`ch`)`	Close channel ch
○	`for x` $:=$ `range` $ch\ \{\overline{s}\}$	Iteration over channel ch
○	`select{case` $\alpha_i : \overline{s_i}\}_{i \in I}$	Guarded choice
	`var` wg `sync.WaitGroup`	Declare a waitgroup wg
●	wg`.Add(`e`)`	Add e to wg
●	wg`.Done()`	Decrement wg by 1
○	wg`.Wait()`	Wait until wg reaches 0
	`var` mu `sync.Mutex`	Declare a Mutex mu
	`var` mu `sync.RWMutex`	Declare a RWmutex mu
○	mu`.Lock()`	Lock mutex/RWmutex
●	mu`.Unlock()`	Lock mutex/RWmutex
○	mu`.RLock()`	Lock for read access
●	mu`.RUnlock()`	Unlock read access

Figure 1: Key statements in Go, some may be blocking (○) and/or may trigger a run-time error (●).

We refer to *concurrency operations* as communication interactions (send, receive, etc) which are operated on concurrency primitives (channels, waitgroups, etc).

Figure 1 gives an overview of the control-flow constructs and concurrency operations of interests in this work. The first part of the figure lists control-flow constructs. Call $f(\overline{a})$ is a blocking call to function f, while `go` $f(\overline{a})$ spawns function f in a concurrent thread of execution. By convention, we write $\overline{a}$ for a (possibly empty) sequence $a_1, \ldots, a_k$ (with $k \geq 0$). Conditionals (if-then-else) and traditional iterations (`for` loops) are standard. Note that `while` loops do not exist in Go. Go additionally provides constructs to `range` over collections or channels. A `for i,x` $:=$ `range` $l\ \{\overline{s}\}$ block executes $\overline{s}$ for each `x` in l (`i` is bound to the index of `x` in the collection). Instruction $ch :=$ `make(chan T,` e`)` creates a new channel ch of capacity e (an integer expression). A channel carries a single type (`T`) of

```
1 msgs := make(chan int, 2)
2 msgs ← 1
3 msgs ← 3
4 close(msgs)
5 for m := range msgs {
6    fmt.Println(m}
7 )
```

Figure 2: Example of a `range` over a channel that receives all values on channel `msgs` until it is empty *and* closed.

messages.

If e evaluates to 0, the channel is synchronous (both send and receive actions are blocking), otherwise, it is asynchronous (send actions are non-blocking as long as the channel has not reached full capacity). Instruction $ch \leftarrow e$ sends e on ch, while $\leftarrow ch$ receives from ch. Invoking **`close`**(ch) closes ch. Receiving on a closed channel is non-blocking, but sending on, or closing, a closed channel triggers a runtime error.

A **`for x`** $:=$ **`range`** ch $\{\overline{s}\}$ block executes $\overline{s}$ for every message received from ch, and exits when ch is closed and empty. Figure 2 shows a program using such a construct.

This program creates an asynchronous channel that can hold up to 2 messages, two messages are enqueued, then the channel is closed. In this case, the body of the `for`-loop will execute twice as two messages were sent on channel `msgs` before it was closed.

A `select` statement allows a goroutine to wait for several operations (e.g., send/receive on a channel). It blocks until one of its cases succeeds, then executes the corresponding branch. Figure 3 shows an example of a `select` with multiple branches. This block can either synchronise with a send action on `x`, synchronise with a receive action on `y`, or, if none of these actions are available, it can take the `default` branch which is executed if all other branches are blocked.

```
1 select {
2   case ←x:
3     fmt.Println("received")
4   case y ← 42:
5     fmt.Println("sent")
6   default:
7     fmt.Println("default")
8 }
```

Figure 3: Example of a `select` statement which waits for a receive on `x` or a send on `y`. The default branch is taken if none of the other communication actions are available.

2.1.2 Waitgroup

A waitgroup is a data-structure that is used to force a goroutine to wait for a number of tasks to be performed before carrying on executing. A waitgroup is declared and initialised using the instruction `var` *wg* `sync.WaitGroup` which creates a new waitgroup *wg*. Operation *wg*.`Add`(*e*) adds *e* (which evaluates to a positive or negative integer) to the waitgroup's counter, while *wg*.`Done`() decrements the counter by 1. Operation *wg*.`Wait`() blocks until *wg*'s counter reaches 0.

2.1.3 Mutex

Go's standard library provides `Mutex` and `RWMutex`. The former is used to protect a critical section with exclusive access, i.e., both writers and readers use *mu*.`Lock`() and *mu*.`Unlock`(). The latter allows several readers to access a critical section (but at most one writer). Readers use *mu*.`RLock`() and *mu*.`RUnlock`(). Both primitives trigger a runtime error when invoking *mu*.`Unlock`() or *mu*.`RUnlock`() on an unlocked mutex.

Waitgroup and mutexes in action. Figure 4 shows a program that uses a waitgroup and a mutex to calculate the sum of the square of all numbers from 1 to 100. The program adds 1 to the counter of waitgroup `wg` at line 7 and creates an anonymous goroutine that increments `sum` by the square of `i` for each iteration of

```
1  func main() {
2    var wg sync.WaitGroup
3    var mu sync.Mutex
4    sum := 0
5
6    for i := 1; i <= 100; i++ {
7      wg.Add(1)
8      go func(i int) {
9        mu.Lock()
10       sum += i * i
11       mu.Unlock()
12       wg.Done()
13     }(i)
14   }
15
16   wg.Wait()
17   fmt.Println("sum of square is ", sum)
18 }
```

Figure 4: Example of a program that uses a waitgroup to wait for all `files` to be parsed concurrently and prints `"Done"`.

the `for`-loop. The writing to `sum` at line 10 is protected by locking and unlocking the mutexes before and after the critical section. Each goroutine decrements the counter of the waitgroup by one at line 12. After spawning 100 goroutines, the `main` goroutine waits for the counter of the waitgroup to reach 0 and prints the `sum`.

2.1.4 Concurrency bugs in Go

We distinguish between *blocking* bugs and *safety* bugs. Blocking bugs occur when a goroutine is permanently stuck waiting for a blocking operation to succeed, e.g., a receive waiting for a message to be sent. Potentially blocking operations are marked with ∘ in Figure 1. Blocking bugs are often referred to as *goroutine leaks* in the Go community. Blocked goroutines may notably cause the whole program to get stuck (global deadlock) or lead to memory leaks, as they cannot be garbage collected, see Example 2.

Safety bugs occur when an operation is invoked on a concurrency primitive unexpectedly (and triggers a run-time error), e.g., sending on a closed channel

```
1  func main() {
2    var wg sync.WaitGroup
3    someList := []int{1, 2, 3}
4    for range someList {
5      go func() {
6        wg.Done() // may trigger a run-time error
7      }()
8      wg.Add(1)
9    }
10   wg.Wait()
11 }
```

Figure 5: Negative counter bug, adapted from Kubernetes (2021a).

causes a run-time error. Operations that may trigger a run-time error are marked with • in Figure 1. Observe that all three concurrency primitives we consider can trigger such errors.

Example 1. Figure 5, adapted from Kubernetes (2021a), shows a typical safety bug. This program spawns several worker goroutines. Each goroutine invokes `wg.Done()` once they have completed their job. However, the parent thread invokes `wg.Add(1)` *after* spawning each goroutine. In an execution where, e.g., the first worker goroutine finishes its job quickly, it may decrement *wg* before it is incremented, thus triggering a run-time error ("`panic: sync: negative WaitGroup counter`").

2.2 Verification of concurrent programs

The reliance on concurrency in programs has been a key ingredient in recent years to increase the speed of execution of programs. This interest has been further amplified by the fact that Moore's Law, which states that the speed of CPU will double each year, has been negated in recent years due to the fact that the maximum number of transistors on a single chip has been reached. This means that we cannot as effectively increase the speed of single cores on a computer. Hence, programmers are relying on software techniques that tries to make the

```
1  func main() {
2    x := make(chan int, 1)
3    y := make(chan int, 1)
4
5    go func() {
6      y ← 0
7      x ← 0
8
9      ←y
10     ←x
11   }()
12
13   x ← 0
14   y ← 0
15
16   ←y
17   ←x
18
19 }
```

Figure 6: Example of a program, adapted from Sulzmann and Stadtmüller (2018) which contains a bug only when the threads of the program execute in a particular scheduling order.

best use of multiple cores to increase the runtime speed of their programs. To achieve this, programmers break their programs into smaller independent programs that communicate with each other by sharing memory, this technique is called concurrency.

Although concurrency has many benefits, such as facilitates better performance and utilisation of resources, concurrent programs can be very hard to reason about and to debug due to the many interleaving of executions of threads. Figure 6 shows a program that contains a blocking bug only when a specific ordering of thread execution is scheduled. If the goroutine spawned at line 5 executes both sends and both receives in order, the program will not block. Note that both channels are instantiated with a capacity of 1. However, if the program executes the send at line 6 followed by the send at line 13, both goroutine will be stuck at line 7 and 14.

```
1  func main() {
2
3    a := 0
4    for i := 0; i < 10; i++ {
5      go func() {
6        a = a + 1
7        fmt.Println(a) // shared access to a
8
9      }()
10   }
11
12   time.Sleep(1 * time.Second)
13 }
```

Figure 7: Example where a variable (a) is read and written by multiple threads concurrently which may lead to a data race.

2.2.1 Data races

A major problem that occurs when introducing concurrency in a program is the presence of *data races* also known as race conditions. A data race can occur when multiple threads access the same memory location and at least one of the threads writes to it. Figure 7 shows a simple case of a data race. At line 5, 10 goroutines, are spawned. Each of these threads writes the value of a at line 6 and then prints a to the console. The expected result is that each number from 0 to 9 is printed once in any order. However, when this program is executed this is rarely the case. Some values are skipped while some are printed multiple times.

To mitigate the problem caused by data races, several techniques have been developed. Three of the most notable techniques are mutexes, message-passing concurrency and synchronisation barriers.

Mutexes. A mutex is a data-structure that is used to restrict access to a critical part of a program to only a single thread at a time. This is achieved by requesting a lock to enter the critical section and releasing the lock after the critical section has been executed. When a thread requests a lock, it is blocked until the lock is available. In Go and other languages like C++ and C#, this is implemented via a data-structure which implements two methods, *Lock()* and *Unlock()*, which

surrounds the critical section and guarantees that the critical section is executed by only a single thread at a time. In Java, for example, mutexes are implemented using the keyword *synchronized* before the type of the declaration of a method or a block of code. This specifies that a synchronised method call or block of code can only be executed by a single thread at a time.

Synchronisation barriers. A synchronisation barrier is a data-structure that allows a thread to wait for one or multiple threads to terminate before carrying on executing itself. Synchronisation barriers have been natively implemented in many languages such as Rust (via sync::Barrier), Java (via util.concurrent.CountDownLatch and util.concurrent.CyclicBarrier) or python (via threading.Barrier). For languages that did not implement barriers natively, libraries such as the OpenMP library (Dagum and Menon (1998)) for C/C++ and FORTRAN contains the implementation of synchronisation barriers.

Message-passing concurrency. Instead of sharing variables between multiple threads, message-passing concurrency is a concurrent model where threads communicate and synchronise by sending messages to each other. Three popular approaches are the Actor model, channel-based message passing and MPI. In the actor model introduced by Hewitt, Bishop and Steiger (1973), each thread (called actors) holds an infinite buffer (called a mailbox) which contains all the messages that the actor has received. If the mailbox is empty, the actors wait until they receive a message and respond accordingly. This concurrency model has been supported natively by languages such as Erlang or Elixir, or using libraries such as AKKA for Scala and Java, Akka.NET for C# and F#, or CAF for C++.

In channel-based message-passing threads synchronise with each other via sending messages over channels. The concept originates from CSP (Communicating Sequential Processes) which is a formal model for concurrency introduced

by Hoare (1978). A CSP program is composed of a list of named processes instantiated at the start of the execution of the program. In CSP, these processes exchange messages synchronously by receiving and sending messages from and to other named processes. This approach has been implemented natively in Go and in Rust. In Go, messages are exchanged via synchronous (unbuffered) or asynchronous (buffered) channels. Whereas in Rust, in addition to asynchronous channels, the capacity of channels may be unbounded, hence supporting true asynchrony.

The message-passing interface (MPI) is a standard library for message-passing introduced by Forum (1993) to write parallel programs running on separate machines over large networks. MPI is a library that supports point-to-point (one machine interacting with a single machine) as well as collective communication (one machine exchanging messages with multiple machines). These communications can be specified to be blocking or non-blocking. The library has been further extended by Geist et al. (1996) to allow remote memory-access, parallel input/output and dynamic creation of processes. MPI guarantees that the interactions between machines are *reliable* (no messages are dropped) and that the messages sent are received in the order they were sent. The communication between the machines is managed by the MPI library which uses the fastest way of interaction such as shared memory or TCP/IP available. MPI excels at breaking single large technical computations over several machines over a network. The MPI library is wildly available via two popular open-source library, OpenMPI (Graham, Woodall and Squyres (2005)) and MPICH2 (Gropp (2002)).

Even though these techniques mitigate data races when used appropriately, they can generate other concurrency-related problems which are difficult to detect. In Go, several errors might occur when using :

- Mutexes: A number of goroutines might be stuck waiting for a lock that is never released by another goroutine. Another issue that can arise is that a

goroutine releases a non-locked mutex which causes a runtime error.

- Synchronisation barriers: A blocked goroutine that is waiting for multiple threads to finish might wait forever if one or more of those threads, for any reasons, never decrement the counter to 0.
- Message-passing concurrency: A send or receive that blocks infinitely, waiting for a sender or receiver respectively.

Many works have been dedicated to developing techniques to find such bugs. There are two types of approaches: *static* and *dynamic*.

2.2.2 Static verification

Static approaches verify that a program meets certain criteria at compile time. Sound static approaches have the advantage of ruling out bugs *before* the code is executed. Two of the main static approaches are type-based and software model checking.

Type-based approach. In type-based approaches, every expression in the program is given a specific type. The program is then type-checked using inductive type rules. If the program is well-typed, then it is proven to be free of specific bugs. Two popular type-based approaches for concurrent programs are session types and ownership types:

- *Session types* is a typing discipline for synchronous and asynchronous concurrent programming languages introduced by Honda (1993) and further developed by Honda, Vasconcelos and Kubo (1998). Session types are protocols defined between two processes in which a set of communication interactions (such as sending and receiving messages) are declared by both participants. The type system enforces that each communication interaction performed by one of the participants in a session is met by the dual

communication interaction of the other (a send is matched by a receive and vice-versa). Hence, if all communications are met by the recipient, then the program is free from deadlocks. Originally, sessions were only binary, meaning that sessions were comprised of exactly two interacting processes. However, this limitation was addressed by Honda, Yoshida and Carbone (2008) with the introduction of multiparty session types which allows sessions with multiple interacting processes. Over the years, the multiparty session types framework has been extended in many ways. For instance, Deniélou and Yoshida (2011) and Castro-Perez et al. (2019) extended multiparty session types to allow an unknown dynamic number of participants at runtime. In addition, Castro-Perez et al. (2019) implemented a tool that infers and decouples role variants from protocols to generate safe Go programs. Bocchi, Yang and Yoshida (2014) improved session types with clocks, clocks constraints and resets. Coppo et al. (2016) extended multiparty session types to support session interleaving which allows the verification of programs with multiple simultaneous sessions at runtime.

Session types can be used to define well-typed protocols between two known interacting components such as a client/server protocol, file transfer protocol or between multiple entities that have a well-defined set of interactions. However, inferring session types statically from programming languages similar to Go is much more complicated because all the parties involved in a session may not be known at compile time. This might be due to a session involving a call to an external library or one of the participants being a third-party computer over the network with unknown message-passing behaviours. In addition, generating a well-typed session might not be feasible due to the number of interactions performed by the participants in a session being undecidable at compile time. This is why works that relies on session types such as Castro-Perez et al. (2019) are more suited to build

correct-by-construction Go software instead of infering sessions from real-world programs. Castro-Perez et al. (2019) proposes a framework to develop safe distributed Go programs using multiparty session types.

- *Ownership types* is a typing discipline introduced by Clarke, Potter and Noble (1998) where each variable in a program is owned only by a single *object* at all times during the execution of a program. The ownership of a variable can be delegated to another object but can only be exclusively accessed henceforth by the new owner. When applied to concurrency, the owner object in question is a thread. This means that all variables in a thread can only be accessed by that particular thread. This implies that a variable cannot be shared between multiple threads. Hence, eliminating data races. Ownership types have been successfully implemented in Rust. Although they mitigate the problem of data races, they can not be used to find deadlocks in concurrent programming languages like Go which is the aim of this thesis.

Software model checking. Model checking is the process of verifying that a model satisfies certain properties by exhaustively checking each state of the model. The models are finite state automatons that are verified exhaustively using a model checker such as SPIN (Holzmann (1997)) or UPPAAL (Larsen, Pettersson and Yi (1997)). Model checking cannot be directly applied to verify programs because, in the undecidability theorem, Turing (1937) proved that any property cannot be soundly and completely verified on a Turing-complete programming language. Instead, a *sound* (which over-approximates the program) or a *complete* (which under-approximate the programs) model is extracted from the program. By over-approximating the behaviour of a program, a sound model extracted from a program that satisfies a particular property automatically confirms that the program it was extracted from satisfies this property in return. However,

if the model does not satisfy the property then this does not mean that the program does not either. This means that approaches that rely on soundness can raise *false alarms*. The reverse can be said about models that under-approximate the behaviours of a program. If a complete model does not satisfy a particular property then the program does not satisfy it as well. Contrary to sound approaches, complete approaches can raise *false negatives*, i.e, the model satisfies a property but not the program.

Software model checking has a long history, see Jhala and Majumdar (2009) for a comprehensive survey.

2.2.3 Dynamic verification

Dynamic approaches execute the program and verify that the execution of the program satisfies a property of interest. Dynamic verification tests that the program does not contain any errors while or after the program is running. This can be achieved by instrumenting the code and/or by having monitors that run concurrently to the main program. Cassar et al. (2017) contains an exhaustive survey on the different runtime monitoring instrumentation techniques. For example, in Vetter and de Supinski (2000), MPI applications are monitored using a tool called Umpire. This tool sits between the application itself and the MPI runtime system. It analyses MPI operations, such as send and receive, and reports if a deadlock occurs. In Francalanza and Seychell (2015), a safety property given in Hennessy–Milner logic is checked against a trace produced by the Erlang Virtual Machine (EVM) to monitor if the running program satisfies the safety property.

Name	Async	Unknown chan size	Chan of chan	Mutex	Wg	Dynamic spawning	Dynamic for	Chan in for	Interface	Chan safety	Liveness
Dingo-hunter	×	×	×	×	×	×	×	×	×	×	✓
Gopherlyzer	×	×	×	×	×	×	×	×	×	×	✓
Gong	✓	×	×	×	×	✓	✓	×	×	✓	✓
Godel	✓	×	×	×	×	✓	×	×	×	✓	✓
Godel2	✓	×	×	✓	×	✓	×	×	×	✓	✓
Nano-go	×	×	×	×	×	×	×	×	×	×	✓
GCatch	✓	×	✓	✓	×	✓	×	×	×	×	✓
GOMELA	✓	✓	×	✓	✓	✓	✓	×	×	✓	✓

Figure 8: Supported features of previous works in the static verification of concurrent Go programs.

2.3 Previous work on the verification of concurrent Go programs.

In this section, we discuss the recent works performed by several research groups to develop a range of theories and tools intended to support developers in finding synchronisation bugs in Go programs, either statically or dynamically.

Table 8 shows a list of previous works based on the static verification of concurrent Go programs as well as our approach, GOMELA, and which Go features and properties they support. All of these Go features or patterns are known to cause trouble to static checkers in terms of applicability and scalability when applied on real world programs. The table is divided into three sections namely, concurrency primitives, complex features of the language and which properties the tools support. The first section shows which static checkers support asynchronous channels, channels declared with a size that cannot be deduced at compile time and channels sent over channels (channels of payload channels). In addition, it shows which works support waitgroups and mutexes. The second section shows which tools support programs that spawn goroutines dynamically, support goroutine spawned or channels declared in `for`-loops and interfaces. Finally, the last section shows which tools verify channel safety properties such as sending on a closed channel and deadlocks.

```
1 ch := make(chan int)
2 ch ← 0
3 go func() {←ch}()
4
```

Figure 9: Example of a goroutine spawned dynamically.

2.3.1 Static verification of Go programs.

Dingo-hunter. The first work on the static verification of concurrent Go program, to our knowledge, is Ng and Yoshida (2016). They proposed a tool to statically detect global deadlocks in Go programs using choreography synthesis introduced by Lange, Tuosto and Yoshida (2015). Their approach consists of inferring *local session types* (a control-flow graph with session primitives) from each goroutine in the SSA representation of a Go program. These local session types represent the concurrent operations performed by each goroutine. Each local type is then translated to a CFSM (Concurrent Finite State Machine) which are FSM labelled with receive and send events. These CFSMs are all synthesised into a single global graph of transitions using the GMC-Synthesis tool introduced in Lange, Tuosto and Yoshida (2015). If the resulting global graph satisfies the conditions described in Lange, Tuosto and Yoshida (2015) then the program is guaranteed to be deadlock-free.

The main limitation of this work is that all goroutines are modeled as being created at the start of the program. This means that a simple program, shown in Figure 9, cannot be verified because a send is performed before the spawning of the receiving goroutine.

Gopherlyzer. Stadtmüller, Sulzmann and Thiemann (2016) proposed a different static verification approach, based on forkable regular expressions, to detect global deadlocks. In this work, Go programs are translated as regular expressions which include a fork operator that models the spawning of a goroutine. The

regular expressions are then translated into a finite state machines based on Brzozowski's derivative construction method Brzozowski (1964) which verifies that the resulting finite state machines are free from global deadlocks (defined as *stuckness* in the paper).

We can see from Table 8 that both of these work (Ng and Yoshida (2016) and Stadtmüller, Sulzmann and Thiemann (2016)) do not support any of the concurrency features listed in the table and only support a small subset of the language. Notably, they do not support asynchronous channels, range-over-channel, `defer` statements, nor structs. In addition, their approaches only verify that programs are free from global deadlocks.

Gong and Godel. Lange et al. (2017, 2018) proposed two more advanced static verification techniques, called Gong and Godel respectively, which approximate Go programs with *behavioural types* Hüttel et al. (2016) through their SSA (static single-assignment) intermediate representation. Various safety and liveness properties can be checked on these behavioural types using bounded executions in Lange et al. (2017) and exhaustive model checking in Lange et al. (2018). The extracted models or behavioural types closely mirror the concurrent behaviours (send, receive, etc) of the programs while abstracting away from data elements. These works aim at being *sound*. This is achieved by over-approximating the behaviour of the program by modelling `if` statements and `for`-loops as non-deterministic choices. In addition to using the model checker mCRL2 (Groote and Mousavi (2014)) to verify the extracted behavioural types, Lange et al. (2018) use the KITTeL termination analyser to check that `for`-loops in the program terminate. Both of these approaches added support for asynchronous channels with statically known bounds and verified various channel safety errors and partial deadlocks. Although Lange et al. (2017) did support goroutines spawned in `for`-loops, it did not scale well (up to 13 minutes to verify a program of 55 lines of

code with only 4 goroutines) and support only a small subset of Go. As a result, it cannot be used to verify real-world programs efficiently. Both are evaluated on small handmade benchmarks which do not reflect real-world Go programs. In the work Lange et al. (2018), the model checker mCRL2 is used to verify modal μ-calculus byKozen (1983) properties while in Lange et al. (2017) the models are verified from specific barbs using an ad-hoc model checker.

Godel2. The work by Gabet and Yoshida (2020) builds on the approach introduced in Lange et al. (2018). In addition to checking the same properties from Lange et al. (2018), it detects mutex-related bugs such as unlocking an unlocked mutex or an infinitely blocked mutex, as well as data races. However, like its predecessor by Lange et al. (2017, 2018), it has limited applicability to real-world programs as shown in Chapter 5.

GCatch. More recently, Liu et al. (2021) developed a different static approach that relies on approximating possible executions and uses an SMT-solver to determine whether they lead to blocking bugs. This detector, dubbed GCatch, combines several *static* bug detectors and includes a novel detector for blocking bugs (BMOC). GCatch is accompanied by GFix which aims at repairing bugs involving at most two goroutines and a single channel. As shown later in Chapter 5, GCatch has a better front-end than GOMELA (fewer crashes) notably, because it relies on the SSA representation of Go program. As a consequence, Go programs must be compiled (all their dependencies met) before they are fed into GCatch.

Nano-go. Midtgaard, Nielson and Nielson (2018) proposed a static verification approach based on abstract interpretation for detecting global deadlocks in a small subset of Go. It does not support asynchronous channels, waitgroups or mutexes nor does it supports recursion.

Unsupported features of previous works. The static approaches mentioned above only provide partial support of the Go language. For instance, as shown in Table 8, none of the static verification frameworks in Ng and Yoshida (2016); Stadtmüller, Sulzmann and Thiemann (2016); Lange et al. (2018); Midtgaard, Nielson and Nielson (2018); Liu et al. (2021); Gabet and Yoshida (2020) can verify programs that spawn new threads within a `for`-loop. The work in Lange et al. (2017) only provides an unsound approximation for such programs. In addition, model checkers based on behavioural types tend to raise too many false alarms, do not scale to large codebases, and support a very limited subset of Go. In contrast, GCatch can handle large codebases, has a low rate of false alarm, but tends to miss many bugs (see Chapter 5). Additionally, it is not easy to predict the type of bugs that GCatch misses.

Limitations of static approaches. One of the key advantages of static approaches is that they verify programs without actually executing them. However, static approaches have certain disadvantages with regards to the scalability and the correctness of feedback of the approaches. Verifying a program with a few lines of code can take a large amount of time to verify, in some cases, only to provide a result that is not correct. Due to these limitations, Johnson et al. (2013) has found that developers were unhappy with static checkers because the feedback provided was unclear and often wrong as well as being too slow.

2.3.2 Dynamic verification of Go programs

In this section, we give an overview of all work related to the dynamic verification of concurrent Go programs. An advantage of dynamic verification approaches is that they raise fewer false alarms because they verify specific executions of the program and may support a larger subset of the language since supporting additional features only requires further instrumentation. However, instrumenting

the code has an impact on the execution of the program which slows down the speed of execution of the programs as well as hiding certain bugs or creating bugs that were not in the original code. In addition, dynamic approaches focus only on a small subset of the potential execution traces that a program can produce which can result in missed bugs in the potential traces that were not verified.

Gopherlyzer-GoScout. Sulzmann and Stadtmüller (2017, 2018) proposed a trace-based method to analyse Go programs which only supports synchronous channels in Sulzmann and Stadtmüller (2017) and an improved approach in Sulzmann and Stadtmüller (2018) which relies on vector clocks and adds supports for asynchronous channels. Both works require the code to be instrumented before the analysis. Their trace-based method explores additional traces by recording which branches in `select` statements have not been chosen at runtime and replays the trace to exhaustively analyse all possible branches using vector clocks. Concretely, the approach instruments the code by reporting *pre(x)* and *post(x)* conditions which specify that the communication action x is available or that it has been successfully carried out respectively. If multiple pre-conditions are available at the same time (concurrently) during the execution of the program, the approach exhaustively analyse all possible interleavings. They developed a tool called *gopherlyzer-Goscout* which can reveal potential global deadlocks and send-on-closed safety errors. This technique mitigates one of the main problems inherent to dynamic approaches, which analyses only one specific trace of execution, by analysing exhaustively (concurrent) alternative traces available. However, they are impacted by the intensive usage of message-passing primitives. For instance, Sulzmann and Stadtmüller (2018) report up to 41% of tracing overhead for programs with high level of concurrency.

GOAT. Another dynamic approach has been proposed by Taheri and Gopalakrishnan (2021) called GOAT. GOAT extends the built-in Go's execution trace

```
1 select {
2   case <-ch:
3     fmt.Println("Received on ch")
4   case ch1 <- 0:
5     fmt.Println("Send on ch1")
6 }
```

Figure 10: Example of a `select` statement which non-deterministically choose between the first available `case`s.

mechanism *runtime/trace* by adding events around concurrency operations in the Go program. GOAT verifies that the trace produced by the program is free from deadlocks. To explore a greater number of potential interleavings rather than a single execution of the program, GOAT injects certain handlers (called `goat.handler()`) which randomly calls the Go runtime to stop executing the currently running goroutine to let other goroutines have a chance to execute. This is achieved by calling `runtime.GoSched()` internally. A downside of this approach is that these injected handlers (purposefully) affect the running time of the program.

GFuzz. The most recent dynamic approach was developed by Liu et al. (2022) called GFuzz. GFuzz uses message reordering (Yuan and Yang (2020)) by mutating the order in which messages are exchanged over channels to find concurrency-related bugs. Instead of looking at all the potential interleaving of messages which increase significantly with every additional communications operation, they use fuzzing guided heuristics to prioritise certain ordering (such as orderings that lead to a full channel). This technique relies on prioritising branches of `select` statements by turning them into sequential conditional statements with timeouts that fall back on the original `select`. In other terms, each `case` branch of the `select` statement is tested one by one alongside a timeout statement. As an example, the Go program that will result from the instrumentation performed by GFuzz on the `select` statement in Figure 10 is shown in Figure 11.

```
switch FetchOrder(_) {
  case 0: // branch one
    select {
      case ←ch:
        fmt.Println("Received on ch")
      case ←time.After(T):
        select { // fall back
          case ←ch:
            fmt.Println("Received on ch")
          case ch1 ← 0:
          fmt.Println("Send on ch1")
        }
    }
  case 1: // branch two
     select {
      case ch1 ← 0:
        fmt.Println("Send on ch1")
      case ←time.After(T):
        select { // fall back
          case ←ch:
            fmt.Println("Received on ch")
          case ch1 ← 0:
          fmt.Println("Send on ch1")
        }
    }
}
```

Figure 11: Instrumentation performed by GFuzz when verifying Figure 10.

From these figures, we can see that both branches have been turned into a `switch` statement that prioritises certain branches over other branches by using a timeout. When comparing GFuzz against GCatch, Liu et al. (2022) found that GFuzz could find many more bugs than GCatch could. However, GFuzz is a dynamic approach that requires running the program and performing much instrumentation to the code.

Limitations of dynamic approaches. Although dynamic approaches have many advantages such as allowing analysis of external libraries, they also have limitations. Firstly, they require the code to be instrumented. Hence, they affect the runtime and behaviours of the program which might, in turn, affect the verification results. In addition, dynamic analysis will always verify a subset of all possible executions of a program whereas static verification attempts to verify all possible traces of execution.

Chapter 3

Empirical Analysis of Concurrency in Go Projects

In this chapter, we present the implementation and results of the empirical analysis on the usage of concurrency that we performed on real-world Go projects. The goal of this empirical analysis is to obtain a better understanding of how and how often the concurrency primitives of Go, and their concurrency operations, are used in practice by analysing publicly available Go projects. This empirical analysis helps us determine what portion of Go code is supported by previous work as well as give us a clearer idea of what needs to be supported the most. In addition, this empirical analysis can also be used by other researchers and practitioners to make well-informed decisions on which direction to take their research in terms of the scalability (towards larger programs) and the applicability (towards a larger subset of Go) of their approaches. In total, we have analysed 125 of the most popular Github projects. To achieve this, we have built a tool that given a Go project, uses Go's native libraries to generate the abstract syntax tree (AST) of the program and pursue an intra-procedural analysis which reports on particular patterns and Go features of interest found in the resulting AST. The empirical analysis aims at answering four research questions:

- **RQ1:** *How often are channels, waitgroups and mutexes used in Go projects?* The Go language provides three popular concurrency mechanisms: channels, waitgroups and mutexes. This research question is about how frequently and intensively these concurrency mechanisms are used in practice and compares which are used more often. We have found that most (96%) Go projects use channel-based operations and use them intensively while 76% use waitgroups and only 58% had at least one mutex declaration.

- **RQ2:** *How often are concurrent operations used relative to their concurrency primitive in Go projects?* The number of concurrency operations per primitive greatly affects the size of the models generated when modelling Go programs. This research question inquires how often each operation is used with regard to their concurrency primitive. This is relevant to both static and dynamic verification since both are impacted by the number of operations and primitives occurring in a program. Static verification frameworks often rely on checking properties of a model (e.g., behavioural type or forkable expression, see Section 2.3) whose size grows with the number of operations and primitives used in the program. In dynamic verification frameworks, the code needs to be instrumented around each operation (see Section 2.3). Hence, if more operations are used, more data need to be recorded and analysed. We have found that the number of operations per concurrency primitives is relatively low, which suggests that programmers use simple protocols or patterns to synchronise threads over concurrency primitives.

- **RQ3:** *How common is the usage of* asynchronous *message passing in Go projects?* The communication channels in Go are *synchronous* by default, which means that both send and receive operations are blocking unless channels are given an explicit bound. The language offers the option of creating

bounded asynchronous channels for which send operations are not blocking as long as the channel is not full. Bounded asynchrony is challenging for a static verification point of view because (*i*) the channel bounds may not be known statically and (*ii*) the state space of the program grows exponentially with the capacity of the channel.

We have found that 59% of the channels in the projects we have analysed are synchronous, while most asynchronous channels are created with a bound of 1 (and 75% have a bound of 4 or less). This suggests that the maximal capacity of asynchronous channels might often be reached in practice.

- **RQ4:** *What concurrent topologies are used in Go projects?* One of the main challenges of statically analysing message passing programs is related to their concurrent topologies, e.g., the number of concurrent threads executing, the number of channels over which they communicate, and whether these numbers are known and finite. It is often impossible to statically determine the (possibly infinite) number of threads and/or channels a program may create at runtime. An infinite or complex concurrent topology may lead to an infinite state-space which renders techniques such as model checking prohibitively costly or impossible. This research question investigates how often complex concurrent topologies are used in practice.

  We found that 97 out of 125 (78%) projects contained programs for which it is not possible to determine the number of threads at compile-time. However, most projects use a number of channels which can be determined statically (78% of projects).

Contributions. In this chapter, in Section 3.1, we describe our methodology, including our data selection and our Go program analyser. In Section 3.2, we present the results of our analysis, answering the four research questions. In Section 3.3, we discuss the limitations of our analysis. We discuss related work in

Table 1: General information about the 150 projects.

Total number of visited projects	150
Number of analysed projects	125
Number of channel-based projects	119
Number of waitgroup projects	95
Number of mutex projects	73
Number of median-sized projects	23

Section 3.4 and give concluding remarks in Section 3.5. Our tool-chain (Dilley and Lange (2022b)) and experimental data (Dilley and Lange (2021d)) are available online.

3.1 Methodology

In this section, we describe the GitHub projects that we have collected and the approach we have used to answer the research questions we set out in the introduction of Chapter 3.

`GoSurvey`. To obtain a better understanding of how the concurrency primitives of Go are used in practice, we implemented a tool-chain, dubbed **`GoSurvey`**, which analyses publicly available Go projects from a list of projects given as input.

Table 1 gives an overview of the total number of projects we have analysed. We have thoroughly analysed 125 projects, totalling 32 million (physical) lines of code. Part of our analysis focuses on two sub-groups: 125 projects which contain at least one concurrency primitive or operation and 23 of similar sizes. From the list of 125 projects, we found that 119, 95, and 73 contained at least one channel, waitgroup and mutex respectively.

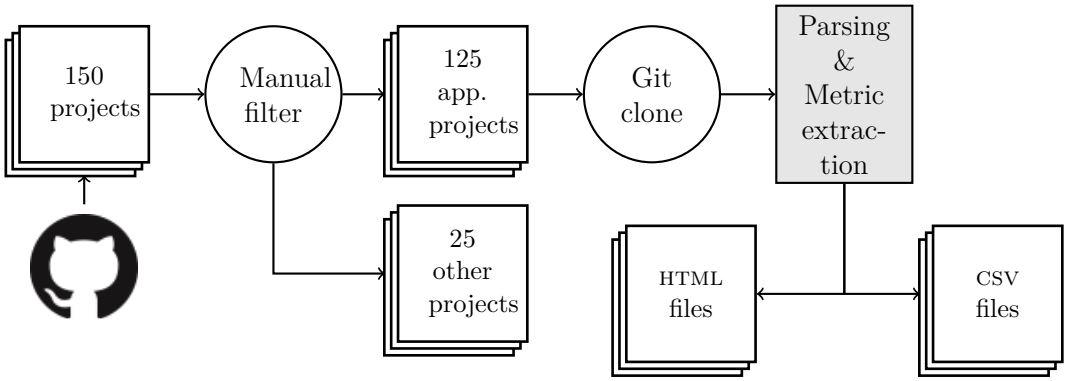


Figure 12: Process of the empirical analysis.

3.1.1 Projects selection

The goal of our empirical analysis is to investigate how and how much developers use the concurrency features of Go in a wide range of application domains. Figure 12 gives an overview of the selection procedure. First, we have selected the *150 most popular Go projects* on GitHub, according to the number of *stars* associated with these projects. Even though this metric might not be the most accurate, Borges and Valente (2018) found that the number of stars generally reflects how many people appreciate or are interested in a project. The selection was made in September 2021 when the star counts of the selected projects ranged from 9899 to 90067 stars. The list was retrieved using a Python script which connects to GitHub's REST API and returns a list of project identifiers. Next, we *manually filtered* the list of projects to remove repositories that do not contain human-made applications. In total, we removed 10 of these projects:

- `quii/learn-go-with-tests`,`chai2010/advanced-go-programming-book`, `unknwon/the-way-to-go_ZH_CN`,`hoanhan101/ultimate-go`,

`greyireland/algorithm-pattern` and `yeasy/docker_practice` which contain exercices from textbooks.

- `astaxie/build-web-application-with-golang` which is a tutorial to build web application in Go.
- `tmrts/go-patterns` which is a list of idiomatic designs and patterns in Go.
- `halfrost/LeetCode-Go` which is a list of Go solutions to difficult interview questions in Go from the famous text platform LeetCode.
- `avelino/awesome-go` which is a list of popular Go frameworks.

We also filtered out 5 projects which could not be parsed by the `go/parser` library, which we use to extract the abstract syntax trees from Go programs, because these projects either contained pre-processors for their programs or were structured in non-conventional ways. These are `golang/dep`, `cockroachdb/cockroach`, `google/gvisor`, `jaegertracing/jaeger`, and `coreybutler/nvm-windows`.

For each of the remaining projects, we have executed a `git clone` command to retrieve the source code locally. Then we automatically removed the top-level `test` and `vendor` directories, to reduce potential noise due to, e.g., usage of third party libraries exposing channels. We note that we preserved unit tests, as they provide insights on, e.g., how an API exposing channels is used. Unit tests related to a given `<file>.go` file are located in the same directory (in a file called `<file>_test.go`). After our analysis, we removed an additional 10 projects from the original list that did not contain any communication feature resulting in a total of 125 projects overall. Note that only 8% of the projects did not contain any concurrency features.

3.1.2 Analysing Go programs

In the next step, our **GoSurvey** traverses the abstract syntax tree of all `.go` files in each cloned repository. The **GoSurvey** is written in Go and relies on Go's internal parser (the `go/ast`, `go/parser` and `go/types` libraries) to compute the number of occurrences of several *concurrency-related features*. We count the occurrences of the following features:

- The channel creation primitive, `make(chan T,` e`)`, with or without a capacity. This feature is useful to determine whether or not a project uses channel-based message-passing concurrency. We also record the capacity e and the type `T` of each channel to determine whether it is asynchronous and/or whether the channel is used to carry other channels respectively.

- The basic channel-based operations: send, receive, and select. As well as the close operation and the range-over-channel statement. These features are useful to determine how intensively channels are used.

- The spawning of a goroutine (e.g., line 16 of Figure 21). We consider occurrences of goroutine and channel creations in `for`-loops as special cases (e.g., lines 22 and 21 of Figure 21).

- The aliasing (or assignment) of a channel within a `for`-loop, as in line 23 of Figure 21. These constructions are used to determine whether a complex topology of concurrent threads is created.

- The usage of channel direction annotations in formal parameters, as in line 6 of Figure 21. These annotations allow us to determine whether developers use this optional stronger typing discipline.

- The declaration of a variable of type waitgroup, `var` wg `sync.WaitGroup`. Variables of type waitgroup or mutex are automatically initialised when

declared in Go compared to channels that needs to be explicitly initialised. This construct is used to determine how intensively synchronisation barriers are used.

- The basic waitgroup operations: `Add(x)`, `Done()` and `Wait()`. We also record the value of `x` when it is a constant and report on the number of `Add(x)` where `x` is a constant.
- The declaration of a variable of type mutex, `var` *wg* `sync.WaitGroup`. This construct allows us to determine how often developers rely on shared memory concurrency compared to message-passing concurrency.
- The two basic operations on mutex: `Lock()` and `Unlock()`.

We expand on some of these features and how they help us answer our research questions in Section 3.2.

The analyser generates a set of CSV files storing the number of occurrences of concurrency-related features, as well as other metrics related to the size of the projects (number of lines of code, files, and packages etc). The analyser additionally generates HTML files. Each HTML file contains the list of features occurring in a given project as well as hyperlinks to their locations on the associated GitHub repository (the links point to a specific line of code and commit snapshot), see Dilley and Lange (2021d).

3.1.3 Project sizes

To compare the level of intensity of message-passing concurrency in projects of significantly different sizes and structures, we present some of our measurements relative to the number of *physical lines of code* (PLOC) using the CLOC command (see Danial (2021) (v1.80)) which discards, e.g., blank and comment lines. Given a project P, we write $|P|$ for its *concurrent size*, i.e., the *sum* of *physical* lines of

code in all `.go` files that contain at least one of the concurrency features described in Section 3.1.2. Mathematically, $|P| = \sum_{f \in F(P)} k\text{PLOC}(f)$ where $F(P)$ is the set of files in P which have *at least one* concurrency-related feature. Focusing on the files with a concurrency aspect allows us to compare the message-passing intensity of projects which may have significantly different sizes but a comparable use of concurrency.

The three biggest projects of the 125 projects in terms of concurrent size are: `Golang (2021a)` (concurrent size = 156 kPLOC), `Vitessio (2021)` (concurrent size = 126 kPLOC), and `Pingcap (2021)` (concurrent size = 115 kPLOC). The `Pingcap (2021)` project holds the well-known distributed Hybrid Transactional and Analytical Processing (HTAP) database *TiDB*. The `Vitessio (2021)` project is a database clustering system for horizontal scaling of MySQL through generalised sharding. This project also has the largest number of waitgroups and mutexes declarations. Finally, `Golang (2021a)` contains the Go compiler, standard library, and runtime. Given the nature of these software, it is not too surprising that they rely heavily upon concurrency-related features, e.g., to allow multiple concurrent requests and having a concurrent runtime engine.

3.1.4 The Core Projects: similarly sized projects

To visualise the usage of concurrency primitives and their operations in absolute terms over similarly sized projects, we selected the projects whose size falls within 30% of the median *concurrent size* $|P|$ of all 125 projects. This resulted in 23 projects. We call these the *core* projects. The median concurrent size of our sample is 4.5 kPLOC, hence the *core projects* consist of projects whose size is between 3.2 and 5.9 kPLOC.

Table 2: Projects using concurrency operations (out of 125 projects).

Feature	projects	proportion
chan	119	95%
receive	120	96%
send	118	94%
select	108	86%
close	83	66%
range	51	41%
waitgroup	95	76%
Add(x)	95	76%
Done()	94	75%
Wait()	95	76%
mutex	73	58%
Lock()	72	58%
Unlock()	72	58%

3.2 Quantitative analysis

In this section, we report and discuss the quantitative results of our analysis for each research question. To answer our research questions, we use our tool-chain to collect occurrences of the different features described in Section 3.1.2. We present our results through descriptive statistics (box plots and numerical tables) and summaries of several manual investigations of a few remarkable projects.

3.2.1 How often are channels, waitgroups and mutexes used in Go projects?

We use the framework described in Section 3.1 to collect occurrences of Go's native concurrency primitives and their operations. Table 2 summarises our findings wrt. occurrence of operations in the 125 projects we have analysed. We note that only 6 projects out of 125 (∼5%) do not create any communication channels, whereas 30 out of 125 (∼24%) do not use waitgroups, and 52 out of 125 (∼42%) do not use mutexes. All 23 core projects had at least one channel creation, one project

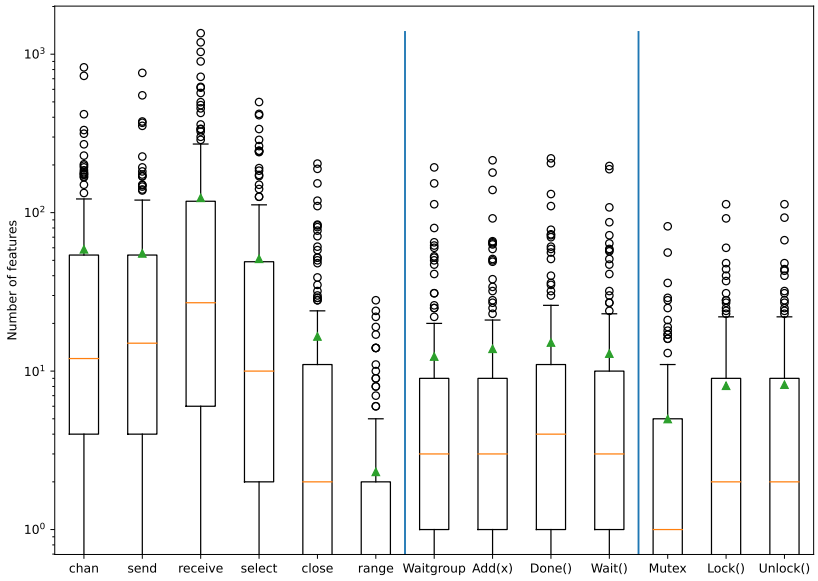


Figure 13: Absolute occurrences of concurrency operations and primitives in 125 projects (y-axis is log scaled). The mean and the median are shown with a green triangle and an orange line respectively.

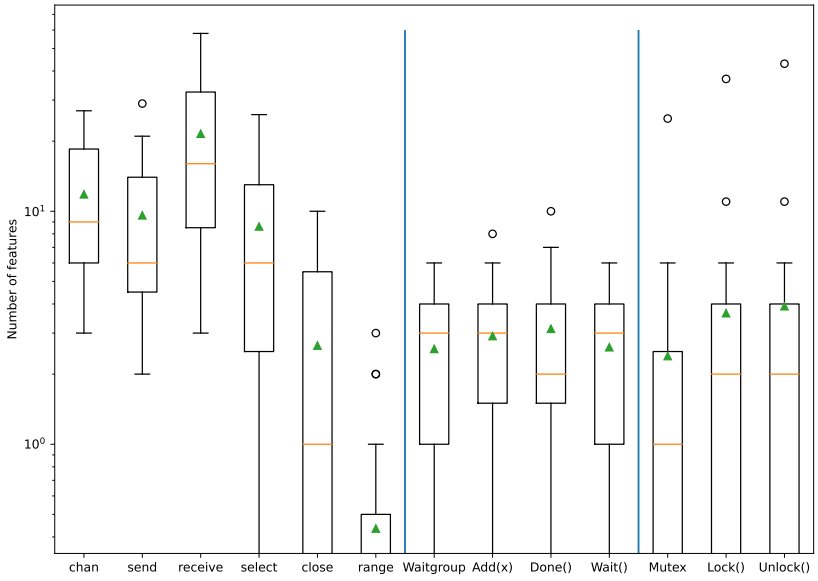


Figure 14: Absolute occurrences of concurrency operations and primitives in 23 core projects (y-axis is log scaled).

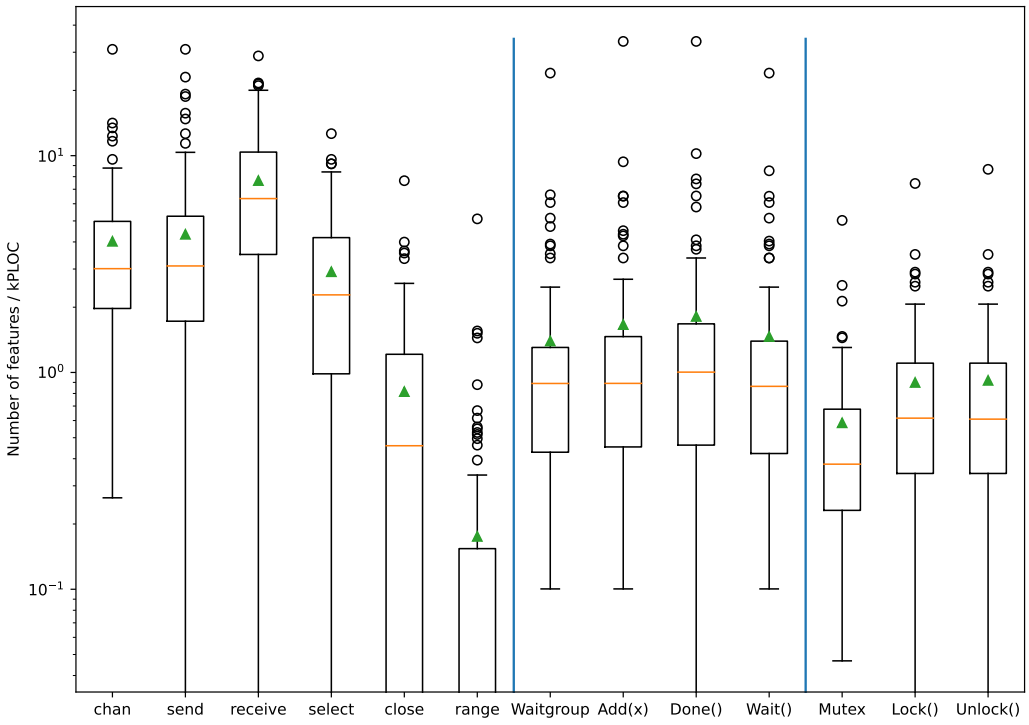


Figure 15: Occurrences of concurrency operations wrt. concurrent size in 125 projects (y-axis is log scaled).

Table 3: Absolute occurrences of concurrency operations and primitives in 125 projects.

Features	mean	std	min	25%	50%	75%	max
chan	58.69	119.75	0	4	15	54	826
send	55.296	105.95	0	4	15	54	762
receive	123.97	235.84	0	6	27	118	1357
select	51.12	93.59	0	2	10	49	499
close	16.54	35.12	0	0	2	11	204
range	2.31	5.08	0	0	0	2	28
waitgroup	12.36	27.33	0	1	3	9	193
Add(x)	13.82	31.08	0	1	3	9	214
Done()	15.15	33.37	0	1	4	11	220
Wait()	12.90	29.37	0	1	3	10	197
mutex	4.98	10.65	0	0	1	5	82
Lock()	8.08	16.43	0	0	2	9	113
Unlock()	8.21	16.78	0	0	2	9	113

Table 4: Absolute occurrences of concurrency operations and primitives in 23 core projects.

Features	mean	std	min	25%	50%	75%	max
chan	11.82	7.97	3.00	6.00	9.00	18.50	27.00
send	9.61	7.28	2.00	4.50	6.00	14.00	29.00
receive	21.52	15.58	3.00	8.50	16.00	32.50	58.00
select	8.61	7.18	0.00	2.50	6.00	13.00	26.00
close	2.65	3.05	0.00	0.00	1.00	5.50	10.00
range	0.43	0.84	0.00	0.00	0.00	0.50	3.00
waitgroup	2.57	1.56	0.00	1.00	3.00	4.00	6.00
Add(x)	2.91	1.95	0.00	1.50	3.00	4.00	8.00
Done()	3.13	2.49	0.00	1.50	2.00	4.00	10.00
Wait()	2.61	1.62	0.00	1.00	3.00	4.00	6.00
mutex	2.39	5.19	0.00	0.00	1.00	2.50	25.00
Lock()	3.65	7.76	0.00	0.00	2.00	4.00	37.00
Unlock()	3.91	8.93	0.00	0.00	2.00	4.00	43.00

did not use any waitgroup, and 10 did not contain any mutex. This suggests that more projects rely on channel-based concurrency and waitgroups rather than mutexes.

Absolute measurements. Figure 13 shows the average (green triangle) and five-number summary (minimum, Q1, median, Q3 and maximum values) of the number of occurrences of primitives and operations in the 125 projects. Note that the y-axis is in logarithmic scale. Figure 13 is divided into three sections, the left part shows the occurrences of channel-based operations, the center part shows the occurrences of waitgroup-related operations while the right section shows the occurrences of mutex-related operations. On average, the 125 projects contained 61.65 occurrences of a channel creation primitive (with a median of 13), 16.26 occurrences of a waitgroup declaration (with a median of 5) and 8.64 occurrences of a mutex declaration (with a median of 4). From this, we can conclude that, on average, channels are used more *intensively* (more occurrences per lines of code) in Go projects than other concurrency mechanisms. This conclusion also holds for the core projects that can be seen in Figure 14 where the average number of channel creations is 11.82 compared to 2.56 waitgroups and 2.39 mutexes per projects. In the rest of this section, we focus on those 125 projects that contain at least one channel, waitgroup or mutex declaration operation. We present both absolute and relative measurements. To give two distinct perspectives on the relative occurrences of concurrent operations, we present results with respect to the *concurrent size* of projects (see Section 3.1.3) and the number of occurrences of concurrency operation with regards to *concurrency primitive* declarations.

Chanel-based absolute measurements. From Figure 13, we can see that the most used operation is the receive operation. The average number of occurrences of receive operations is 130.18 with a median of 28. This operation is also used to model delays and timeouts, which explains why the number of projects with

Table 5: Number of branches in `select` statements.

	mean	std	min	25%	50%	75%	max
branches	2.17	0.65	0.00	2.00	2.00	2.00	14.00

receive operations is greater than the number of projects with channel creations. For instance, the program below waits 2 seconds then prints "Done.".

```
←time.After(2 * time.Second)  // receive
fmt.Println("Done.")
```

Table 3 also shows that the range-over-channel construct is not used intensively. On average, the projects we have analysed contained only 2.43 of such constructs (with a median of 0).

Table 5 shows the size of select statements in terms of the number of cases they contain (including a possible default branch). We observe that select statements have ∼2.17 branches on average. Over the 6390 select statements we have analysed, 1829 (28.6%) included a default branch.

The top project in terms of absolute numbers of channel-oriented features is `Golang (2021a)` (concurrent size = 156 kPLOC) which contains the Go compiler, standard library, and runtime. This project has the largest number of concurrency-related features, i.e., 826 channel creations, 762 send, 1357 receive, 204 close operations, and 412 select statements.

Waitgroup-related absolute measurements. From Table 2 (middle section), we observe that `Add(x)`, `Done ()` and `Wait()` appear in 76% of the projects. 95 projects overall contained the occurrence of at least one waitgroup, `Add(x)`, and `Wait()`. However, one of them, K3s-io (2021b), did not contain any `Done()` call because the method is passed as a higher-order function parameter in K3s-io (2021a) as shown below:

```
once.Do(wg.Done)
```

```
1    wg.Add(2)
2    didRead := make(chan bool, 1)
3
4    go func() {
5      defer wg.Done()
6      ...
7      didRead ← true
8    }()
9
10   go func() {
11     defer wg.Done()
12     ...
13     ←didRead
14   }()
15   wg.Wait()
16 }
```

Figure 16: Simplified example from Golang (2021c) which showcase the uses of multiple `Done()` per waitgroup.

The construct `once` is found in the `go/sync` package and includes a method `Do(f)` which invokes the high order function `x` exactly once even if `Do(x)` is called multiple times. Essentially, `once.Do(wg.Done)` certifies that `wg.Done()` will be called once no matter how many times `once.Do(wg.Done)` is invoked.

In the rest of this section, we focus on those 95 projects that contain at least one waitgroup declaration primitive. In Figure 13 (middle section) and Table 3 (middle section), we can see that the average numbers of occurrences of waitgroup declaration, `Add(x)`, `Done()` and `Wait()` primitives are very similar which suggests that waitgroup usage is generally very simple with one of every action per waitgroup declaration.

The top project in terms of absolute numbers of waitgroup-related operations is `Vitessio (2021)` (concurrent size = 126 kPLOC)

which is the project with the largest amount of waitgroup-related operations overall. It contains 193 waitgroup declarations, 214 `Add(x)`, 220 `Done()` and 194 `Wait()`.

Mutex-related absolute measurements. From Table 2 (bottom section), we observe that `Lock()` and `Unlock()` appear in 58% of the projects. A total of 73

projects contained the occurrence of at least one mutex declaration. However, one of them, Sirupsen (2021b), contained neither a `Lock()` or an `Unlock()` because the mutex variable declared at line 593 in Sirupsen (2021a) is passed to a struct and subsequently referred to as a field of the struct (see lines 603, 607, 617 and 621 in Sirupsen (2021b)) which our tool does not support. Interestingly, we can see that from Figure 13 (right section) and Table 3 (bottom section), the average numbers and medians of `Lock()` and `Unlock()` are double the average of mutex declaration. This suggests that a mutex, on average, is locked and unlocked twice. 308 and to `Unlock()` at lines 272 and 309.

The `Vitessio (2021)` project (concurrent size = 126 *k*PLOC) contains the highest absolute number of mutex primitives. It contains 82 mutex declarations, 113 `Lock()` and `Unlock()`.

Measurements from the 23 core project. Similarly to Figure 13 and Table 3, Figure 14 and Table 4 show the average, standard deviation and five-number summary for the occurrence of concurrency operations in the 23 *core projects*. From these projects, we observe that, even though the y-axis does not show the same numbers, the shapes of the box plots are very similar to the absolute values of Figure 13 and Table 3.

We can also see that, overall, the usage of waitgroup declarations is roughly similar to the number of `Wait()` and that the `Done()` operation is used more intensively than others as explained in section 3.2.1. Within the core projects, two projects, `Docker-slim (2021)` and `Drone (2021)`, have the most channel creation with 27 creations per project in total. `Docker-slim (2021)` (an API that optimizes and secures docker containers) is the project with the highest project size with 5296 PLOC. It contains the second largest number of receive operations (41). The `Drone (2021)` project provides a continuous delivery system built on container technology. This project contains the most receive and close operations

```
1  func wsWriter(..., writeMutex *sync.Mutex) {
2    pingTicker := time.NewTicker(15 * time.Second)
3    ...
4    for {
5      select {
6      case ←pingTicker.C:
7        writeMutex.Lock()
8
9        if ... {
10         writeMutex.Unlock()
11         return
12       }
13
14       if ... {
15         writeMutex.Unlock()
16         return
17       }
18
19       writeMutex.Unlock()
20     ...
21     }
22   }
23 }
```

Figure 17: Simplified example from Photoprism (2021a) that shows why in some project the number of `Unlock()` is greater than `Lock()`.

(58 and 18 respectively). The project with the highest amount of waitgroup operations is `Chi (2021a)` (a router for building Go HTTP services) with an average of 24.04 waitgroup declarations and `Wait()` operations, and 33.65 `Add(x)` and `Done()` operations per 1000 concurrent lines. This is mainly due to one single test file Chi (2021b) making heavy usages of waitgroups.

A project that stands out from the rest of the core projects is `Photoprism (2021b)` because it has four times the amount of mutex declarations (25) and six times the amount of `Lock()` (37) and `Unlock()` (43) than the second largest project.

The fact that the number of `Unlock()` is higher than `Lock()` is explained by the repetition of a pattern shown in Figure 17 from Photoprism (2021a). The parameter `writemutex` of function `wsWriter`, is used as part of a for-select pattern which infinitely waits for one of the branches of the select to fire. In this case, a ticker is used to trigger the branch at line 6 every 15 seconds (the other branches are omitted for brevity). Then, if certain properties hold at lines 9 and 14, the

```
1  func TestEchoStart() (...) {
2    errChan := make(chan error)
3
4    go func() {
5      // code omitted
6      if err != nil {
7        errChan ← err
8      }
9    }()
10
11   waitForServerStart(e, errChan, false)
12 }
```

Figure 18: Simplified example from Labstack (2021b) which contains no receive because the receive is performed in an external function.

mutex is unlocked and the function returns. Otherwise, the mutex is also unlocked and the rest of the branch is executed.

Measurements relative to concurrent size. Our first relative measurements are given with respect to the concurrent size of projects, i.e., $|P|$, the PLOC in the files which contain at least one concurrency feature. For each project P, we divide the number of occurrences of each concurrency feature by $|P|$. Figure 6 and Table 6 summarise our findings for all concurrency operations per 1000 concurrent lines.

Channel-based measurements. On average, we observe that message passing operations are used intensively in concurrency-related files. We find 6.34 channels for every 1000 concurrent lines of code (with a median of 3.01). The relative average number of occurrences of send, receive and select operations are 6.65, 10.31 and 2.67 respectively. The other operations are used significantly less intensively (less than 2).

The `Labstack (2021a)` project, a minimalist web framework, has the largest number of channel creations and send operations relative to its concurrent size (30.94 per kPLOC in concurrency-related files). The fact that their number is similar is explained by the occurrence of a pattern similar to the shown in Figure 18. In Figure 18, `errChan` is used to specify if an error occurred in the spawned gorou-

```
1  func ThrottleWithOpts(ctx context.context,limit int) {
2
3    tokens := make(chan token, limit)
4    backlogTokens := make(chan token, limit)
5
6    select {
7
8      case <-ctx.Done():
9        return
10
11     case <-backlogTokens:
12       timer := time.NewTimer(time.Second * 60)
13
14       select {
15         case <-timer.C:
16           return
17         case <-ctx.Done():
18           timer.Stop()
19           return
20         case tok := <-t.tokens:
21           defer func() {
22             timer.Stop()
23             tokens <- tok
24           }()
25       }
26   }
27 }
```

Figure 19: Simplified example from Go-chi (2021) that shows a common pattern where a channel is returned via a method call and received on. This pattern is generally used to implement *timeouts* and *cancellation signals*.

tine. The number of receive primitives (21.04 per 1000 concurrent lines) is inferior to the number of send operations because the channel is then passed to a function `waitForServerStart` which takes care of the reception.

The `Chi (2021a)` project, a router for building Go HTTP services, has the largest number of receives relative to its concurrent size with 28.85 receives per 1000 concurrent lines of code and the second largest amount of select with 9.61 select operations. This can be explained by the fact that the project makes use of multiple selects that are guarded by receives on timing data-structures (such as Go's native libraries *time* and *context*) which contains a channel that will send after a given amount of time or after a specific action has been performed respectively. A simplified example of this pattern taken from Go-chi (2021) is shown in Figure 19. The `timer` declared at line 12 will send on its channel `C` after 60

seconds. Hence, after 60 seconds the branch at line 15 will be available. Whereas the `ctx` field is a *context* data-structure which is used to carry cancellation signals and deadlines. The channel returned by calling `ctx.Done()` will be closed when the context is cancelled or as reached a timeout. As a result, the receives at lines 8 and 17 will become available.

Waitgroup-related measurements. On average, we observe from Figure 6 and Table 6 that waitgroups are used three times less intensively than channels. We can see that, overall, the usage of waitgroup declarations is roughly similar to the number of `Wait()` and that the `Done()` operation is used more intensively than others. The project with the highest amount of waitgroup operations is `Chi (2021a)` with an average of 24.04 waitgroup declarations and `Wait()` operations, and 33.65 `Add(x)` and `Done()` operations per 1000 concurrent lines. This is mainly due to one single test file (see Chi (2021b)) making heavy usage of waitgroups.

Mutex-related measurements. Figure 6 (bottom section) and Table 6 (bottom section) summarise our findings for mutex-related operations per 1000 concurrent lines. On average, we observe that mutexes are used less intensively than channels and waitgroups. The project with the highest amount of mutex-related operations per 1000 concurrent lines is `Photoprism (2021b)` with 5.03 mutex, 7.44 `Lock()` and 8.65 `Unlock()` per 1000 concurrent lines of code.

3.2.2 How often are concurrent operations used relative to their concurrency primitive in Go projects?

Our second relative measurements are made relative to the number of occurrences of concurrency operations over their relative primitive in each project. Hence, we divide the number of occurrences of each operation such as send, `Add(x)`, etc. by the number of occurrences of its relative concurrency primitive (channel creation

RQ1: We found that 95% of the projects we have analysed used at least one concurrency primitive. This suggests that concurrency is heavily used in real-world Go projects and therefore, developing approaches and implementing tools that verify the concurrent behaviour of Go programs is relevant and important.
We found that channels were used in 20% more projects than waitgroups and 37% more projects than mutexes. This suggests that Go developers rely more on channels than other forms of concurrency.
We found that the receive primitive is the most commonly used operation because we have seen that it is commonly used for implementing delays and timeouts. However, in some cases, the syntactic number of receives were lower than the number of sends and channel creations due to receives being performed within function calls.
We have found that, on average, a mutex is locked and unlocked twice while waitgroups are incremented and decremented on average once.

for sends, waitgroup declaration for `Done()`, etc.). This measurement gives us an approximation of the number of operations invoked on each concurrency primitive. Figure 20 and Table 7 summarise our results.

Measurements relative to the number of channels declarations. On average, there are 1.26 send operations per channel creation (with a median of 1) while there are 2.08 receive operations per channel creation(with a median of 1.56). The slightly higher number of receive operations can be explained by the fact that on average there is approximately a select for every other channels. In turn, select statements have more than two branches on average, see Table 5, and they are generally guarded by receive operations.

Two projects stand out with respect to the number of channel-oriented operations per channel creation: `grpc-gateway` and `node_exporter`, which we have manually analysed. In the `grpc-gateway` project (a gRPC to JSON proxy generator) most channel usages are contained in examples showing how to use the gRPC API.

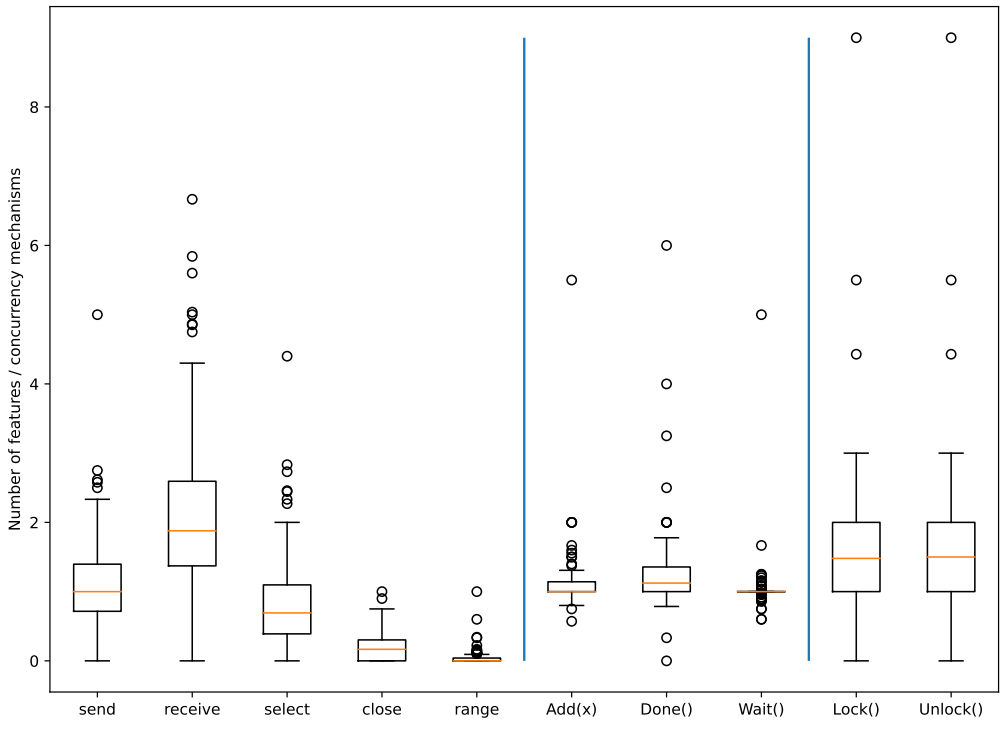


Figure 20: Occurrences of concurrency operations wrt. respective concurrency primitives.

Table 6: Relative occurrences wrt. concurrent size in 125 projects.

Features	mean	std	min	25%	50%	75%	max
chan	6.34	6.43	0.23	2.78	4.69	7.83	71.43
send	6.65	7.86	0.00	2.33	4.63	7.84	75.28
receive	10.31	10.28	0.00	4.26	7.95	12.95	123.66
select	2.67	3.09	0.00	0.52	1.92	3.70	34.41
close	1.54	2.99	0.00	0.00	0.51	1.79	34.48
range	0.44	2.60	0.00	0.00	0.00	0.20	58.82
waitgroup	1.39	2.65	0.10	0.43	0.89	1.30	24.03
Add(x)	1.66	3.66	0.10	0.45	0.89	1.46	33.65
Done()	1.81	3.72	0.00	0.46	1.00	1.68	33.65
Wait()	1.46	2.74	0.10	0.42	0.86	1.39	24.04
mutex	0.59	0.70	0.04	0.23	0.38	0.68	5.03
Lock()	0.90	1.06	0.00	0.34	0.62	1.10	7.44
Unlock()	0.92	1.16	0.00	0.34	0.61	1.10	8.65

The `node_exporter` project contains several instances of send operations sending several (53) hard-coded variations of a struct. These two examples are extreme cases of the operation to channel ratio. However, as shown in Figure 20 (left section) and Table 7 (top section), the interquartile range is very close to the mean. Therefore, our results suggest that the number of syntactical occurrences of features over a given channel is fairly low, which further suggests that channels are used to support simple synchronisation protocols.

Measurements relative to the number of waitgroups declaration. Our second relative measurements are made relative to the number of occurrences of waitgroup declarations in each project. Hence, we divide the number of occurrences of each waitgroup operation `Add(x)`, `Done()` and `Wait()` by the number of occurrences of waitgroup declarations. This measurement gives us an approximation of the number of operations invoked on each waitgroup. Figure 20 (middle section) and Table 7 (middle section) summarise our results.

Table 7: Relative occurrences wrt. their relative concurrency primitives in 125 projects.

Features	mean	std	min	25%	50%	75%	max
send	1.11	0.67	0.00	0.71	1.00	1.39	5.00
receive	2.13	1.20	0.00	1.23	1.86	2.55	6.67
select	0.46	1.33	0.00	0.28	0.50	0.77	2.50
close	0.21	0.28	0.00	0.00	0.17	0.31	1.00
range	0.12	0.15	0.00	0.00	0.00	0.04	1.00
Add(x)	1.15	0.52	0.57	1.00	1.00	1.14	5.50
Done()	1.30	0.70	0.00	1.00	1.12	1.36	6.00
Wait()	1.04	0.43	0.60	1.00	1.00	1.00	5.00
Lock()	1.70	1.17	0.00	1.00	1.48	2.00	9.00
Unlock()	1.72	1.17	0.00	1.00	1.5	2.00	9.00

The project with the highest number of waitgroup-related operations per waitgroup declarations is Etcd-io (2021) with 5.5 `Add(x)`, 6 `Done()` and 5 `Wait()` per waitgroup declarations. After manual analysis, we have found such high numbers can be explained by the fact that one waitgroup is declared as a global variable and used by several functions in the package. However, for all the projects on average, the number of `Add(x)`, `Done()` and `Wait()` is very close to the number of waitgroup declarations (1.15, 1.30, 1.04 respectively). This also shows that waitgroups are generally used for simple protocols where one waitgroup declaration is followed, on average, by a single occurence of each waitgroup operation (`Add(x)`, `Done()` and `Wait()`).

Measurements relative to the number of mutex declarations. The project with the highest number of mutex-related operations per mutex declaration is Beego (2021a) with 9 `Lock()` and `Unlock()` per mutex declarations. After manually analysing the project, we found that such large numbers are explained by the fact that a mutex is declared as a global variable and used in the body of 15 function declarations in the same package. This pattern can be found in Beego

(2021b).

Except for this project, the average number of `Lock()` and `Unlock()` is almost double the number of mutex declarations (1.70 and 1.72 respectively).

RQ2: We found that the number of operations per concurrency primitive is low. Waitgroups and mutexes had slightly more than one of each operation on average per declaration, while channels had 1.11 sends and 2.13 receives per channel declaration. This suggests that concurrency primitives are used for simple synchronisation protocols. This also suggests that, as the protocols are simple, slicing techniques based on the usage of concurrency primitives might drastically reduce the size of the models generated. In Chapter 4, we discuss how we partition Go programs into independently verifiable components.

3.2.3 How common is the usage of *asynchronous* message passing in Go projects?

Go offers two types of channels: synchronous (default) and asynchronous. In this section, we study how frequently programmers use asynchronous channels compared to synchronous ones. For asynchronous channels, we investigate how often their bounds can be determined statically and give statistics on their sizes. We use the framework described in Section 3.1 to collect occurrences of channel creation primitives and record channel bounds, whenever the capacity of a channel is known at compile time. Table 8 lists the number of occurrences of each type of channel. Overall the projects we have analysed contained more than 7k channels. For a large majority (91%) of the channels, we were able to determine their bounds statically: either synchronous (60%) or a non-zero capacity known at compile time (32%), i.e., a hard-coded integer or a constant.

Table 9 gives our results concerning the sizes of *asynchronous* channels whose bounds are statically known. We observe that most asynchronous channels are set to hold at most one message, while a capacity of over 4 is uncommon. A

Table 8: Communication channels in 119 projects.

Type	occurrences	proportion
All channels	7336	100%
Channels with known bounds	6680	91%
Synchronous channels	4338	60%
Asynchronous channels (known)	2342	32%
Channels with unknown bounds	656	9%

Table 9: Known sizes of asynchronous channels.

	mean	std	min	25%	50%	75%	max
size	125.04	2129.28	1	1	1	4	10^5

few projects use channels with a very large capacity to simulate unbounded asynchrony. For instance, the project `dgraph` contains one channel of size 10^5 to implement a channel which is used to receive a statically unknown number of requests without blocking. Similar uses-cases can be found in the `minio` and `vitess` projects with channels of size 10^4.

RQ3: We observed that synchronous channels are the most commonly used channels (60%). Whenever asynchronous channels are used, they are generally created with a *statically known* bound, which is less than or equal to 4 in 75% of the cases. This means that model-based static approaches can generate models that are far more precise when the size of a channel is known statically. Otherwise, static approaches must model asynchronous channels as unbounded channels which increases the complexity of the model and increase the number of false alarms or, generate multiple models with different bounds for the channel which might not reflect the actual bound given at runtime.

3.2.4 What concurrent topologies are used in Go projects?

In this section, we investigate whether programs containing complex concurrent topologies are common in practice. We measure the complexity of a concurrent topology by counting the occurrences of programming patterns which may (i)

create one or more goroutines, (*ii*) create one or more channels, or (*iii*) store channels in complex data structures.

Figure 21, adapted from Golang (2021b), gives an example of a complex concurrent program implementing a concurrent version of the Sieve of Eratosthenes (an algorithm to compute all prime numbers under a bound) which contains several of such concurrent programming patterns.

The program consists of three functions. Function `generate` iteratively sends an integer on channel `ch`. Function `filter` iteratively reads an integer from channel `in` and, if it is not divisible by `p`, sends it over channel `out`. Function `main` is the entry point of the program. It spawns an instance of function `generate`, then reads a `bound` given by the user (the definition of `readFromUser()` is elided). Next, the function loops `bound` times, spawning new instances of `filter` which are linked together by freshly created channels (`ch1`).

The concurrent prime sieve program contains several complex concurrency patterns which are generally not supported by existing static verification techniques, e.g., a goroutine is spawned within a `for`-loop, see line 22 and a channel is also spawned within the same `for`-loop at line 21. In particular, `bound` is not known at compile time. Hence, for any statically computed abstraction to be sound, one needs to assume that the number of goroutines and channels created by the program is potentially infinite. Additionally, because of the channel aliasing occurring in line 23, these goroutines and channels form a complex topology by linking each pair of threads with a distinct channel.

Goroutine creation. The first part of Table 10 summarises our analysis on the frequency of different patterns of goroutine creations in the 125 projects. The table shows that 99% of the projects we have analysed contain at least one thread creation (i.e., the keyword `go`) and 85% contain at least one occurrence of a thread creation within a `for`-loop (i.e., `go` in (any) `for`). We distinguish between thread

```
1  func generate(ch chan int) {
2    for i := 2; ; i++ {
3      ch ← i // send
4    }
5  }
6  func filter(in ←chan int, out chan← int, p int) {
7    for {
8      i := ←in // receive
9      if i%p != 0 {
10       out ← i // send
11     }
12   }
13 }
14 func main() {
15   ch := make(chan int)
16   go generate(ch)
17   bound := readFromUser()
18   for i := 0; i < bound; i++ {
19     prime := ←ch // receive
20     fmt.Println(prime)
21     ch1 := make(chan int)
22     go filter(ch, ch1, prime)
23     ch = ch1
24   }
25 }
```

Figure 21: Concurrent prime sieve.

creation within a *bounded* `for`-loop, as in line 7 of Figure 7 and *unknown* `for`-loop, as in line 18 of Figure 21. A `for`-loop is *bounded* if our analyser found a constant limiting the number of iterations. For the purpose of static verification, a `for`-loop with a known bound could be unfolded. Table 11 summarises the size of the bounds we have encountered and the top of Table 12 summarises the relative occurrences of patterns in projects which contain at least one occurrence of such a pattern. Note that the bounds of bounded `for`-loops are generally rather large (∼603 on average, with a median of 10).

Channel creation. The second part of Table 10 gives the proportion of projects where channels are created within a `for`-loop. The second part of Table 12 summarises the *relative* number of occurrences of these patterns in projects which contain at least one occurrence of such a pattern. Such patterns can be used to create potentially infinitely many communication links between threads. Observe

Table 10: Frequency of concurrency patterns in 125 projects.

Feature	projects	proportion
`go`	124	99%
`go` in (any) `for`	106	85%
`go` in bounded `for`	61	49%
`go` in unknown `for`	97	78%
`chan` in (any) `for`	27	22%
`chan` in bounded `for`	10	8%
`chan` in unknown `for`	26	21%
channel aliasing in `for`	1	0.08%
channel in slice	31	8%
channel in map	0	0%
channel of channels	14	11%

Table 11: Known bounds of `for`-loops containing `go`.

	mean	std	min	25%	50%	75%	max
bound	603.34	5858.85	1	5	10	100	120000

that channel creation within a `for`-loop is much less common that thread spawning. Again, we distinguish between channel creations within *bounded* `for`-loops as these could be unfolded as part of a static analysis. Only 22% of the projects that we have analysed included a `for`-loop containing a channel creation. The usage of channel creation within a *bounded* `for`-loop is less common (8%). A pattern of specific interest is "channel aliasing in `for`" which corresponds to `for`-loops where a channel variable is assigned to another channel (as in line 23 of Figure 21). Channel aliasing can be used to create a potentially unbounded chain of linked threads as in the concurrent prime sieve program (Figure 21). We have manually analysed all occurrences of "channel aliasing in `for`" in our sample and found *no* occurrence resembling the pattern in Figure 21. In fact, our investigation revealed that most occurrences of channel aliasing or creation within a `for`-loop are used to initialise dynamic structures containing channels (e.g., an array of `struct` whose

Table 12: Relative occurrences wrt. concurrent size.

Patterns	mean	std	min	25%	50%	75%	max
`go`	245.09	338.64	1.99	30.38	132.61	314.24	1844.19
`go` in (any) `for`	54.46	86.79	0.66	7.17	24.75	64.12	474.49
`go` in unknown `for`	44.94	71.75	0.66	6.69	18.52	55.29	433.67
`go` in bounded `for`	23.17	39.53	1.85	4.59	9.43	22.66	208.47
`chan` in (any) `for`	13.38	16.68	1.01	3.64	8.50	13.31	84.75
`chan` in unknown `for`	10.43	11.38	1.01	3.66	6.83	12.42	55.93

records contain channels). See Elastic (2021) for a concrete example.

Channel storage. Another challenge for static verification is related to the usage of dynamic structures (arrays, lists, etc.) to store channels because, e.g., static analyses generally cannot determine at compile time which index of an array is being accessed. Go supports arrays, slices (lists in Go) and maps natively. The last part of Table 10 shows that only 8% of the projects we have analysed use slices (including arrays) to directly store channels. Note that there was no occurrence of a map of channels. The last line of Table 10 shows that very few projects (11%) use channels to carry other channels, i.e., `make(chan chan T)`. Channel passing is a remarkable feature as it allows channel references to be passed around, as in the π-calculus (see Milner, Parrow and Walker (1992)).

Finally, we have analysed the occurrences of channels as formal parameters of Go functions, which may be specified as send or receive only, as in line 6 of Figure 21. Channel direction annotations restrict the concurrent topologies: they enforce channels to be unidirectional. We found that in 55% of the cases channel formal parameters had a specified direction.

RQ4: 85% of the projects we have analysed include thread creations within `for`-loops, a pattern which is not (soundly) supported by earlier static verification frameworks. This is the main challenge that we are tackling in our approach. Most projects (79%) use a *bounded* number of communication channels. In addition, we found that channel passing and channel contained in slices were rarely used (<11% of projects).

3.3 Limitations of our analysis

The main limitations of our analysis are related to data selection and metric extraction. Kalliamvakou et al. (2014) found that extracting data from GitHub involves the risks of including repositories which contain personal or inactive projects, or are used as free storage. For this analysis, we are interested in *any* code written as part of a Go program, hence inactive or personal projects do not pose a particular problem. Repositories used as free storage are unlikely to attract more than 9899 GitHub stars.

Our analysis relies on a traversal of the *abstract syntax tree* of Go files in which we count the syntactical occurrences of different concurrency features. All of the projects we have analysed parsed successfully. We do *not* conduct an inter-procedural analysis. This implies that we under-approximate the number of goroutines and concurrency created in `for`-loops if these are created within a (non-anonymous) function itself called within the `for`-loop. Also, we may fail to recognise channels that are sent over channels if they are packaged into a struct. It is also possible that some programmers may wrap Go primitives such as send and receive in ad-hoc functions in which case the number of such primitives will be under-approximated by our approach. To count the number of occurrences of channel aliasing in `for`-loops, our tool records which identifier refers to channels with respect to *syntactic* equality. Hence, we may fail to identify channels which are referred to by two equivalent, but syntactically different, identifiers, e.g., arrayChan[2] and arrayChan[1+1]. This implies that we may under-approximate

the number of occurrences of channel aliasing within a `for`-loop. The analysis of concurrent topologies considers `for`-loops as the only iterative construct from which complex topologies can be created. This assumption rules out complex topology constructions based on recursive functions. However, we note that Go being an imperative programming language, `for`-loops are more common. We note that `while` loops do not exist in Go.

We have chosen two metrics to study the relative occurrences of message passing primitives: the size $|P|$ of a project and the number of channels. It is possible that choosing different measurements would be a better choice to study the intensity at which message passing is used in Go projects.

Concerning the applicability of our study, we note that our experimental data and analyser are available online at Dilley and Lange (2022b, 2021d).

3.4 Related surveys on the usage of concurrency in programs.

Tu et al. (2019) have performed a study on concurrency bugs in real-world Go programs. They manually analyzed a total of 171 bugs that they found in six of the most popular open-source Go projects (these included Docker and Kubernetes). Their main observation is that Go developers tend to rely more on shared memory primitive such as Mutex. However, our analysis shows that channels are the most used concurrency primitive overall. This can be explained by the fact that our analysis looks at the syntactical occurrence of specific constructs (such as $ch := \texttt{make(chan T}, e\texttt{)}$ or $\texttt{var}\ wg\ \texttt{sync.Mutex}$) and, therefore, may undermine the actual numbers. Tu et al. (2019) also showed that message-passing-based software is as liable to errors as other concurrent programming techniques.

Chabbi and Ramanathan (2022) have studied the Uber Github repository for the presence of data races. They have found over 2000 data races spread over 46

million lines of code and fixed 1011 of them. One of their observations was that Go developers tends to use more concurrency and synchronisation constructs than in Java. They also observe that mixing the use of channels and shared memory makes code more difficult to reason about and, as a result, more susceptible to data races.

Several studies have investigated the usage of concurrency constructs in different programming languages, using publicly available source code. Marinescu (2014) studied the usage of Message Passing Interface (MPI) in open source applications, where the usage of MPI functions were extracted using a string matching algorithm rather than traversing the abstract syntax tree. Wu et al. (2016, 2015) studied the usage of concurrency in C++ through an analysis of nearly 500 open-source applications and a developers' survey. Their analysis focuses on traditional concurrency mechanisms such as thread-based and lock-based constructs. Pinto et al. (2015); Torres et al. (2011) have conducted a study of more than 2000 Java projects from Sourceforge and a survey of 164 programmers. Their findings show that traditional concurrent programming constructs (e.g., threads and `synchronized` methods) are used often (contained in more than 75% projects) and intensively. These results echo the frequency and intensity at which message-passing is used in Go projects. Okur and Dig (2012) analysed 655 open-source applications which use Microsoft's libraries for parallel programming. They notably show that 37% of their data-set of C♯ applications use multi-threading and that 90% of library usage was focused on a small fraction of API methods. Tasharofi, Dinges and Johnson (2013) studied Scala programs that mix actor-based concurrency and other concurrency models. They found that 80% of them mix the actor model with another concurrency model. Whether Go programmers mix channel-based concurrency with other concurrency models is currently an unanswered question.

The applicability of static analyses in real-world programs is the focus of other

related works. Landman, Serebrenik and Vinju (2017) study the usage of Java reflection in a wide range of open source applications. They focus on understanding the limits of a large corpus of static analysis approaches due to the usage of Java reflection. They found that most projects include parts that are hard to analyse. Our findings lead to a similar conclusion for Go projects, most of which include code that is hard to verify statically. We note that the current literature on verification of Go programming is much more limited than that of Java programming. Saboury et al. (2017) study the presence of code smells in JavaScript projects and their relationship to faulty software. Our work may be a starting point for a similar study on code smells and message-passing-related errors in Go.

Other studies have investigated the concurrency-related problems programmers face and how they address them. Lu et al. (2008) study the characteristics of real-world concurrency bugs. They analysed a set of randomly selected bugs from the bug tracking databases of MySQL, Apache, Mozilla, and OpenOffice. All the bugs analysed concern traditional shared-memory concurrency. Pinto, Torres and Castor (2015) study the top 250 most popular questions about concurrent programming on StackOverflow. They have found that most common questions concern threading and synchronisation in mainstream programming languages such as Java. It would be interesting to conduct similar studies with a focus on message-passing programming languages.

3.5 Conclusions

To conclude, through a syntactic analysis of Go projects on GitHub, we have discovered that most projects do use message-passing concurrency, but most use simple synchronisation patterns involving a few send and receive primitives for each (generally synchronous) channel, a few `Lock ()` and `Unlock()` per mutex and a few `Add(x)`, `Done ()` and `Wait()` per waitgroup.

Furthermore, we have identified a set of popular concurrency constructs which are not supported by previous works that rely on extracting behavioural types. These constructs are goroutines spawned in `for`-loops, waitgroups, mutexes and unbounded asynchronous channels.

Goroutines spawned in `for`-loops. One of the most important challenges for future static verification tools of concurrent Go programs concerns `for`-loops which spawn a statically unknown number of goroutines. We have shown that this pattern appears frequently in Go projects. We have found that 85% of projects (see Figure 10) contained at least one `for`-loops which spawned goroutines. In addition, 49% of the projects contained a goroutine spawned in a `for`-loop where the bound of the `for`-loop could not be determined statically. Goroutine spawned in `for`-loop are not supported (soundly) by previous works. Supporting this feature will greatly increase the number of Go projects supported.

Mutex and Waitgroups. We have found that waitgroups and mutexes are commonly used in Go projects. They are used in 76% and 58% of projects respectively. All previous work do not support waitgroups nor mutex except Liu et al. (2021) and Gabet and Yoshida (2020) which supports mutexes.

Asynchronous channels. We have found that out of 7336 channels, 2998 (40%) were asynchronous and 656 (9%) had a bound that could not be determined statically. Asynchronous channels with a statically unknown bound are not supported by previous works.

Other concurrency constructs. Finally, we have found that other potentially un-tractable topologies involving an unbounded number of channels or channels carrying other channels are much less common.

Main takeaways. This empirical analysis allowed us to define which of the concurrency features not supported by previous work was used most often in real-world projects. In particular, we have found that a majority of the projects had at least one goroutine spawned in a `for`-loop and made use of waitgroups and mutexes which are all features not fully supported by previous work. We have also found that the number of concurrency operations per concurrency primitives were low which convinced us that Go programs could effectively potentially be partitioned based on the usage of concurrency primitives. In the next chapter, we give a detailed explanation of how we added support for those features and the strategy we use to partition Go programs.

Chapter 4

Verifying Concurrent Go Programs.

In this chapter, we present the static verification approach that we have developed which extracts parametrized Promela models from Go programs and verifies each model from a set of user-provided bounds using the model checker SPIN. The verification process verifies that the resulting models are free from various concurrency-related safety errors and global deadlocks. Our approach is inspired by a behavioural types approach first formalised in Lange et al. (2017) and Lange et al. (2018) and adds support for additional key constructs that are often used in real-world projects as we have seen in Chapter 3. These constructs are goroutines spawned in `for`-loops, waitgroups, mutexes and statically unknown bounded channels. Our approach consists of a combination of four key insights:

(1) To deal with programs whose concurrent structure depends on arguments that are decided at run-time, we extract *parameterised* behavioural types (i.e., models) from programs. These models can then be verified up-to user-provided bounds. Tracking parameters that affect the concurrent structure of programs allows us to decrease the number of false alarms, which plagued earlier behavioural types-based approaches.

(2) Based on the results of the empirical analysis in Chapter 3, we have found that, on average, there were few concurrent operations per concurrency primitive which suggests that the concurrency primitives are used for simple protocols.

With this in mind, our approach isolates the concurrent behaviours and scopes of concurrency primitives by partitioning the program into the function declarations in the program that do not take concurrency primitives as parameters. We infer a model for each of these function declarations. These models can then be verified separately reducing the overall verification time. Software model-checking verification approaches are known to have scalability issues due to the state explosion problem. Thus, reducing the size of the model by partitioning the program into smaller models can greatly reduce the overall number of state in the models. As we will see in Chapter 5, this technique allows us to deal with large codebases which means that we can verify a large project such as Kubernetes (2021b) (more than 3 million LoC) in 26 minutes. Most projects are checked in under 4 minutes.
(3) Our approach supports programs that coordinate over the three main concurrency primitives (channels, waitgroups, mutexes), which are used in the majority of projects analysed as we have seen in the empirical analysis from Chapter 3. Our approach can easily be extended to support more primitives.
(4) Our approach is explicit wrt. the subset of Go it supports, and which constructs are over-approximated. Additionally, our tool returns confidence levels when it is applied to parameterised programs. Hence, it is easier for developers to understand the risk and potential cause of false alarms.

We describe the technical insights of our approach using two bugs our tool called GOMELA discovered in the wild with Example 2 and Example 3.

Example 2. The program in Figure 22 is adapted from code found on the GitHub repository of `trillian` Google (2021), a verifiable data store developed at Google. At line 5, function `preload()` spawns |`trees`| worker goroutines which

```
1  func preload(trees []string, n int) {
2    ch := make(chan string, n)                    // new chan with capacity n
3    limitCh := make(chan int, runtime.NumCPU())
4    for i := 0; i < runtime.NumCPU(); i++ {
5      limitCh ← 1 // send token on chan limitCh
6    }
7    var wg sync.WaitGroup
8    for _, t := range trees {
9      wg.Add(1)           // increment wg counter
10     go func(v string) { // spawn goroutine
11       ←limitCh // receive token before starting work
12       s := DoSomeWork(v)
13       ch ← s
14       limitCh ← 1 // return token
15       wg.Done()    // decrement wg counter
16     }(t)
17   }
18   go func() { // spawn goroutine
19     wg.Wait() // wait for wg to reach 0
20     close(ch) // set ch to closed
21   }()
22   for s := range ch { // receive message from ch
23     if IsError(s) {
24       return
25     }
26   }
27 }
```

Figure 22: Example of a blocking bug, adapted from Google (2021).

send the result of `DoSomeWork()` over channel `ch`. To limit the number of concurrent threads, each goroutine acquires a token (receive at line 11) before executing `DoSomeWork()`, and returns it (send at line 14) before terminating. Note that the parent thread fills channel `limitCh` with tokens at lines 4-5.

At line 18 the parent thread spawns another goroutine that waits for the worker goroutines to finish using a waitgroup (`wg`). When all goroutines have invoked `wg.Done()`, operation `wg.Wait()` succeeds, and channel `ch` is `close`d.

After spawning |`trees`|+1 goroutines, the parent is ready to consume the data sent on `ch` via a `range` loop on `ch` (line 22). This construct iterates over each element sent on `ch` until the channel is `close`d. If a message contains an error (i.e., `isError(s)` returns `true`), `preload()` returns.

Figure 22 contains a subtle bug that leads to several goroutines becoming permanently stuck. Consider the case where $0 <$ `runtime.NumCPU()` and $0 <$ `n` $<$ |`trees`|-1. The number of send actions on channel `ch` is greater than

```
1  func FindAll(K int, M int) []P {
2    var wg sync.WaitGroup
3    wg.Add(K)
4    found := make(chan int)
5    limitCh := make(chan bool, M)
6
7    for _, pr := range K {
8      limitCh ← true
9      go func() {
10       found ← ... // Produce value
11       wg.Done()
12       ←limitCh
13     }()
14   }
15   go func() {
16     wg.Wait()
17     close(found)
18   }()
19   var results []P
20   for p := range found {
21     results = append(results, p)
22   }
23   return results
24 }
```

Figure 23: FindAll example adapted from Google (2020).

its capacity, hence some worker goroutines will be blocked at line 13 until the parent thread receives some messages (line 22). If `preload()` returns (line 24) before consuming all messages, it may leave up to |`trees`|−`runtime.NumCPU()` goroutines *permanently* blocked.

Example 3. Figure 23 shows an example of a bug found by our approach in a `Google` open source project called `GOPS`. The program makes use of two channels, `found` and `limitCh`, to coordinates the `K` threads spawned at line 9. `found` is used to receive the values computed by each goroutine synchronously (line 20) while `limitch` limits the number of goroutines executing simultaneously to `M` (the capacity of the channel). Furthermore, the program uses a waitgroup `wg` (line 2) that waits for all goroutines (line 16) to finish executing to `close` channel `found` (line 17). This piece of code contains a global deadlock when the value of `K` is strictly greater than `M` which are both given as parameters of function `FindAll`. This is because the number of sends at line 8 will be greater than the size of

channel `limitCh` and therefore the M^{th} send will block forever and all goroutines spawned at line 9 will block waiting to send on `found`.

Blocked goroutines are problematic even when the program as a whole is not stuck. They cannot be garbage collected, hence they silently consume resources until the whole program terminates. For instance, if function `preload` or `FindAll` is called often with the "wrong" arguments, this would affect significantly the memory footprint of the program.

Programs like Example 2 and Example 3 are difficult to reason about because they use several coordination mechanisms in non-trivial ways which makes it hard to enumerate all possible interleavings. This is particularly challenging when the number of spawned goroutines and/or the capacity of channels are unknowns, as the programmer has to think of how different values will affect the possible executions.

Our main insight to verify these programs in a scalable way is to first identify the parameters which directly affect their concurrent structures, e.g., |`trees`|, `n`, and `runtime.NumCPU()` in Example 2. Given such parameters, we extract parameterised models from Go code which are then verified using bounded model checking. We use SPIN as a back-end, but our approach is not necessarily bound to it. Given a user-defined set of values, we model-check every possible instantiation of a parameterised model. Our tool returns the result of each verification, as well as an aggregate score to reflect the number of instantiations that failed.

Contributions. In this chapter, we describe in Section 4.2.1 a novel algorithm to extract parameterised behavioural types from Go code. In addition, we describe a bounded model checking technique to verify these behavioural types in Section 4.3. We have implemented these algorithms in a tool, GOMELA, outlined in Section 4.4.

4.1 Promela as behavioural types

The crux of our technique is to over-approximate Go programs with behavioural types as in Lange et al. (2017), where each Go function is assigned a type codifying its interactions with concurrency primitives, thus abstracting away from non-concurrency related constructs but preserving the concurrent behaviours. Our approach has three key differences compared to earlier works. (*i*) We use a subset of Promela (the language of SPIN) as our behavioural type language. This has the advantage of giving us a direct implementation strategy, while keeping the extraction function relatively abstract. Indeed the subset of Promela we consider here is very close to the language of types defined in Lange et al. (2017). (The model extraction we present in Section 4.2 can easily be adapted to other modelling languages that feature processes communicating over channels.) (*ii*) While the work in Lange et al. (2017) and its extensions Lange et al. (2018); Gabet and Yoshida (2020) abstract away from *all* computational aspect, our behavioural types do keep track of *some* data when it directly affects the structure of the concurrent programs. (*iii*) We support the three main concurrency primitives of Go.

SPIN, implemented by Holzmann (1997), verifies models specified in Promela wrt. properties expressed in linear temporal logic (LTL). A Promela model consists of a set of processes that interact over channels.

4.1.1 A Promela primer

In this section, we present a brief overview of the subset of Promela used in our approach using Figure 24 which shows the main Promela constructs. Promela processes are declared using `proctype` $f(\overline{p})\{\overline{\mathrm{S}}\}$ where f is the name of the process, $\overline{p}$ is a list of (typed) parameters, and $\overline{\mathrm{S}}$ is a list of Promela statements. The starting state of the model is declared using a special proctype called `init`. Promela statements and expressions include basic Boolean and arithmetic expressions, as

`S`	A Promela statements
p	A (typed) parameters
`proctype` $f(\overline{p})\{\overline{\texttt{S}}\}$	Definition of process f
`init`$\{\overline{\texttt{S}}\}$	Initial state of the model
`for` $n..m$ $\{\overline{\texttt{S}}\}$	Bounded for-loop
g	A guard (Boolean expression)
`do` $\overline{g}$ `od`	Unbounded iteration statement
`if` $\overline{g}$ `fi`	Conditional
$[g]\alpha/\texttt{S}$	Guard statement
`run` $f(\overline{e})$	Spawn f with arguments $\overline{e}$
`chan` c `[x] of` $\{\texttt{T}\}$	Channel creation of size x with type T
$ch?\langle\overline{x}\rangle$	Receive
$ch!\langle\overline{e}\rangle$	Send
break	Terminates an iteration statement

Figure 24: Key statements of Promela.

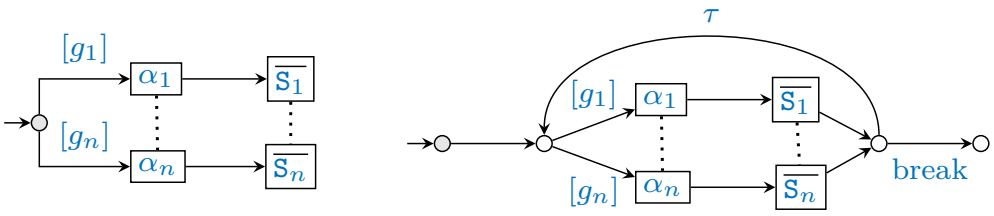


Figure 25: Graphical representation of `if` and `do` statements in Promela.

well as declarations and instantiations of variables and structures.

Promela provides two types of control-flow constructs: loops and branching. Loop `for` $n..m$ $\{\overline{\texttt{S}}\}$ repeats statements $\overline{\texttt{S}}$ for $(m-n)+1$ iterations. Branching constructs encode (possibly non-deterministic) choices between several behaviours. Promela uses `if` and `do` constructs to encode choices. The branches of a choice may be guarded or not. A guarded branch is labelled with $[g]$ or $[g]\alpha/\texttt{S}$ where g is a guard (Boolean expression), α is either a send or receive action, and `S` is a Promela instruction. An `if` statement, which has a similar semantic as *switch statements* in other programming languages, is a non-deterministic choice between multiple guarded branches, while `do` statements are used as an iterative guarded choice between multiple guarded branches. In other words, a `do`

statements infinitely repeats a conditional statement until its execution is terminated by a **break** statement. Figure 25 shows a graphical representation of an **if** (left) and a **do** (right) statements. A guarded branch is fireable when its guard g holds and a matching event for α is available. Upon firing, instruction S is executed. Unguarded branches (labelled with τ) can fire (silently) at any point. Statement **run** $f(\overline{e})$ spawns a new instance of process f with arguments $\overline{e}$. Channels are a special data-type over which processes can communicate. For example, **chan** c [2] **of** {**bool**} creates a new channel c with capacity 2 that can carry Boolean values. Like goroutines in Go, processes may send expressions $\langle \overline{e} \rangle$ over channel ch with $ch!\langle \overline{e} \rangle$. They can receive messages with $ch?\langle \overline{x} \rangle$ which binds the received values to variables $\overline{x}$. We omit the payload $\langle e \rangle$ when a send/receive is used as signal only.

Example 4. The Promela model shown in Figure 26 shows a non-deadlocking implementation of the well-known dining philosophers problem in Promela. The program is made of 3 proctypes (processes), **init**, philo and aFork. The **init** proctype creates three channels, which represent the forks of the model, spawns three philosophers by giving them their respective left and right forks and finally spawns the aFork processes. The forks in the model can be taken by the philosophers via receiving from their respective channels and put back via sending. This is why the aFork process infinitely **do**es a sequence of sends and receives to allow the philosophers to take and put the fork back respectively. A philo process takes two channels as parameters, the left and right forks and repeatedly tries to grab either the left or the right fork at line 22 and line 31 respectively. If it succeeds to grab the left fork at line 24, it will either try to grab the right fork or put back the left fork at line 28. If it succeeds at grabbing both forks, it releases both forks by sending on both forks and goes back to the outer **do** loop.

```
1  init {
2    chan fork1 = [0] of {int}
3    chan fork2 = [0] of {int}
4    chan fork3 = [0] of {int}
5    run philosopher(fork1,fork2)
6    run philosopher(fork2,fork3)
7    run philosopher(fork3,fork1)
8    run aFork(fork1)
9    run aFork(fork2)
10   run aFork(fork3)
11 }
12
13 proctype aFork(chan fork) {
14   do
15   :: fork!0 →
16     fork?0
17   od
18 }
19
20 proctype philo(chan lFork, rFork){
21   do
22   :: lFork?0 →
23     do
24     :: rFork?0 →
25       rFork!0;lFork!0  // release both forks
26       break
27     :: true →
28       lFork!0
29       break
30     od
31   :: rFork?0 →
32     do
33     :: lFork?0 →
34       lFork!0;rFork!0  // release both forks
35       break
36     :: true →
37       rFork!0
38       break
39     od
40   od
41 }
```

Figure 26: Example of the dining philosopher problem with 3 philosophers encoded in Promela using channels as forks (adapted from Lange et al. (2017)).

4.1.2 The SPIN model checker

SPIN checks properties of Promela models that are encoded either via an LTL formula and/or using assertions (*error states*) in the model. SPIN checks that all possible executions of the model validate the LTL formula and/or that no execution reaches an error state. SPIN explores all possible states of a model, hence models must be *finite-state*, i.e., they cannot spawn infinitely many processes.

While other general-purpose model checkers have support for inter-process communication such as mCRL2 (Groote and Mousavi (2014)) or the LTS Analyzer (Magee and Kramer (1999)), we found that Promela is closer to Go and thus allows a nearly one-to-one translation from Go. SPIN has notably been used to verify multi-threaded Java programs in Havelund and Pressburger (2000) and multi-threaded C programs in Zaks and Joshi (2008).

4.2 Extracting parameterised models

In this section, we describe our approach to extract several Promela models from a Go program and how they are verified using SPIN.

Partitioning a Go program. For each Go function that does *not* take a concurrency primitive as parameter, we generate a model. Each function becomes an entry point to a model that can be verified independently. Concretely, a partition corresponds to a function whose concurrent behaviour (wrt. message-passing concurrency) is not affected by other parts of the program, i.e., the function does not send/receive messages to/from other parts of the program. We say that a function is *independent* when it does not take channels, waitgroups nor mutexes as arguments and it does not return a channel, a waitgroup nor a mutex. We generate a model for each declaration of an independent function, and each model is verified independently. This strategy allows us to decompose the verification of

large programs into smaller pieces. Besides the benefit wrt. scalability, this decomposition allows our tool to give function-level feedback when a bug is detected by the model checker.

The model extraction is done via three procedures: (i) a top level procedure translates declarations of Go functions to Promela processes; (ii) procedure $\mathcal{E}_S(\overline{s})$ identifies the concurrency parameters in Go statements $\overline{s}$; (iii) procedure $\mathcal{T}_S(\overline{s})$ extracts a (parameterised) model from Go statements $\overline{s}$.

The models we generate consist of two types of Promela processes: a) *primitive processes* which model concurrency primitives (channels, waitgroups, mutexes) and b) *function processes* which model Go functions and goroutines.

4.2.1 Extracting concurrency parameters

Our goal is to identify the computational elements of a function that affect its concurrent structure, i.e., integer expressions in the source program that affect the number of spawned goroutines, the number of exchanged messages, and the values held in the counters of waitgroups.

We define function $\mathcal{E}_E(e)$ which extracts the concurrency parameters from a Go expression e, by computing its free variables and other unknown references. We give the definition $\mathcal{E}_E(e)$ for the key cases below:

$$\mathcal{E}_E(e) \triangleq \begin{cases} \emptyset & \text{if } e \text{ is an integer } \textit{literal} \\ \{x\} & \text{if } e \text{ is an integer } \textit{variable} \\ \mathcal{E}_E(e_1) \cup \mathcal{E}_E(e_2) & \text{if } e = e_1 \otimes e_2 \text{ with } \otimes \in \{\texttt{+}, \texttt{*}, \ldots\} \\ \{l\} & \text{if } e = l \text{ or } e = \texttt{len}(l) \\ \{f.\overline{a}\} & \text{if } e = f(\overline{a}) \end{cases}$$

The first two cases deal with integer literals and variables. The second cases applies $\mathcal{E}_E(e_i)$ recursively on arithmetic expressions. The fourth case deals with

collections which are abstracted to the name of the collection itself. The final case deals with function calls for which we generate a fresh name, based on the name of the function — these will become global parameters in the generated models. We extend the definition of $\mathcal{E}_E(\overline{e})$ to lists of expressions in the natural way.

Next, we define $\mathcal{E}_S(s)$ which extracts concurrency parameters from expressions that occur in selected Go statements.

1. If $s = ch := \texttt{make(chan T}, e)$, then $\mathcal{E}_S(s) \triangleq \mathcal{E}_E(e)$, i.e., the parameters that set the capacity of the channel.

2. If $s = wg.\texttt{Add}(e)$, then $\mathcal{E}_S(s) \triangleq \mathcal{E}_E(e)$, i.e., we return the parameters that set the delta added to the waitgroup.

3. If $s = \texttt{for i} := e_1; e_2; r\ \{\overline{s}\}$, we apply heuristics depending on the shape of the loop to identify a variable that can represent the number of iterations.

4. If $s = \texttt{for i,x} := \texttt{range}\ l\ \{\overline{s}\}$, then $\mathcal{E}_S(s) \triangleq \{l\} \cup \mathcal{E}_S(\overline{s})$, i.e., we abstract a collection to its size.

5. If $s = f(\overline{a})$ or $s = \texttt{go}\ f(\overline{a})$, we first extract the concurrency parameter of f. Assume we have $\texttt{func}\ f(\overline{x\ \texttt{T}})\ \texttt{T}\ \{\overline{s}\}$ such that $\mathcal{E}_S(\overline{s}) = Y$. Then we construct the sub-sequence of arguments whose position match a concurrency parameter of f, i.e., $\overline{b} = [a_i \mid x_i \in Y,\ 1 \leq i \leq k]$. We have $\mathcal{E}_S(s) \triangleq \mathcal{E}_E(\overline{b})$.

6. For select, conditionals, and range over channels, we apply $\mathcal{E}_E(\_)$ recursively following the abstract syntax tree, e.g., if $s = \texttt{if}\ e\ \texttt{then}\ \overline{s_1}\ \texttt{else}\ \overline{s_2}$, then $\mathcal{E}_S(s) \triangleq \mathcal{E}_S(s_1) \cup \mathcal{E}_S(s_2)$.

We define two families of concurrency parameters: *mandatory* and *optional*. An *optional* parameter x is only used as bounds of a *non-dynamic* `for`-loop, i.e., Cases (3) or (4) above when the loop is not dynamic. All other parameters are *mandatory*. We will see below that mandatory parameters must be instantiated to

```
1  func main() {
2    a := make(chan int)
3
4    numWorkers := os.Args[1]
5    numResponses := os.Args[2]
6
7    for i := 0; i < numWorkers; i++ {
8      go worker(i, a)
9    }
10   for i := 0; i < numResponses; i++ {
11     k := <-a        // receive
12     fmt.Println(k)
13   }
14   close(a)
15 }
16
17 func worker(j int, x chan<- int) {
18   x <- j: // send
19 }
```

Figure 27: Example of concurrent workers. This program contains a bug when `numResponses` $\neq$ `numWorkers`. If `numResponses` $<$ `numWorkers` the programs leads to a send on close runtime error. In the case where `numWorkers` $<$ `numResponses`, executing the program leads to a global deadlock.

construct models that can be verified effectively. Instantiating optional parameters is not necessary, but helps discard false alarms (see Section 4.3).

Example 5. Let $\overline{s}$ be the body of `preload()` from Figure 22. We extract three concurrency parameters from these statements, i.e., $\mathcal{E}_S(\overline{s}) = \{n, \mathit{trees}, \mathit{runtime.NumCPU}\}$. These are all *mandatory* parameters. The first two are instantiated at every invocation to `preload`, and *runtime.NumCPU* is a global (implicit) parameter (instantiated once per model).

Example 6. Let $\overline{s}$ be the body of `ThrottleWithOpts()` from Figure 19. In this example, there is only one *mandatory* communication parameter `limit` which defines the capacity of channel `tokens` , i.e., $\mathcal{E}_S(\overline{s}) = \{\mathit{limit}\}$.

Example 7. In this example, we showcase the extraction of an *optional* parameter. Let $\overline{s}$ be the body of `main()` from Figure 27. In this example, there are two

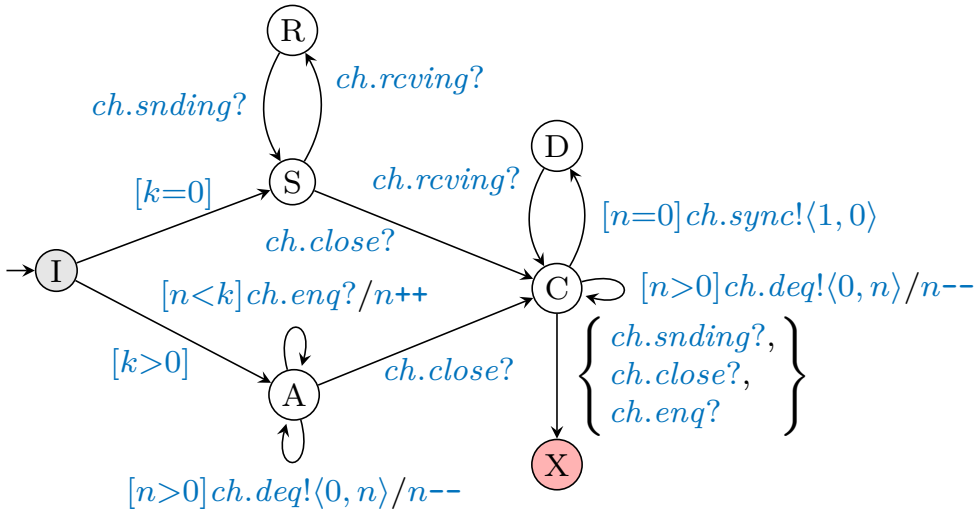


Figure 28: Primitive processes for channels.

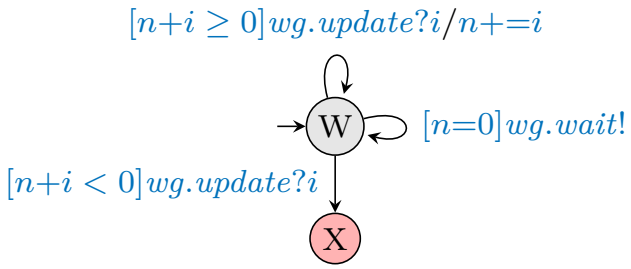


Figure 29: Primitive processes for waitgroups.

communication parameters, `numWorkers` and `numResponses`. In this case, $\mathcal{E}_S(\overline{s})$ returns $\{numWorkers, numResponses\}$.

`numWorkers` is used as the bound of a dynamic `for`-loop and is, as a result, mandatory while `numResponses` is used as the bound of a non-dynamic `for`-loop and, as a result, is *optional*.

4.2.2 Primitive processes: channel, waitgroup, and mutex

We describe Promela processes which model the main concurrency primitives of the Go language (channels, waitgroups, and mutexes). Each of these processes uses several Promela channels and variables stored in a structured data type.

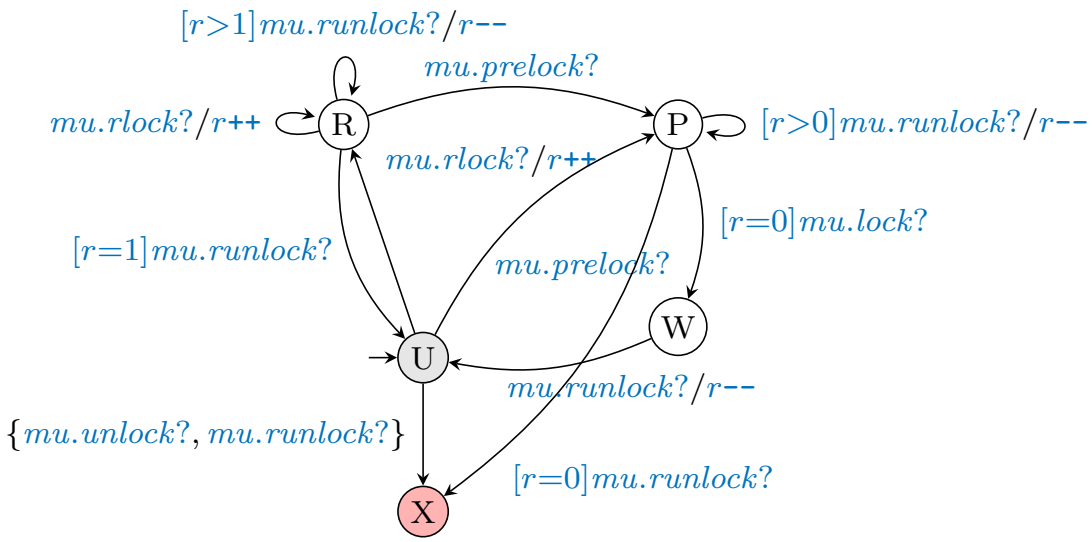


Figure 30: Primitive processes for mutexes.

The declaration of a concurrency primitive in Go is translated to spawning the corresponding primitive process in Promela.

Channels. Figure 28 gives a graphical representation of the Promela process (*channel process*) that models a Go channel. This process monitors all channel interactions: it keeps track of the state of the channel to detect any safety bug. Interactions over asynchronous Go channels are fully mediated by the channel process; interactions over Go synchronous channels also rely on Promela's own synchronous channels.

The channel process uses two (local) variables: k represent the capacity of the buffer (k=0 when the channel is synchronous), and n is the number of messages currently stored in the buffer (note that $n \leq k$).

The channel process interacts with its environment via six synchronous Promela channels: *sync* is a channel which directly models the corresponding *synchronous* Go channel; *snding* and *rcving* monitor sending and receive actions on the *synchronous* channel; *enq* and *deq* model enqueue (send) and dequeue (receive) operations on *asynchronous* channels; *close* is used to receive closing requests. Only

channels *sync* and *deq* carry payloads, i.e., pairs $\langle c, n \rangle$ where c represents the state of the channel (c=1 if the channel is closed, c=0 otherwise) and n is the number of messages in the channel.

The behaviour of the channel process depends on its capacity (k). If k=0, the channel is synchronous and the channel process merely monitors sending and receiving actions on channel *ch*. Goroutine processes send on *snding* (resp. *rcving*) whenever they send (resp. receive) on channel *sync* (see Section 4.2.4). If k>0, the channel is asynchronous and all interactions over that channel are mediated via the channel process. When the channel is not full (n<k), it is ready to enqueue more messages (via *enq*); as long as the channel is not empty, it is ready to emit messages (via *deq*).

Synchronous and asynchronous channels behave equivalently once they are closed. Sending on a closed channel (via *sync* or *enq*), or closing it again (via *close*), leads to the error state. It is always possible to receive from a closed channel, the channel process is always ready to match such requests on *ch.sync* or *ch.deq*. In Section 4.2.4, we show that all function processes are always ready to interact with either the synchronous or asynchronous version of a channel process.

The interactions allowed by the processes exactly mirror the behaviours of synchronous and asynchronous channels in Go except that Gomela abstracts away the exchanged data as explained in Section 4.1 and only model the (possibly blocking) concurrent operations. Essentially, instead of modelling channels as First in First out (FIFO) queues, each channel primitives holds a counter (n) to model the number of messages currently held by the channel.

Waitgroups. Figure 29 gives a representation of the Promela *waitgroup process*, representing a Go waitgroup. The process uses one local variable and two synchronous Promela channels: *wait* and *update*. Variable n represents the current number of threads *wg* is waiting for. When n=0, it is ready to fire *wg.wait!* (thus

unblocking a process waiting on it). Other processes interact with the waitgroup by adding value i to n (where $i \in \mathbb{Z}$). Any update that renders n negative leads to the error state. The waitgroup primitives exactly model the behaviours allowed by waitgroups in Go. The specification of waitgroups in Go can be found in Google (2022a). Note that, `Done()` calls are modelled as *wg.update?-1*.

Mutex. Figure 30 gives an automata representation of the *mutex process*, representing a mutex *mu*. We use the same process to model Go's `Mutex` (traditional mutex) and `RWMutex` (read/write mutex).

The mutex process models interactions over a Go mutex using one local variable (r) and four synchronous Promela channels. Channel *lock* (resp. *unlock*) is used to take (resp. release) the lock of a traditional mutex. Channel *rlock* (resp. *runlock*) is used to take (resp. release) the lock of a read/write mutex. Variable r keeps track of the number of readers that have acquired the read-only lock. Unlocking a mutex that is not locked leads to the error state.

The mutex primitive was inspired and generated from Go's own documentation(Google (2022b)). When evaluating our tool in Section 5.2, we found one intricate behaviour when using `RWMutex` in Go that is not modelled appropriately. The bug arises due to the behaviour of rwmutex which blocks subsequent `Rlock()` when a `Lock()` is ready to trigger. This behaviour is missed by our approach because it is not supported by our current implementation of rwmutexes. The automaton of rwmutex can be seen in Figure 30. There are two reasons why this behaviour is missed. Firstly, when in state R, the mutex does not allow for any mu.lock?. Secondly, when the automaton reaches the W state, it does not allow any mu.runlock?. We plan to remedy this in future work. This behaviour is discussed in more detail in Section 5.2.

4.2.3 Function processes: declaration and call sites

Given a Go program, we generate a Promela model for each of its functions that does not take any concurrency primitive as parameter *and* that initialises at least one concurrency primitive in its body. For instance the program in Figure 27 will produce one Promela model, whose entry point corresponds to function `main()`. Given such a function, we analyse all the functions it invokes (inter-procedurally), and for each invoked function that takes at least one concurrency primitive (such as `worker(i, a)` which takes `a`, a channel, as parameter at line 8), we model it with a Promela *function process*, which we include in the model of the entry point function.

Function declarations. Our models abstract away from non-concurrency related aspects, hence the definitions of function processes include only parameters of their corresponding Go functions that pertain to concurrency. Given a function signature `func ExampleFunction(..., x T,...)`, parameter `x` is abstracted away if `x` is not a concurrency parameter of `ExampleFunction` or if `T` is not the type of a concurrency primitive. Each parameter `x` whose type is a concurrency primitive is mapped to one primitive process. Each parameter `x` that is a concurrency parameter is mapped to an integer parameter, e.g., if the type of `x` is a collection in Go, it is mapped to an `int` in Promela (corresponding to the size of the collection).

We illustrate the approach with the Go program in Figure 31 (top). Assume `OuterFunc` does not take any concurrency primitive as parameter, hence it becomes the entry point of a Promela model. Because `InnerFunc` is invoked by `OuterFunc`, the model will also contain a (function) process definition corresponding to its declaration.

Assume `x` is the only concurrency parameter of `InnerFunc`, i.e., `y` $\notin \mathcal{E}_S(\overline{s})$. Then, `InnerFunc` is mapped to a Promela process which takes three parameters:

```
func InnerFunc(ch1 chan int, x map[string]int, y int) { s̄ }
func OuterFunc(...) {
    // ...
    InnerFunc(ch2, 10, 20)
    // ...
    go InnerFunc(ch2, z, z*2)
    // ...
}
```

```
init { // model entry point
       // ...
    run OuterFunc(...)
}
proctype InnerFunc(ChannelProcess ch1, int x, chan ret) {
    T_S(s̄)
    ret!0
}
proctype OuterFunc(...) {
    // ...
    chan ret1 = [1] of {bool}
    run InnerFunc(ch2, 10, ret1)
    ret1?0
    // ...
    chan ret2 = [1] of {bool}
    run InnerFunc(ch2, z, ret2)
    run (ret2?0)
    // ...
}
```

Figure 31: Blocking vs. concurrent function calls in Go (top) and their models in Promela (bottom).

(i) a `ChannelProcess` (`ch1`) structure which implements the channel process discussed in Section 4.2.2; (ii) an integer (`x`) which corresponds to the length of the `map` parameter; and (iii) the *return channel*, i.e., a buffered Promela channel with capacity 1 (`ret`) which is used to model blocking function calls, as we explain below. The body of a function process consists in the translation of the body of its Go counterpart — using $\mathcal{T}_S(\overline{s})$, see Section 4.2.4 and Section 4.2.5 — followed by a send on `ret` (notifying that the function has returned).

Call sites. Our translation deals with statements of the form $f(\overline{a})$ or `go` $f(\overline{a})$ as follows. If f is an external function or if it does not take a concurrency primitive as a parameter, the corresponding call site is *skipped*. In the former case, we optimistically assume that f is not buggy; in the latter case, f is an entry point to its own model which is verified independently. If the declaration of f is available, then we translate both $f(\overline{a})$ and `go` $f(\overline{a})$ to the creation of a new Promela channel followed by the spawning of a new function process. For blocking calls, the process spawning is followed by a (blocking) reception on the return channel. We illustrate this aspect with the translation of calls to `InnerFunc` in Figure 31.

4.2.4 Operations on concurrency primitives

All operations on channels, waitgroups, and mutexes are translated to Promela operations that interact with one of the primitive processes. Figure 32 gives an overview of the mapping from Go operations to Promela using a graphical representation of the latter.

Translating channel operations is slightly more involved as the bounds of Go channels might not be known at compile-time, hence a function process taking a channel ch as a parameter needs to be ready to send or receive on a synchronous or asynchronous version of ch. As a consequence, both send and receive operations are modelled as composite Promela operations. A send operation ($ch \leftarrow e$) is

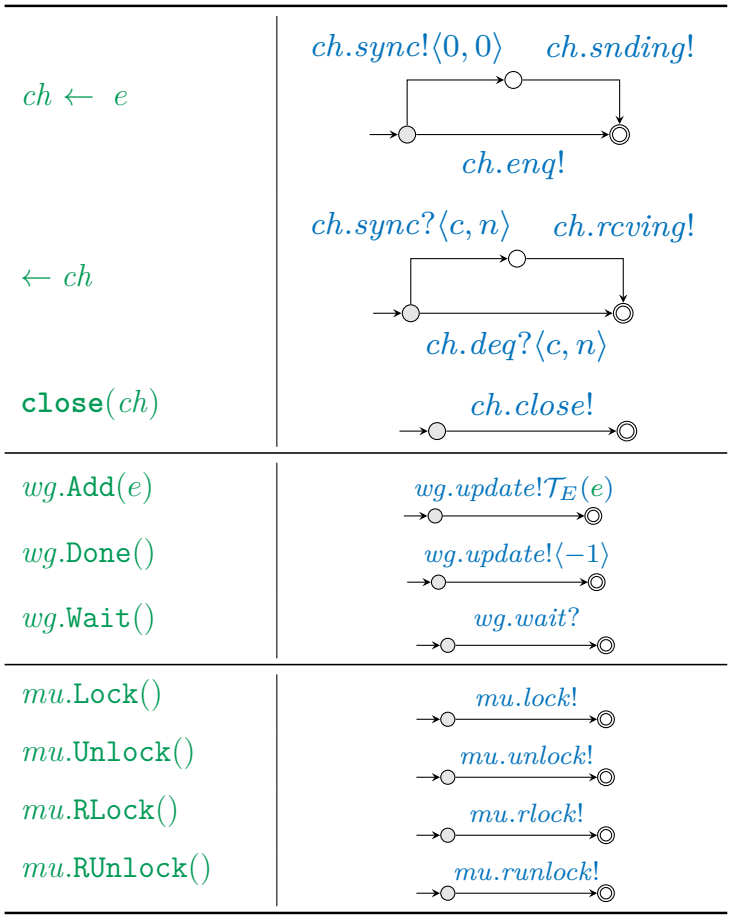


Figure 32: Overview of the translation function $\mathcal{T}_S(s)$.

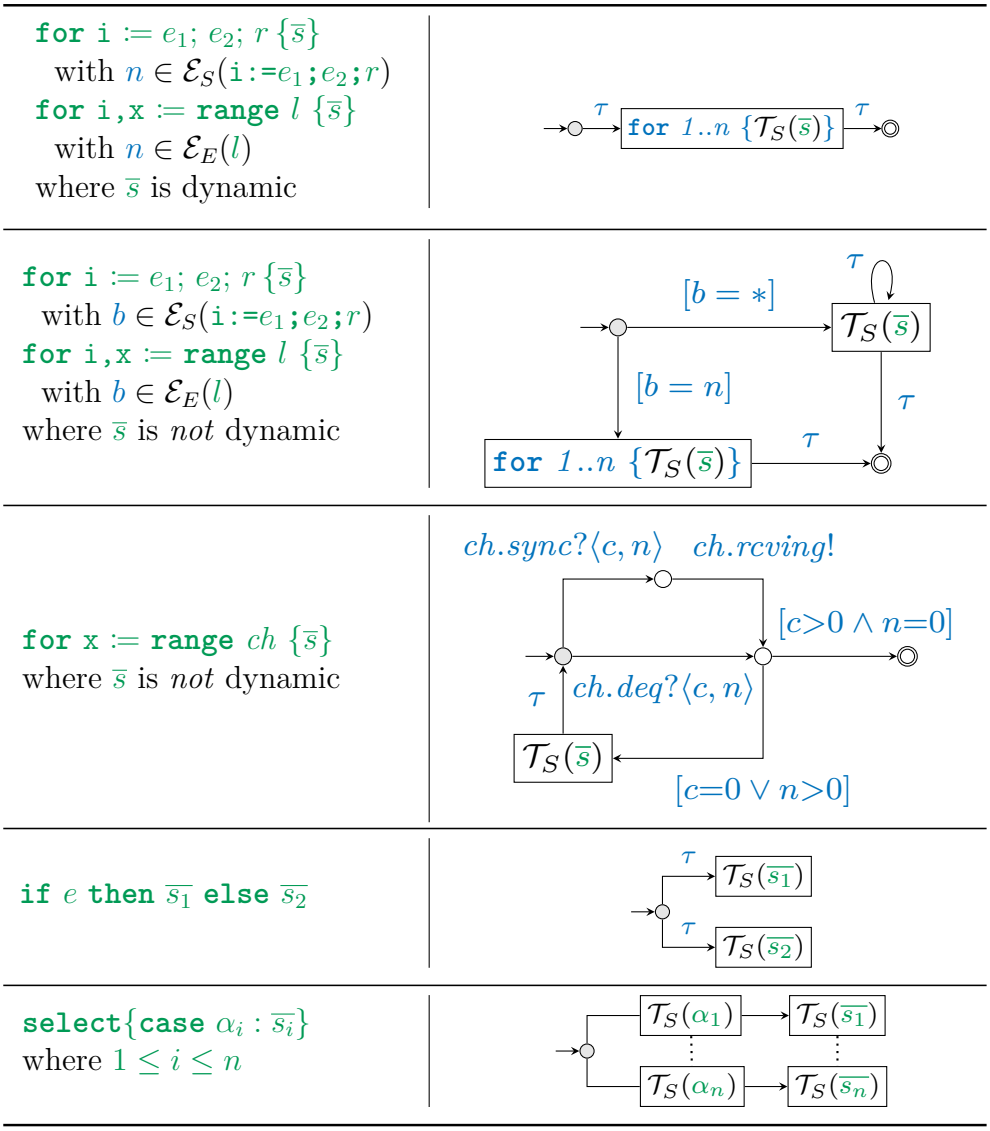


Figure 33: Overview of the translation of control flow constructs with function $\mathcal{T}_S(s)$.

translated to a guarded choice between two branches that represent an operation on a synchronous channel, or an asynchronous one. In the synchronous case, the process synchronises directly with another by sending on the (synchronous) Promela channel $ch.sync$, then notifying the channel process over $ch.snding$. The process sends a pair $\langle 0, 0 \rangle$ over $ch.sync$ (where the first 0 means that the channel is not closed, and second is the number of messages stored on the channel). In the asynchronous case, the process sends a message to its corresponding channel process over channel $ch.enq$. Notice that in both cases, we abstract away from the data sent over the channel (expression e is not modelled).

A receive operation ($\leftarrow ch$) is modelled in the dual way. The process executing the receive operation may either receive a pair $\langle c, n \rangle$ from another process over $ch.sync$ or from the channel process over $ch.deq$. We assume that variables c and n are fresh. They are unused for simple receive operations but are necessary when channels are ranged over.

Closing a channel (`close`(ch)) is translated as a Promela send operation over channel $ch.close$.

Go operations on waitgroups (wg.`Add`(e), wg.`Done`$()$, wg.`Wait`$()$) are translated to send and receive operations on the Promela channels of the corresponding waitgroup process (using channels $wg.update$ or $wg.wait$). Note that wg.`Add`(e) may increment or decrement the waitgroup counter by (the evaluation of) e, hence it is necessary to translate e to Promela. Function $\mathcal{T}_E(e)$ is a *partial* function from Go expressions to Promela expressions (when e cannot be translated, it aborts).

The translation of Go operations on mutexes to Promela is straightforward. Each operation is mapped to a send on the corresponding Promela channel of the mutex process.

4.2.5 Control flow and branching constructs

The core idea behind our modelling technique is to generate models that can perform *at least* all concurrent actions that can be performed by the program. Hence, if the program performs a particular send, the model must be able to perform the send as well. In this section, we describe how each Go statement is over-approximated by the model.

Figure 33 gives an overview of the mapping from Go constructs to Promela constructs using a graphical notation. Traditional `for`-loops and loops over collections are translated differently depending on whether they are dynamic, see Section 4.2.1. When a loop is dynamic, its number of iterations must be known before checking the model (e.g., the bounds are constant or they are instantiated by some concurrency parameters). Hence such loops are translated to Promela's own (finite) iteration constructs. When a loop is not dynamic, then it behaves as a fixed loop (as above) or as a non-deterministic loop (i.e., a loop that executes an arbitrary number of times). To enable this behaviour, we use the special value $*$ to flag a concurrency parameter as unspecified. When a bound is unspecified, the number of iterations is non-deterministic; if it is provided at model checking time, a fixed `for`-loop is used.

A `for`-loop that ranges over a channel ch executes its body every time a message is received on ch (until the channel is empty and closed). To encode this behaviour in Promela, we use a similar technique to the translation of the receive operation (see the second line of Figure 32). After each receive operation the values of variables c and n are tested, $c = 1$ means that the channel is closed, while n is the number of messages held in it. Note that because such a loop might be executed an arbitrary number of times, they can be translated only when the loop is not dynamic (we abort otherwise).

Go conditionals are mapped to internal choices with two branches(i.e., a choice with two unguarded branches). Hence, while only one branch of a Go if-then-else

may be taken, both branches are fireable in its generated model. A `select` block is translated to a choice in Promela, where each branch is either unguarded or guarded by the translation of a send or receive operation, each branch leads to the translation of the bodies $\overline{s_i}$. For send and receive operations, we use the construction given in the first two lines of Figure 32. Any branch of the `select` block that is `default` or guarded by a timeout is mapped to an unguarded branch (τ). Hence, the `default` of a `select` is always fireable by the model.

As in the work by Lange et al. (2017) which this technique is inspired by, the translation of loops, if-then-else, and select statements *over*-approximates the behaviour of a Go program. Hence, since the models over-approximate the program, our abstraction technique is *sound* as proven by Theorem 5.1 (subject reduction) in Lange et al. (2017). However, since our approach adds support for communication parameters, which increases the precision of the models and, as a result, our technique is only sound with respect to the instantiated value of the communication parameters.

To formaly prove the soundness of our approach, we would have to revisit (Lange et al. 2017, Theorem 5.1), the MiGo language and the behavioural types introduced in Lange et al. (2017) to support communication parameters, wait-groups, mutexes (and their associated operations) and for range over channel.

As explained above, Gomela translates certain for-loop into deterministic for-loops (as opposed to Lange et al. (2017)) when they are dynamic or the communication paremeter is specified. Deterministic for-loop are missing from MiGo.

4.3 Verifying models

We describe our approach to verify the models generated from Go programs in Section 4.2. We break down this description in two steps: properties of valuated models and automated generation of valuations.

4.3.1 Properties of valuated models

Given a Go program P, we generate several models such that each model M has a (possibly empty) list of concurrency parameters that are either mandatory or optional. We say that a model M is *valuated* if all its mandatory parameters are instantiated by values in $\mathbb{N}$ and all its optional parameters are instantiated by values in $\{*\} \cup \mathbb{N}$. Recall that setting a concurrency parameter to $*$ allows some loops to iterate an arbitrary number of times, see Figure 33.

Given a model M with mandatory parameters $\overline{x}$ and optional parameters $\overline{y}$, and vectors $\overline{u} \in \mathbb{N}^{|\overline{x}|}$ and $\overline{v} \in (\{*\} \cup \mathbb{N})^{|\overline{y}|}$. We write $M[\overline{x} := \overline{u}, \overline{y} := \overline{v}]$ for the valuation of M where each x_i (resp. y_i) is replaced by value u_i (resp. v_i).

Properties. We consider four properties, corresponding to the types of errors discussed in Section 2.1.4. All of these properties are either specified as Promela processes not reaching their end states, or reaching an error state. Assume M is a valuated model, we define the following properties:

1. $M \models \phi_{\text{MD}}$ (model deadlock) holds whenever no execution in M leads to a situation where *all* goroutine processes are stuck. Since several models may be extracted from a given program, ϕ_{MD} can identify some partial deadlocks in the source program.

2. $M \models \phi_{\text{CS}}$ (channel safety) holds whenever no execution in M leads to a situation where a channel process reaches its error state (i.e., sending on/closing a closed channel).

3. $M \models \phi_{\text{WS}}$ (waitgroup safety) holds whenever no execution in M leads to a situation where a waitgroup process reaches its error state (i.e., the waitgroup reaches a negative number).

4. $M \models \phi_{\text{MS}}$(mutex safety) holds whenever no execution in M leads to a situation where a mutex process reaches its error state (i.e., an unlocking

function *verify*(S, ϕ, M)
  $\overline{x} \leftarrow mandatory(M)$
  $\overline{y} \leftarrow optional(M)$
  $V \leftarrow \{(\overline{u}, \overline{v}) \mid \overline{u} \in S^{|\overline{x}|},\ \overline{v} \in (\{*\} \cup S)^{|\overline{y}|}\}$
  while $V \neq \emptyset$ **do**
    $(\overline{u}, \overline{v}) \leftarrow pickMax(V)$
    $b \leftarrow M[\overline{x} := \overline{u}, \overline{y} := \overline{v}] \models \phi$
    if $b \ \vee\ * \notin \overline{v}$ **then**
      $\Delta(\overline{u} \cdot \overline{v}) \leftarrow b$
      $V \leftarrow V \setminus \{(\overline{u}, \overline{w}) \in V \mid \overline{v} \succeq \overline{w}\}$
    otherwise
      $V \leftarrow V \setminus \{(\overline{u}, \overline{v})\}$
  return Δ

Algorithm 1: Verification of model M with property ϕ and values S. Auxiliary function $mandatory(M)$ (resp. $optional(M)$) computes the mandatory (resp. optional) concurrency parameters of M. Function $pickMax(V)$ returns a maximal element of set V wrt. $\succeq$.

operation is invoked on an unlocked mutex).

We aim for a sound verification approach, i.e., any behaviour of the source program can be simulated by the extracted model (assuming a precise valuation). Since each Go entry-point function P is over-approximated by its model, for any of the properties ϕ above, we should have that $M \models \phi$ implies that $P \models \phi$, whenever the parameters of P and M are instantiated to the same values. The reverse implication does not hold, i.e., if $M \not\models \phi$ we cannot conclude that $P \not\models \phi$.

4.3.2 Automated generations of model valuations

Next, we present a technique to perform a bounded verification of a parameterised model (up-to a finite set of possible values). Hereafter we assume a set $S \in \mathbb{N}$ given by the user, from which values of concurrency parameters are selected.

We define a partial ordering $\succeq$ on the set $(\{*\} \cup S)^k$ (with $k \in \mathbb{N}$) to identify a valuation that subsumes another.

$$(v_1, \ldots, v_k) \ \succeq\ (u_1, \ldots, u_k) \iff \forall 1 \leq i \leq k :\ v_i \in \{u_i, *\}$$

Algorithm 1 describes our approach to verify a model M for a property ϕ (e.g., absence of deadlock), wrt. values in S. The algorithm returns a map Δ that records the result of the verification by mapping valuations to Booleans. For instance if $\Delta(1,2) = \top$ for a model M with concurrency parameters x_1 and x_2, then property ϕ holds for $M[x_1 \coloneqq 1, x_2 \coloneqq 2]$.

Algorithm 1 starts by computing the list of mandatory and optional parameters ($\overline{x}$ and $\overline{y}$ respectively). Then it computes the set V of all possible valuations for these parameters (only optional parameters may be set to $*$). The algorithm then repetitively checks the property ϕ on model M where the parameters are instantiated with a maximal element from V (wrt. the $\succeq$-ordering). After each verification the set V is updated, removing all valuations that are subsumed by the current valuation if it was successful, or the current valuation otherwise. We only record in Δ those verifications that are successful or that do not involve any optional parameter set to $*$. Indeed when a verification with a parameter y set to $*$ fails, it is likely to be a false alarm. This will be "compensated" by further verifications where y is instantiated to values in S. In contrast, if a verification with a parameter y set to $*$ succeeds, then fewer verifications will be performed (i.e., that verification subsumes all instantiations of y).

Note that if M does not contain any parameter ($\overline{x} = \overline{y} = \epsilon$) then V is initialised to $\{\epsilon\}$, i.e., the singleton set containing the empty vector.

For each map Δ obtained from Algorithm 1, we compute a score based on the ratio of failed verifications over the total number of recorded verifications, i.e.,

$$score(\Delta) = \frac{|\{\overline{v} \in dom(\Delta) \mid \Delta(\overline{v}) = \bot\}|}{|dom(\Delta)|}$$

We use this scoring mechanism later on in Section 5.3 to give an aggregate result to reflect how many valuations of a particular model contained a bug.

Example 8. Given the program in Figure 22, our translation will generate

one model with three mandatory parameters (corresponding to |`trees`|, `m`, and `runtime.NumCPU()`). Assuming we are checking for ϕ_{MD} and we set $S = \{0, 1, 2, 3\}$, Algorithm 1 will perform 4^3 verifications, 45 of which are successful ($\top$). Hence we obtain a score of $45/64 \simeq 0.7$.

Example 9. Consider the (correct) program in Figure 34, which consists of two threads that exchange `x` messages over channel `a` (the value of `x` is unknown at compile-time). Our approach generates a unique model M for it, with one parameter x. Assuming we are checking for ϕ_{MD} and we set $S = \{0, 1, 2, 3\}$, Algorithm 1 performs five verifications, one for each element in $\{*\} \cup S$. The case where $x := *$ fails since both `for`-loops are modelled as loops that can terminate (independently) after an arbitrary number of iterations. Hence, that valuated model contains executions that lead to a deadlock (where either loops are waiting for a send or receive). However, when x is given a concrete value both loops iterate the same number of times, and thus each send/receive action is matched.

Thus we obtain a map Δ containing four elements, s.t. $\Delta(i) = \top$ for $0 \leq i \leq 3$. Hence, we have $score(\Delta) = 0/4 = 0$, i.e., no recorded verification failed.

Example 10. Now consider the buggy program in Figure 27, which consists of `numWorkers` + 1 (including `main()`) threads. The program contains two bugs based on the values of `numWorkers` and `numResponses`. The program will deadlock when `numWorkers` < `numResponses` because the number of sends, at line 18, on channel `a` is smaller than the number of receives at line 11 which will block the `main` thread. The program will also lead to a channel safety error when `numResponses` < `numWorkers`. Due to the number of receives being smaller than the number of sends, channel `a` will be closed at line 14 which will throw a run-time error when the next `worker` tries to send over channel `a` after it has been closed. Our approach generates a unique model M for it, with two parameters $numWorkers, numResponses$. `numWorkers` is mandatory because it is used as the

```
1  func sender(a chan int, x int) {
2    for i := 0; i < x; i++ { // sends x times
3      a ← i
4    }
5  }
6  func receiver(a chan int, x int) {
7    for i := 0; i < x; i++ { // receives x times
8      ←a
9    }
10 }
11 func main() {
12   x, _ := strconv.Atoi(os.Args[1])
13   a := make(chan int)
14   go sender(a, x)
15   receiver(a, x)
16 }
```

Figure 34: Program with an optional concurrency parameter.

bound of a dynamic `for`-loop at line 7. While, `numResponses` is an optionnal parameter because it is used as the bound of the non-dynamic for loop at line 10. Assuming we are checking for $\phi_{\text{MD}} \wedge \phi_{\text{CS}}$ and we set $S = \{0, 1, 2, 3\}$, Algorithm 1 performs 20 verifications, one for each element in the set $S \times (\{*\} \cup S)$. All of the valuations where $numResponses$ is set to $*$ will lead to model deadlocks because the receiving `for`-loop at line 10 will be modelled as a non-deterministic loop. When a valuation fails and contains at least one optional parameter set to $*$, GOMELA investigates it further by valuating the optionnal parameters of the model, in this case $numResponses$, with each values in S. Hence arriving at a total of 4×5 valuations. In this case only the 4 valuations where $numWorkers = numResponses$ will succeed. Hence, we have $score(\Delta) = {}^{16}/_{20} = 0.8$, i.e., 16 verifications out of 20 failed.

4.4 Implementation

We have implemented our approach in a tool called GOMELA which extracts models from Go code, and uses SPIN as a backend to automatically verify models up-to user-provided bounds. As shown in Figure 35, the workflow of GOMELA is

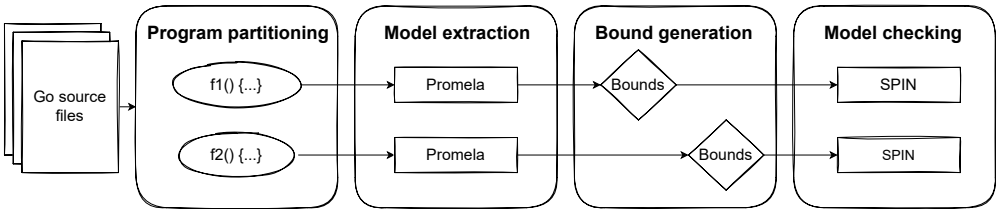


Figure 35: Workflow of GOMELA.

divided into 3 core phases:

1. **Program partitioning:** Go programs are partitioned into independent components using 2 steps:

   (a) AST generation: Similarly to **`GoSurvey`**, the toolchain developed for the empirical analysis described in Section 3.1, GOMELA relies on Go's `go/ast`, `go/parser` and `go/types` libraries for the front-end parsing and analyses a Go codebase package by package. Unlike Liu et al. (2021); Gabet and Yoshida (2020), our analysis is done on the surface language, rather than its lower-level representation (SSA).

   (b) Program partitioning: The AST generated is partitioned into the function declarations in the program which do not take concurrency primitives as parameters nor return any concurrency primitives, see Section 4.2.

2. **Model extraction:** Promela models are extracted from each partition in 3 steps as described in Section 4.2:

   (a) Communication parameter extraction: The AST extracted from each partition is traversed to extract the list of communication parameters inter-procedurally.

   (b) Model extraction: a Promela model is extracted from the body of the function declarations inter-procedurally.

(c) Generation of a complete model: GOMELA places the body extracted in the previous step inside an `init` proctype and the definition of the primitive processes used in the model. i.e, if a channel process is declared in the model, its definition needs to be provided. This is crucial because SPIN will return an error if an unused definition is provided.

(d) Removing unnecessary constructs: We reduce the complexity of the model generated by removing loops which do not contain any concurrency operations (such as receives, `Lock()`, etc), we assume that such loops eventually terminate. In addition, models that do not create any concurrency primitives are deleted for obvious reasons.

3. **Bound generation:** The tool analyses the complete model to find the occurrences of communication parameters in the model and computes a set of several models by valuating the mandatory parameters based on user-provided values as explained in Section 4.3.1.

4. **Model checking:** The Promela models extracted are verified in 2 steps:

   (a) Verification of each valuations: Each valuation is fed to SPIN, which verifies that the model is free from all properties described in Section 4.3.1.

   (b) Calculating the score: A score is generated as described in Section 4.3.2.

Gomela implementation's design. One of the aims when designing GOMELA was to *ease* the support of more Go constructs and additional concurrency primitives. Hence, GOMELA was designed to be *modular*. GOMELA generates an abstract syntax tree from a Go program using Go's parsing library and inductively goes through each node in the AST, as described in Figure 33 and Figure 32, and invokes specific functions based on the type of the node, which all return Promela code. This is better known as the *visitor* pattern. The concatenation of these

Promela code segments generates a complete Promela model. Adding supports for new Go constructs requires either to add specific functions for said constructs (such as linked lists) or to modify an existing function to accommodate the new concurrency primitive (such as `sync.Cond`, another popular concurrency primitive). As an example, adding support for waitgroups and mutexes in GOMELA was done particularly quickly (less than a week of work per feature by one developer). While adding support for `break` statement was implemented in a day's worth of work.

4.4.1 Supporting advanced language constructs

In addition to the language constructs listed in Figure 1, our tool handles structures, methods, anonymous functions, `break` and `defer` statements (when they do not occur inside conditionals/loops).

Supporting structures. Go structures that contain concurrency primitives are analysed inter-procedurally and flattened into a concatenation of their primitive names. This means that a channel `ch` initialised as a field of a structure `a` (`a.ch = make(chan T,e)`) will be translated to a Promela channel primitive named a_ch. Similarly, fields of structs that are used as communication parameters are treated in the same way.

Figure 36 shows a Go program (top) which declares an asynchronous channel of size `s.numWorkers` and the Promela model (bottom) extracted from the program. The declaration of channel `s.ch` is translated to a declaration of a Promela channel primitive process named s_ch where its capacity `s.numWorkers` will be set to s_numWorkers. The first two statements from the Go program are simply ignored because they do not contain any concurrency constructs. The channel declaration, at the last line, is translated to a channel primitive structure called a Chandef where depending on its capacity, s_numWorker, will either be spawned as an

```
s := Cluster{}
s.numWorkers = getNumOfWorker()
s.ch = make(chan int, s.numWorkers)
```

```
Chandef s_ch;
int s_numWorker = getNumOfWorker
if
:: s_numWorker > 0 →
    s_ch.size = s_numWorker;
    run AsyncChan(s_ch);
:: else →
    run SyncChan(s_ch);
if;
```

Figure 36: Struct fields in Go (top) and its model in Promela (bottom).

AsyncChan or a SyncChan process (state `A` or `S` from Figure 28 respectively). Here s_numWorker is used to set the size of the channel and, therefore, is a mandatory parameter of the model. This means that at verification time getNumOfWorker will have to be given an actual value.

Supporting methods. A method `m` on struct `S`, `func` $(x\ \mathtt{S})\ \mathtt{m}(\overline{y\ \mathtt{T}})\ \mathtt{T}\ \{\overline{s}\}$, is processed in two steps. First, they are normalised into a function named `S_m` whose parameters are the conjunction of the primitives contained in receiver x and the parameters $\overline{y}$ of method `m`. Second, the declaration of `S_m` is dealt with like a normal function declaration, and each call site is normalised in a consistent way. GOMELA aborts when it encounters virtual method calls (when the method parameters include concurrency primitives).

Supporting anonymous functions. Anonymous functions are normalised to (freshly) named functions, after computing their closures (limited to concurrency primitives and concurrency parameters). Hence they are dealt with like other Go functions.

Supporting `return` and `defer` statements. A `defer` f statement in Go specifies that function f has to be called right before returning from the function. To

support `defer` statements, our models uses a stack where function calls are popped and pushed using a combination of Promela labels and `goto` statements. Our approach inserts a proc_end: label at the end of each function process (including `init`) in the model which represent the bottom of the stack. The translation of each f in each `defer` f statement in the program will be stacked over their respective proc_end: label. Hence, all `return` statements in the program can be translated as a `goto` statement which redirects to the top of the stack.

Supporting `break` and `continue` statements. To support `break` statements and `continue` statements, each loop is given a starting and an ending label. As a result, each `continue` and `break` statement is translated as a `goto` to the starting and ending label of the most inner loop.

Unsupported constructs. We do not support dynamic data-structures (e.g., linked lists). If a partition contains a list, array or map typed as a concurrency primitive, our approach aborts, reports that the partition cannot be verified and proceeds with the rest of the program. Our approach aborts because having concurrency primitives contained in a (potentially infinite) list can lead to an infinite number of concurrency primitive which we cannot generate a finite state model out of. In addition, our approach does not support struct embedding. In Go, structs can be extended from existing structs (close to inheritance in object oriented programming languages), however, the implementation of the existing structs might be declared in a third-party library.

4.5 Limitations of our modelling approach

Key limitations need to be tackled to address the full Go language. We assume that variables are immutable, as a consequence we cannot soundly analyse programs that, e.g., mutate a list `files` in between using `len(files)` as a

communication-related parameter. This is due to the fact that two unknown communication parameters `len(files)` in the same scope will be given the same user-defined Promela variable. Go has object-oriented-like features, such as structs, methods, and interfaces. Although our approach supports structures and methods, it does not support structs embedding and interfaces. As the previous works shown in Table 8, we do not support channel passing (since we abstract away the data sent over channels), nor concurrency primitive stored in dynamic structures (such as linked lists or maps), nor channels created inside `for`-loops. We note that our empirical analysis from Chapter 3 found that only 11% of projects used channels that carry channels, 8% used slice to store channels and 22% had channel creations inside `for`-loops.

4.6 Conclusion

We described a novel approach to verifying Go programs using bounded model checking of parameterised behavioural types. Our work builds on the approach in Lange et al. (2018) and improves it to support statically unknown communication-related parameters via a bounded analysis. Our approach allows us to support programs that spawn a parameterised number of goroutines or channel capacities. In addition, our approach works on the surface language (instead of its SSA representation) and extract the AST of Go programs using Go's own parsing libraries, which means that it is easily upgradable to support additional constructs and should continue to work with newer versions of the language.

The aim of our approach is to be sound, as described in Section 4.2.5 and 4.3.1, this is achieved by over-approximating the behaviours of the program. Note that our approach is only sound with respect to what it supports and to the actual values set as the communication parameters (if there are any). In the next chapter, we devised a set of benchmarks which showcases the soundness, applicability and

scalability of our approach.

Chapter 5

Empirical Evaluation of Gomela

In this chapter, we conduct an empirical evaluation of Gomela on three sets of benchmarks which evaluate the functionalities supported, the applicability and the scalability of Gomela. We structure the evaluation of our approach into three research questions that aim at evaluating the real-world usability of our approach.

RQ1: *How do Gomela's functionalities compare to the state-of-the-art?* To answer this question we have generated a set of 220 buggy Go programs which is aimed at evaluating which concurrent operations and Go features the tool supports. We compare our tool with GCatch and Godel2.

RQ2: *How applicable Gomela is to real- world programs?* To answer this question we have created and evaluated our tool on a set of real-world inspired buggy programs to determine if our approach is applicable to real-world usage. We also compare the results of our tool against GCatch and Godel2.

RQ3: *How does Gomela scale to real-world programs?* To answer this question we fed Gomela with the list of 150 Go projects devised in our empirical analysis discussed in Chapter 3. We measure the number/parameters of models generated and the verification runtimes. We also report on the error scores obtained for four

properties.

Contributions. In this chapter, we devised a set of benchmarks which showcases all possible safety errors and blocking operations that could arise from the use of channels, waitgroups and mutexes in Go. We performed an evaluation and comparison of GOMELA with two other state-of-the-art static checkers on this set of benchmarks. We applied GOMELA on 125 Github projects to assess its scalability.

5.1 RQ1: How do Gomela's functionalities compare to the state-of-the-art?

In this section, we describe the set of benchmarks that we have devised to test and compare the functionalities of our tool with state-of-the-art static checkers, GCatch (Liu et al. (2021)) and Godel2 (Gabet and Yoshida (2020)). In addition, this set of benchmarks can also be used by other researchers and practitioners to test that their tool supports a large subset of the Go languages as well as different concurrency primitives.

Overall, we have devised a set of 220 buggy benchmarks that determines which safety and blocking errors can be verified by the tool as well as which Go constructs are supported by the tool. This set was generated in two steps. First, by generating a list of simple buggy code snippets for each possible bug that can arise through the use of each concurrency operations and secondly by introducing them into code contexts that contained particular Go constructs (such as `for`-loops or `defer` statements).

Ref.	Declaration (CP)	Name ($\widehat{CP}$)	Parameter form ($\langle CP \rangle$)
ch	`ch := make(chan int)`	`ch`	`ch chan int`
wg	`var wg sync.WaitGroup`	`wg`	`wg sync.WaitGroup`
mu	`var mu sync.Mutex`	`mu`	`mu sync.Mutex`
rwmu	`var rwmu sync.RWMutex`	`rwmu`	`rwmu sync.RWMutex`

Figure 37: Declaration and usage of concurrency primitives.

	CP	Snippet name	Code snippet (CS)	Description
○	ch	*blocking-send*	ch←0	Blocking send.
•	ch	*send-close*	`close`(ch);ch ← 0	Send on a closed channel.
○	ch	*blocking-rcv*	←ch	Blocking receive.
•	ch	*double-close*	`close`(ch);`close`(ch)	Closing a closed channel.
○	ch	*range*	`for range` ch {}	Blocking `range`.
○	ch	*select*	`select` {`case` ←ch: }	Blocking `select` statement.
○	wg	*blocking-wait*	`wg.Add(1);wg.Wait()`	Inifinite `Wait()`.
•	wg	*negative-add*	`wg.Add(-1)`	Negative counter error.
•	wg	*negative-done*	`wg.Done()`	Negative counter error.
○	(rw)mu	*blocking-lock*	`go` $func()${`mu.Lock()};`mu.Lock()`	Blocking lock.
○	(rw)mu	*double-lock*	`mu.Lock();mu.Lock()`	Locking own (rw)mutex.
•	(rw)mu	*unlock-unlocked*	`mu.Unlock()`	Unlocking an unlocked (rw)mutex.
○	rwmu	*double-rlock*	`mu.RLock();mu.Lock()`	Locking own rwmutex.
•	rwmu	*runlock-unlocked*	`mu.RUnlock()`	RUnlocking an unlocked rwmutex.

Figure 38: Code snippets that contain a bug for each bug that can arise through the use of each concurrent operation. Some are blocking(○) or may trigger a runtime error (•), see Figure 1.

5.1.1 A Comprehensive Set of Concurrency Bugs

To compare the functionalities of Go static-checkers, we selected a set of synthetic programs which are comprised of three components: A concurrency primitive declaration, a code snippet and a parameterised context which can take additional parameters such as bounds.

The four different concurrency primitive declarations (*CP*) and their usages are shown in Figure 37. These are the declaration of channels, waitgroups, mutexes and rwmutex. The name and parameter forms are used in the contexts when the concurrency primitive needs to be passed as an argument of a function call or as a parameter of a function declaration respectively. The code snippets (*CS*) are aimed at being the minimal possible code elements in terms of the number of operations that generate one of all possible concurrent bugs from Figure 1 (shown as ∘ and •) excluding data-races. We assume that no other concurrency operations are performed on the concurrency primitive used in the snippet outside of the code snippet itself. All of the 14 code snippets are shown in Figure 38. The column *CP* (Concurrency Primitive) shows the type of the concurrency primitive that needs to be declared for the code snippet to be valid. The declaration of the concurrency primitive is external to the code snippet, because as we will see later, the context will place the concurrency primitive declaration in various Go features to test whether such features are supported by the tool.

- *blocking-send* and *send-close* are the two different bugs that can arise from using a send statement. The send statement will block if a channel is full and there is no receiver or will generate a runtime error if it is performed on a previously closed channel, i.e., `"panic: send on closed channel"`..
- *blocking-rcv* is the dual of *blocking-send*, i.e., a blocking bug where the executing thread is stuck waiting for a matching send.
- *double-close* triggers a runtime error by closing a closed channel, i.e., `"panic:`

`close of closed channel"`.

- *range* contains a blocked `for range` loop that receives on a channel that is never closed.
- *select* contains a blocked `select` statement where all branches are stuck.
- *blocking-wait* contains a waitgroup that is incremented and then waited upon without being decremented which leads to a deadlock.
- *negative-add* and *negative-done* lead to runtime errors due to decrementing the counter of a waitgroup to a negative value, i.e., `"panic: sync: negative WaitGroup counter"`.
- *blocking-lock* leads to a blocking bug where one thread permanently waits for the lock. The program may terminate only if the main thread acquires the lock. This snippet targets both mutex and rwmutex.
- *double-lock* leads to a deadlock due to a mutex being locked twice by the same goroutine. This snippet targets both mutex and rwmutex.
- *unlock-unlocked* leads to a runtime error due to unlocking an unlocked mutex, i.e., `"fatal error: sync: unlock of unlocked mutex"`
- *double-rlock* leads to a deadlock because a rwmutex is read-locked and locked sequentially by the same goroutine.
- *runlock-unlocked* leads to a runtime error due to read unlocking an unlocked rwmutex, i.e., `"fatal error: sync: RUnlock of unlocked RWMutex"`.

Each code snippet listed in Figure 38 and their respective target primitive *CP* (using the standard mutex when a choice was available) have been instantiated into the *minimal* context (shown in Figure 39). The resulting set of benchmarks has been fed to all three tools and their results are shown in Table 13. We use ✗

Table 13: Listing of all buggy snippets (*CS*) integrated into the *minimal* context with their respective communication primitive declared (*CP*). ✗ means false negative, ✓ means the bug was found (true positive). GCatch falsely reports a blocking bug even though the program leads to a runtime error (shown as ✗[1]).

CP	Snippet	GOMELA	GCatch	Godel2
ch	*blocking-rcv*	✓	✓	✓
ch	*blocking-send*	✓	✓	✓
ch	*send-close*	✓	✗[1]	✓
ch	*double-close*	✓	✗	✓
ch	*blocking-select*	✓	✓	✓
ch	*range*	✓	✗	✓
wg	*blocking-wait*	✓	✗	✗
wg	*negative-add*	✓	✗	✗
wg	*negative-done*	✓	✗	✗
(rw)mu	*blocking-lock*	✓	✓	✓
(rw)mu	*double-lock*	✓	✓	✓
(rw)mu	*unlock-unlocked*	✓	✗	✓
rwmu	*double-rlock*	✓	✓	✓
rwmu	*runlock-unlocked*	✓	✗	✓

to show that the tool missed the bug (false negative) while we use ✓ to show that the tool found the bug (true positive). We use † to show that the tool explicitly reported that it aborted, i.e, it did not support the program. GCatch reports a false positive for the *send-close* snippet by identifying a blocking bug when the actual bug is a violation of a safety property (shown as ✗[1]).

From Table 13, we can see that GOMELA finds the bug in each test cases while GCatch and Godel2 do not support any waitgroup operation and miss bugs on multiple benchmarks. We have found that GCatch reports a blocking bug with *send-close* even though the benchmark contains a runtime error and simply misses the runtime error in *double-close*. This is not surprising as Liu et al. (2021) state that they focus on blocking bugs (blocking misuse-of-channels). However, GCatch fails to catch the blocking bug that arises from having a blocked `range`

minimal(*CP*, *CS*)	`func main() {`
	`  CP`
	`  CS`
	`}`

Figure 39: The *minimal* context is used as a sanity check to see which type of concurrency bugs are supported by the verification tool.

over a channel. GCatch also misses the bug in *unlock-unlocked* and *runlock-unlocked*. Godel2 finds all channel and mutex related bugs but does not support waitgroups.

5.1.2 Instantiating parameterised contexts

The benchmark suite is made by instantiating programs with holes, that we call *parameterised contexts*. A parameterised context `contextname`$(CP, CS, b)\{S\}$ is a function named `contextname` that takes a concurrency primitive declaration CP, a code snippet CS, and a bound b as parameters, and uses these parameters in a non-buggy program S to return a buggy program.

A concurrency primitive creation CP declares and initiliases a concurrency primitive. Bound b is an integer that is used as the bound of a communication parameter in the program.

Composition of a list of parameterised contexts. There are two types of contexts, *non-restrictive* contexts that can take code snippets of all types (channel, waitgroup and (rw)mutex) as parameter and *channel-specific* contexts which work only with channel code snippets. Here is the list of all non-restrictive contexts:

1. *minimal*: The *minimal* context, shown in Figure 39, acts as a sanity check to test which concurrency bugs, that can arise through the use of all basic concurrency operations, are supported. This context was used to produce the results of Table 13 discussed above in Section 5.1.1.

non-dynamic-for(*CP*, *CS*)	`func main() {`
	`  CP`
	
	`  for i := 0; i < b; i++ {`
	`  CS`
	`  }`
	`}`

Figure 40: The *non-dynamic-for* context tests whether non-dynamic `for`-loops are supported.

dynamic-for(*CP*, *CS*, *b*)	`func main() {`
	`  CP`
	
	`  for i := 0; i < b; i++{`
	`  go func() {`
	`  CS`
	`  }()`
	`  }`
	`}`

Figure 41: The *dynamic-for* context tests whether dynamic `for`-loops are supported.

primitive-for(*CP*, *CS*, *b*)	`func main() {`
	
	`  for i := 0; i < b; i++ {`
	`  CP`
	`  CS`
	`  }`
	`}`

Figure 42: The *primitive-for* context tests whether concurrency primitives declared inside `for`-loops are supported.

defer(*CP*, *CS*)	`func main() {`
	`  CP`
	
	`  defer func() {`
	`  CS`
	`  }()`
	`}`

Figure 43: The *defer* context tests whether communication operations within `defer` statements are supported.

$closure(CP,\ CS)$	`func main(){`
	`  CP`
	
	`  apply(func() {`
	`  CS`
	`  })`
	`}`
	
	`func apply(f func) {`
	`  f()`
	`}`

Figure 44: The *closure* context tests whether closures are supported.

$recursion(CP,\ CS)$	`func main() {`
	`  CP`
	`  rec(`$\widehat{CP}$`,10)`
	`}`
	
	`func rec(`$\langle CP \rangle$`,i int){`
	`  if i > 0 {`
	`  CS`
	`  rec(`$\widehat{CP}$`,i-1)`
	`  }`
	`}`

Figure 45: The *recursion* context tests whether recursion is supported.

$timeout(CP,\ CS)$	`func main() {`
	`  CP`
	
	`  select {`
	`  case <-time.After(3 * time.Second):`
	`  CS`
	`  }`
	`}`

Figure 46: The *timeout* context tests whether timeouts are supported.

2-branch-select(*CP*, *CS*)

```
func main() {
  ch2 := make(chan int)
  ch1 := make(chan int)

  CP
  go func() {
    select {
    case <-ch1:
    case <-ch2:
      CS
    }
  }()

  ch2 <- 0
}
```

Figure 47: The *2-branch-select* context tests whether `select` statements with two branches are supported.

interface(*CP*, *CS*)

```
type I interface {
  f()
}

type A struct {
}

func (a A) f(⟨CP⟩) {
  CS
}

type B struct {
}

func (b B) f(⟨CP⟩) {

}

func main() {

  var v I

  if len(os.Args) > 0 {
    v = A{}
  } else {
    v = B{}
  }
  CP
  v.f(ĈP)
}
```

Figure 48: The *interface* context tests whether interfaces are supported.

2. *non-dynamic-for*: A context, shown in Figure 40, that tests which of all concurrency operations are supported inside `for`-loops.

3. *dynamic-for*: A context, shown in Figure 41, that tests if goroutines spawned inside a `for`-loop are supported. Recall that in Table 10 from our empirical analysis, we found that 78% of projects contained an unbounded dynamic `for`-loop while 49% of projects had a bounded dynamic `for`-loop.

4. *primitive-for*: A context, shown in Figure 42, that tests if concurrency primitives declared inside a `for`-loop are supported.

5. *defer*: A context, shown in Figure 43, that tests if concurrency operations are supported inside `defer` statements.

6. *closure*: Figure 44 shows a context that tests whether closures (high order functions) are supported.

7. *recursion*: Figure 45 shows a context that tests whether recursion is supported. The number of iterations has been set arbitrarily to 10.

8. *timeout*: We have seen in Section 3.2.1 that Go's `timer` library was frequently used to implement timeouts in Go projects. Such constructs are generally used in a `select` statement to wait for a specific amount of time before triggering the timeout branch. Figure 46 shows a context that tests whether timeouts are supported.

9. *2-branch-select*: This context, shown in Figure 47 is used to check if the tool supports a `select` statement with multiple branches.

10. *interface*: Figure 48 shows a context that tests whether interfaces are supported.

```
1  func main() {
2
3    ch := make(chan int, 1)
4
5    ch <- 0
6    <-ch
7
8    CS
9
10 }
```

```
1  func main() {
2
3    ch := make(chan int, 4)
4
5    ch <- 0
6    ch <- 0
7    ch <- 0
8    ch <- 0
9    <-ch
10   <-ch
11   <-ch
12   <-ch
13
14   CS
15
16 }
```

Figure 49: A context that tests whether asynchronous channels of size 1 (left) and 4 (right) are supported.

Channel-specific contexts. There are 2 channel-specific contexts : *async-channel-1* and *async-channel-4*. Both contexts can be seen in Figure 49. These contexts are used to test whether asynchronous channels are supported. to determine the size of the channels, we used results from Section 3.2.3. We used a size of 1 and 4 because they are the bound given to the majority of the channels that we analysed in our empirical analysis (the mean and Q3 respectively). The max value has not been included because the max value is 100000. This would generate an overly complicated context (a sequence of 100000 send and receives) while the aim of this set of benchmarks is to test the functionalities supported by the tool instead of its scalability.

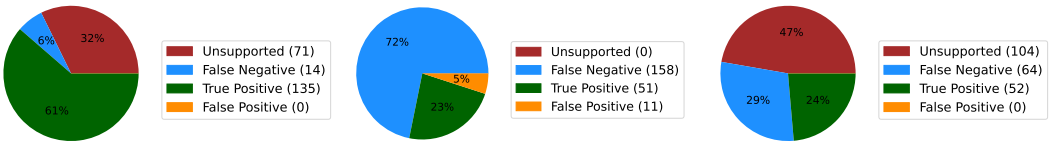
5.1.3 Results for 220 Buggy Programs

In this section, we describe the results of evaluating GOMELA, GCatch and Godel2 on the 220 benchmarks generated by instantiating each parameterised contexts with all code snippets listed in Figure 38 with their respective communication primitive.

Figure 50 shows the results of evaluating each individual tool on this set of

Table 14: Verification of all contexts instantiated with an empty primitive declaration and code snippet with GOMELA, GCatch and Godel2. ⊖ means *true negative*, i.e, the tool correctly reported that there was no bug. † shows that the tool reported that the benchmark was not supported.

Context	GOMELA	GCatch	Godel2
minimal(ϵ, ϵ)	⊖	⊖	⊖
non-dynamic-for$(\epsilon, \epsilon, 10)$	⊖	⊖	⊖
non-dynamic-for$(\epsilon, \epsilon,$`len(os.Args)`$)$	⊖	⊖	⊖
dynamic-for$(\epsilon, \epsilon, 10)$	⊖	⊖	⊖
dynamic-for$(\epsilon, \epsilon, 100)$	⊖	⊖	⊖
dynamic-for$(\epsilon, \epsilon, 120000)$	⊖	⊖	⊖
dynamic-for$(\epsilon, \epsilon,$`len(os.Args)`$)$	⊖	⊖	⊖
primitive-for$(\epsilon, \epsilon, 10)$	⊖	⊖	⊖
primitive-for$(\epsilon, \epsilon,$`len(os.Args)`$)$	⊖	⊖	⊖
defer(ϵ, ϵ)	⊖	⊖	⊖
closure(ϵ, ϵ)	⊖	⊖	†
recursion(ϵ, ϵ)	†	⊖	⊖
timeout(ϵ, ϵ)	⊖	⊖	⊖
2-branch-select(ϵ, ϵ)	⊖	⊖	†
interface(ϵ, ϵ)	†	⊖	†
async-chan-1(ϵ)	⊖	⊖	⊖
async-chan-4(ϵ)	⊖	⊖	⊖



(a) Gomela. **(b)** GCatch. **(c)** Godel2.

Figure 50: Summary of the evaluation on 220 benchmarks of all three tools.

Table 15: Results of verifying channel deadlock code snippets integrated into each context with *CP* = ch (Part I).

Snippet (*CS*)	Context	GOM.	GCatch	Godel2
	minimal(CS,CP)	✓	✓	✓
	non-dynamic-for(CS,CP,10 000)	✓	✗	✓
	non-dynamic-for(CS,CP,`len(os.Args)`*)*	✓	✗	✓
	dynamic-for(CS,CP,10)	✓	✗	†
	dynamic-for(CS,CP,100)	✓	✗	†
	dynamic-for(CS,CP,120 000)	†	✗	†
	dynamic-for(CS,CP,`len(os.Args)`*)*	✓	✗	†
blocking-send	*primitive-for(CS,CP,10)*	†	✗	†
	primitive-for(CS,CP,`len(os.Args)`*)*	†	✗	†
	defer(CS,CP)	✓	✓	✗
	closure(CS,CP)	✗	✗	†
	recursion(CS,CP)	†	✗	✓
	timeout(CS,CP)	✓	✓	✗
	2-branch-select(CS,CP)	✓	✓	†
	interface(CS,CP)	†	✗	†
	minimal(CS,CP)	✓	✓	✓
	non-dynamic-for(CS,CP,10 000)	✓	✗	✓
	non-dynamic-for(CS,CP,`len(os.Args)`*)*	✓	✗	✓
	dynamic-for(CS,CP,10)	✓	✗	†
	dynamic-for(CS,CP,100)	✓	✗	†
	dynamic-for(CS,CP,120 000)	†	✗	†
	dynamic-for(CS,CP,`len(os.Args)`*)*	✓	✗	†
	primitive-for(CS,CP,10)	†	✗	†
	primitive-for(CS,CP,`len(os.Args)`*)*	†	✗	†
blocking-rcv	*defer(CS,CP)*	✓	✓	✗
	closure(CS,CP)	✗	✗	†
	recursion(CS,CP)	†	✗	✓
	timeout(CS,CP)	✓	✓	✗
	2-branch-select(CS,CP)	✓	✓	†
	interface(CS,CP)	†	✗	†
	async-chan-1(CS)	✓	✗	✓
	async-chan-4(CS)	✓	✗	✓

Table 16: Results of verifying channel deadlock code snippets integrated into each context with CP = ch. GCatch erroneously reports a blocking error in the wrong place when *2-branch-select* is instantiated with the *range* code snippet (marked as ✗²) (Part II).

Snippet (*CS*)	Context	Gom.	GCatch	Godel2
	minimal(CS,CP)	✓	✓	✓
	non-dynamic-for(CS,CP,10 000)	✓	✗	✓
	non-dynamic-for(CS,CP,`len(os.Args)`*)*	✓	✗	✓
	dynamic-for(CS,CP,10)	✓	✗	†
	dynamic-for(CS,CP,100)	✓	✗	†
	dynamic-for(CS,CP,120 000)	†	✗	†
	dynamic-for(CS,CP,`len(os.Args)`*)*	✓	✗	†
	primitive-for(CS,CP,10)	†	✗	†
select	*primitive-for(CS,CP,*`len(os.Args)`*)*	†	✗	†
	defer(CS,CP)	✓	✓	✗
	closure(CS,CP)	✗	✗	†
	recursion(CS,CP)	†	✗	✓
	timeout(CS,CP)	✓	✓	✗
	2-branch-select(CS,CP)	✓	✓	†
	interface(CS,CP)	†	✗	†
	async-chan-1(CS,CP)	✓	✗	✓
	async-chan-4(CS,CP)	✓	✗	✓
	minimal(CS,CP)	✓	✗	✓
	non-dynamic-for(CS,CP,10 000)	✓	✗	✓
	non-dynamic-for(CS,CP,`len(os.Args)`*)*	✓	✗	✓
	dynamic-for(CS,CP,10)	✓	✗	†
	dynamic-for(CS,CP,100)	✓	✗	†
	dynamic-for(CS,CP,120 000)	†	✗	†
	dynamic-for(CS,CP,`len(os.Args)`*)*	✓	✗	†
	primitive-for(CS,CP,10)	†	✗	✓
range	*primitive-for(CS,CP,*`len(os.Args)`*)*	†	✗	✓
	defer(CS,CP)	✓	✗	✗
	closure(CS,CP)	✗	✗	†
	recursion(CS,CP)	†	✗	✓
	timeout(CS,CP)	✓	✗	✗
	2-branch-select(CS,CP)	✓	✗²	†
	interface(CS,CP)	†	✗	†
	async-chan-1(CS)	✓	✗	✓
	async-chan-4(CS)	✓	✗	✓

Table 17: Results of verifying channel safety error code snippets in each contexts with $CP = \text{ch}$.

Snippet (*CS*)	Context	Gom.	GCatch	Godel2
double-close	*minimal(CS,CP)*	✓	✗	✓
	non-dynamic-for(CS,CP,10 000)	✓	✗	✓
	non-dynamic-for(CS,CP,`len(os.Args)`)	✓	✗	✓
	dynamic-for(CS,CP,10)	✓	✗	†
	dynamic-for(CS,CP,100)	✓	✗	†
	dynamic-for(CS,CP,120 000)	†	✗	†
	dynamic-for(CS,CP,`len(os.Args)`)	✓	✗	†
	primitive-for(CS,CP,10)	†	✗	†
	primitive-for(CS,CP,`len(os.Args)`)	†	✗	†
	defer(CS,CP)	✓	✗	✗
	closure(CS,CP)	✗	✗	†
	recursion(CS,CP)	†	✗	✓
	timeout(CS,CP)	✓	✗	✗
	2-branch-select(CS,CP)	✓	✗[1]	†
	interface(CS,CP)	†	✗	†
	async-chan-1(CS)	✓	✗	✓
	async-chan-4(CS)	✓	✗	✓
send-close	*minimal(CS,CP)*	✓	✗[1]	✓
	non-dynamic-for(CS,CP,10 000)	✓	✗	✓
	non-dynamic-for(CS,CP,`len(os.Args)`)	✓	✗	✓
	dynamic-for(CS,CP,10)	✓	✗	†
	dynamic-for(CS,CP,100)	✓	✗	†
	dynamic-for(CS,CP,120 000)	†	✗	†
	dynamic-for(CS,CP,`len(os.Args)`)	✓	✗	†
	primitive-for(CS,CP,10)	†	✗	†
	primitive-for(CS,CP,`len(os.Args)`)	†	✗	†
	defer(CS,CP)	✓	✗[1]	✗
	closure(CS,CP)	✗	✗	†
	recursion(CS,CP)	†	✗	✓
	timeout(CS,CP)	✓	✗[1]	✗
	2-branch-select(CS,CP)	✓	✗[1]	†
	interface(CS,CP)	†	✗	†
	async-chan-1(CS)	✓	✗	✓
	async-chan-4(CS)	✓	✗	✓

Table 18: Results of verifying mutex code snippets with GOMELA, GCatch and Godel2 in each context with CP = mu (Part I).

Snippet (CS)	Context	GOM.	GCatch	Godel2
	minimal(CS,CP)	✓	✓	✓
	non-dynamic-for(CS,CP,10 000)	✓	✓	†
	non-dynamic-for(CS,CP,`len(os.Args)`*)*	✓	✓	†
	dynamic-for(CS,CP,10)	✓	✓	†
	dynamic-for(CS,CP,100)	†	✓	†
	dynamic-for(CS,CP,120 000)	†	✓	†
	dynamic-for(CS,CP,`len(os.Args)`*)*	✓	✓	†
blocking-lock	*primitive-for(CS,CP,10)*	†	✓	†
	primitive-for(CS,CP,`len(os.Args)`*)*	†	✓	†
	defer(CS,CP)	✓	✓	✗
	closure(CS,CP)	✗	✗	†
	recursion(CS,CP)	†	✓	✗
	timeout(CS,CP)	✓	✓	✗
	2-branch-select(CS,CP)	✓	✓	†
	interface(CS,CP)	†	✗	†
	minimal(CS,CP)	✓	✗	✓
	non-dynamic-for(CS,CP,10 000)	✓	✗	✓
	non-dynamic-for(CS,CP,`len(os.Args)`*)*	✓	✗	✓
	dynamic-for(CS,CP,10)	✓	✗	†
	dynamic-for(CS,CP,100)	✓	✗	†
	dynamic-for(CS,CP,120 000)	†	✗	†
	dynamic-for(CS,CP,`len(os.Args)`*)*	✓	✗	†
{*unlock*	*primitive-for(CS,CP,10)*	†	✗	✓
-unlocked}	*primitive-for(CS,CP,*`len(os.Args)`*)*	†	✗	✓
	defer(CS,CP)	✓	✗	✗
	closure(CS,CP)	✗	✗	†
	recursion(CS,CP)	†	✗	✗
	timeout(CS,CP)	✓	✗	✗
	2-branch-select(CS,CP)	✓	✗[1]	†
	interface(CS,CP)	†	✗	†

Table 19: Results of verifying mutex code snippets with GOMELA, GCatch and Godel2 in each context with CP = mu (Part II).

Snippet (CS)	Context	GOM.	GCatch	Godel2
	minimal(CS,CP)	✓	✓	✓
	non-dynamic-for(CS,CP,10 000)	✓	✓	✓
	non-dynamic-for(CS,CP,`len(os.Args)`*)*	✓	✓	✓
	dynamic-for(CS,CP,10)	✓	✓	†
	dynamic-for(CS,CP,100)	✓	✓	†
	dynamic-for(CS,CP,120 000)	†	✓	†
	dynamic-for(CS,CP,`len(os.Args)`*)*	✓	✓	†
double-lock	*primitive-for(CS,CP,10)*	†	✓	✓
	primitive-for(CS,CP,`len(os.Args)`*)*	†	✓	✓
	defer(CS,CP)	✓	✓	✗
	closure(CS,CP)	✗	✗	†
	recursion(CS,CP)	†	✓	✗
	timeout(CS,CP)	✓	✓	✗
	2-branch-select(CS,CP)	✓	✓	†
	interface(CS,CP)	†	✗	†

benchmarks. We can see that overall GOMELA found the majority of bugs (True positive) compared to Godel2 and GCatch which found 24% and 23% of bugs. GCatch supports all benchmarks compared to GOMELA and Godel which does not support 32% and 47% of benchmarks respectively. However, we can see that GCatch generates mostly false negatives.

Table 13 to Table 21 show the individual results of each tool on each of all 220 benchmarks. The first column shows which snippet the contexts have been instantiated with. The second column shows the name of the instantiated context and the bound used when required. Note that the CS and CP parameters have been omitted for clarity. The third, fourth and fifth columns show the results of GOMELA, GCatch and Godel2 respectively. We represent a bug found by the tool as ✓ (true positive) and ✗ when the tool missed the bug (false negative). We use $\ominus$ to show that the tool did not report a bug when there was no bug to discover (true positive) and finally, we use † to show that the tool reported that it did not support the program respectively. When evaluating these benchmarks,

Table 20: Results of verifying rwmutex (read and write mutex) code snippets with CP = rwmu.

Snippet (CS)	Context	Gom.	GCatch	Godel2
	minimal(CS,CP)	✓	✗	✓
	non-dynamic-for(CS,CP,10 000)	✓	✗	✓
	non-dynamic-for(CS,CP,`len(os.Args)`*)*	✓	✗	✓
	dynamic-for(CS,CP,10)	✓	✗	†
	dynamic-for(CS,CP,100)	✓	✗	†
	dynamic-for(CS,CP,120 000)	†	✗	†
	dynamic-for(CS,CP,`len(os.Args)`*)*	✓	✗	†
{*runlock*	*primitive-for(CS,CP,10)*	†	✗	✓
-unlocked}	*primitive-for(CS,CP,*`len(os.Args)`*)*	†	✗	✓
	defer(CS,CP)	✓	✗	✗
	closure(CS,CP)	✗	✗	†
	recursion(CS,CP)	†	✗	✗
	timeout(CS,CP)	✓	✗	✗
	2-branch-select(CS,CP)	✓	✗[1]	†
	interface(CS,CP)	†	✗	†
	minimal(CS,CP)	✓	✓	✗
	non-dynamic-for(CS,CP,10 000)	✓	✓	†
	non-dynamic-for(CS,CP,`len(os.Args)`*)*	✓	✓	†
	dynamic-for(CS,CP,10)	✓	✓	†
	dynamic-for(CS,CP,100)	✓	✓	†
	dynamic-for(CS,CP,120 000)	†	✓	†
	dynamic-for(CS,CP,`len(os.Args)`*)*	✓	✓	†
double-rlock	*primitive-for(CS,CP,10)*	†	✓	†
	primitive-for(CS,CP,`len(os.Args)`*)*	†	✓	†
	defer(CS,CP)	✓	✓	✗
	closure(CS,CP)	✗	✗	†
	recursion(CS,CP)	†	✓	✗
	timeout(CS,CP)	✓	✓	✗
	2-branch-select(CS,CP)	✓	✓	†
	interface(CS,CP)	†	✗	†

Table 21: Results of verifying waitgroup code snippets with CP = wg in each contexts .

Snippet (CS)	Context	Gom.	GCatch	Godel2
	minimal(CS,CP)	✓	✗	✗
	non-dynamic-for(CS,CP,10 000)	✓	✗	✗
	non-dynamic-for(CS,CP,`len(os.Args)`*)*	✓	✗	✗
	dynamic-for(CS,CP,10)	✓	✗	✗
	dynamic-for(CS,CP,100)	✓	✗	✗
	dynamic-for(CS,CP,120 000)	†	✗	✗
{*blocking-wait*,	*dynamic-for(CS,CP,*`len(os.Args)`*)*	✓	✗	✗
negative-add,	*primitive-for(CS,CP,10)*	†	✗	✗
negative-done}	*primitive-for(CS,CP,*`len(os.Args)`*)*	†	✗	✗
	defer(CS,CP)	✓	✗	✗
	closure(CS,CP)	✗	✗	†
	recursion(CS,CP)	†	✗	✗
	timeout(CS,CP)	✓	✗	✗
	2-branch-select(CS,CP)	✓	✗[1]	†
	interface(CS,CP)	†	✗	†

we have chosen `len(os.Args)` (returns the number of arguments given to the program) as an unknown communication parameter because its value is unknown at compile time but always greater than 0. This constraint is important because every bounded instantiated context that take a bound greater than 0 contains a bug.

Gomela reports a program as a *true positive* if at least one verification fails (score returned >0), and *false negative* if all verifications succeed (no bug found), all properties are checked with $S = \{1, 2, 3\}$. Note that $0 \notin S$ because `len(os.Args)` is always greater than 0. For GCatch and Godel2, a *true positive* is a correctly identified buggy program, while a *false negative* corresponds to a missed bug. GCatch reports *false positives* (marked as ✗[1] and ✗[2]). This is due to the tool reporting that certain benchmarks contain a deadlock instead of a safety error or erroneously reporting a blocking bug on a concurrent operation that does not block respectively. The *dynamic-for* context has been instantiated with different bounds based on the results from our empirical analysis in Section 3.2.4.

The bs used are 10, 100 and 120000 which are the median, Q3 and max values given to the bounded dynamic loops reported in our empirical analysis in Table 11 respectively. In addition, the benchmark is instantiated with a bound of `len(os.Args)`. The *primitive-for* and *recursion* contexts have been instantiated with an arbitrary bound of 10 and `len(os.Args)`. The *non-dynamic-for* context has been instantiated with a bound of 10000 and `len(os.Args)`. We chose 10000 because the bound is large enough to test whether tools that rely on exhaustive checking can verify it in a timely manner.

Table 14 shows the results of each tool when instantiating each context with no primitive declaration and an empty snippet (represented as ϵ). As stated earlier, the contexts are purposefully bug-free. Therefore, we except all results to be $\ominus$. The purpose of this context is to determine if the tools reports any false alarm. GOMELA does not generate any model for all contexts, expect *interface* and *recursion*, because they do not contain any concurrency primitive (see Section 4.4). However, for *interface* and *recursion*, a channel has been declared and initialised to make the resulting program valid. GOMELA rejects both of these because GOMELA neither support interfaces nor recursion. GCatch reports a true negative for all contexts while Godel2 does not support *closure*, *interface* and *2-branch-select* and reports no bugs on all others.

Table 15 and Table 16 show which channel blocking bugs the tools support within all contexts. Overall, we can see that GOMELA supports most of these benchmarks compared to GCatch and Godel2 and only misses the blocking bugs from the *closure* and *interface* contexts (which it does not support). In addition, GOMELA does not support *dynamic-for* when the bound is set to 120000 because of the limitation of SPIN which does not support models with more than 255 processes. Godel2 also misses few benchmarks but does not support many of them. GCatch misses most benchmarks and reports a few true positives and does not support blocking `range` statements. GCatch finds a bug when integrating the

range snippet within *2-branch-select* (✗[2]) although the bug reported is a false positive i.e, the bug reported by GCatch is not an actual bug. GCatch falsely reports that a branch in the `select` statement is blocked. Note that, the *blocking-send* snippet has not been integrated within *async-chan-1* and *async-chan-4* because the send would not block in these contexts.

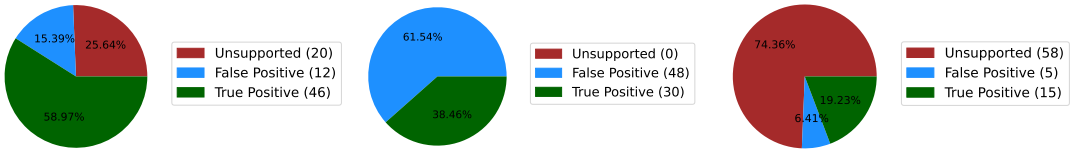
Table 17 shows which channel safety errors the tools support. GOMELA and Godel2 results are very similar to Table 15 and Table 16. GCatch does not support safety errors, however, for five benchmarks (✗[1]) it falsely reports a blocking bug.

Table 18, Table 19 and Table 20 show the evaluation of all mutex (Part I and Part II) and rwmutex snippets within all contexts respectively. GOMELA results are consistent with previous tables whereas GCatch finds many more bugs than previous channel benchmarks. Godel2 supports most of the code snippets (except *double-rlock*) but only a few of the contexts. Interestingly, Godel2 only supports the *primitive-for* benchmarks when instantiated with (rw)mutexes code snippets.

Table 21 shows the evaluation of integrating waitgroup snippets into all contexts. Godel2 and GCatch do not support waitgroups while GOMELA finds all waitgroup bugs in the contexts that it supports.

We acknowledge that there might be a bias involved in selecting and assembling the contexts and code snippets ourselves. To counter this, we plan in the future, to extract contexts and code snippets from bugs found in real-world projects as well as, involving third parties to help us generate a random sample of programs.

RQ1: Overall, we can see that GOMELA finds all the bugs in the contexts that it claims to support. It supports the most benchmarks with a total of 135 true positive. On the other hand, GCatch results are unpredictable because the majority of its results are false negatives (158 out of 220) and the reason why certain contexts are not supported when using specific snippets are unclear. Godel2 only supports a few benchmarks, 52 out of 220, while generating 64 false negatives. The results from applying each tool on all 220 benchmarks are shown in Figure 50.



(a) GOMELA **(b)** GCatch **(c)** Godel2

Figure 51: Proportion of true/false positives in 78 buggy programs, *unsupported* indicates that the tool has aborted or crashed.

```go
func parseFiles(files []os.File) {
  ch := make(chan string)

  for file := range files {
    go parseFile(ch, file)
  }

  for v := range ch { // stuck bc
    close() missing
    fmt.Println(v)
  }
}

func parseFile(ch chan string, file
    os.File) {
  ch ← parseToken(file)
}
```

```go
func main() {
  a := make(chan int)
  b := make(chan int)
  for i := 0; i < 3; i++ {
    go func() {
      ←b // blocked
      a ← 1
    }()
  }
  ←a // blocked
  b ← 1
}
```

Figure 52: Examples of buggy programs caught by GOMELA, missed by GCatch, and unsupported by Godel2.

5.2 RQ2: How applicable Gomela is to real-world programs?

To evaluate the applicability of our tool to real-world programs, we have collected a set of 72 buggy programs that consists of blocking examples from Gabet and Yoshida (2020); Yuan et al. (2021), and six additional programs with intricate concurrency patterns (all benchmarks are available online Dilley and Lange (2021b,c)). These programs range from 12 to 298 LoC (mean: 83, median: 70). Similarly to Section 5.1, we evaluate GOMELA on this set and also compare the result against GCatch (Liu et al. (2021)) and Godel2 (Gabet and Yoshida (2020)). We omit benchmarks from Gabet and Yoshida (2020) that contain data-races or

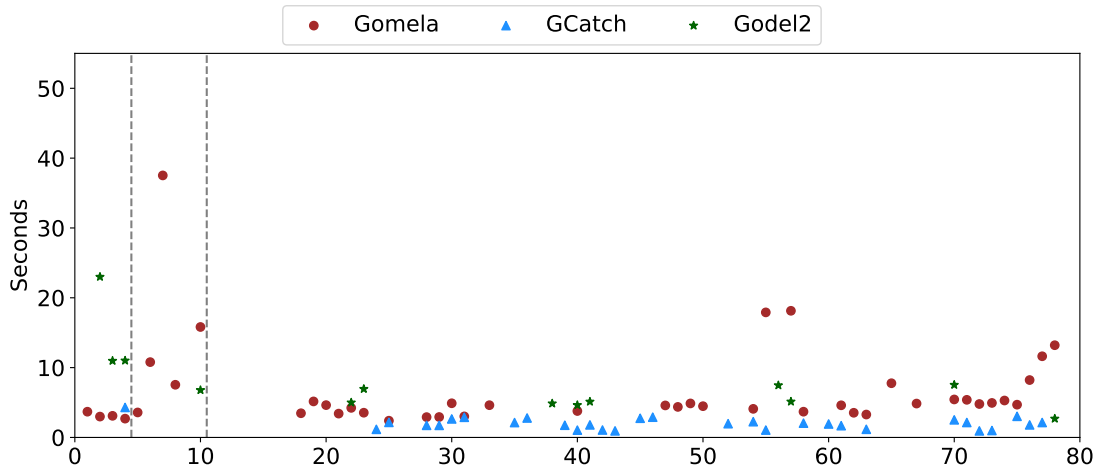


Figure 53: Run-times for true positives in 78 buggy programs.

do not contain any bugs. We focus on the benchmarks from Yuan et al. (2021) that contain blocking bugs as GOMELA does not target other bugs. To ensure that our experiments are precise, we only use the bug *kernels* from Yuan et al. (2021). It is not always straightforward to distinguish false alarms from real bugs raised by these tools, hence we could not confidently evaluate their respective performance on the *real* bugs listed Yuan et al. (2021). Figures 51 and 53 give the results of our experiments. From left to right, the first group is comprised of all four programs that contain a blocking bug from Gabet and Yoshida (2020). The second group consists of a set of 6 benchmarks which contains intricate popular concurrency patterns. The set is made of :

1. The program from Figure 5.
2. A program with a blocking range-over-channel, see Figure 52 (right).
3. A program (`FindAll()`) adapted from a bug report in Google's `gops` project Google (2020); Siebenmann (0 09) (see Example 3).
4. A program with a circular dependency and a bounded `for`-loop, see Figure 52 (left).

5. A program invoking function `preload()` from Figure 22 (see Example 2).

6. a buggy version of Figure 34 where `receiver` receives `x+1` times.

`FindAll()`, `preload()` and the program from Figure 5 are all real-world bugs found using GOMELA when answering research question 3 (see Section 5.3). `FindAll()` and `preload()` use multiple types of concurrency primitives and create a set of goroutines in a **`for`**-loop that send messages on a channel which are then received by the main goroutine. In addition, the bugs in `FindAll()`, `preload()` and the program from Figure 5 only arise when specific values are used for the communication parameters.

The last group consists of the 68 blocking programs from the GoKer benchmark suite devised in Yuan et al. (2021), which are all adapted from real-world programs. This suite is mainly aimed at evaluating *run-time* bug detectors. We modified 29 of them to meet the limitations of the front-end of GOMELA and Godel2. Note that GCatch did also benefited from most of the massaging performed. Our modifications consist in inlining concurrency primitive declarations, inlining higher-order functions, and replacing virtual method calls by static ones.

Overall GOMELA has the highest rate of true positive (58.97%), followed by GCatch (38.46%) and Godel2 (19.23%). While GCatch never crashes/aborts, it misses 61.54% of the bugs. Our experiments show that Godel2 has very limited support for real-world examples.

Figure 53 shows how the three tools perform wrt. execution time for true positives. GCatch is the fastest tool (all processed within 5s) and most examples are verified by GOMELA within 10s. For readability, we have omitted GOMELA's run-time for `preload()` (120s for 27 verifications). Observe that GOMELA is the only tool that can process correctly all programs in the second group of Figure 53. Godel2 identifies the last program of this group as buggy, but also raises a false alarm for the non-buggy version, i.e., the program in Figure 34.

In the 68 *kernels* benchmarks from Yuan et al. (2021), we found a particular

```
1  func main() {
2    var mu sync.RWMutex
3
4    mu.Rlock()
5
6    go func() {
7      mu.Lock()
8      mu.Unlock()
9    }()
10
11   mu.Rlock()
12   mu.RUnlock()
13   mu.RUnlock()
14 }
```

Figure 54: Simplified example adapted from Yuan (2022) which showcases a particular blocking bug that happens when a single goroutine calls `RLock()` twice on the same RWMutex.

rwmutex-related bug, discussed in Section 4.2.2, that is missed by GOMELA. This benchmark is shown in a simplified form in Figure 54.

Even though we have tried to be as inclusive as possible in terms of the choice of benchmarks, we will aim in future work to involve third parties in the selection process of the benchmarks to reduce the potential selection bias.

RQ2: Overall, we have found that GOMELA finds the most bugs overall while GCatch, even though it is faster on average than GOMELA, generates many false alarms and false positives. We have found that Godel2 only supports 26% of the benchmarks but finds 75% of bugs in the benchmarks that it supports.

5.3 RQ3: How does Gomela scale to real-world programs?

To evaluate the scalability of our approach we ran GOMELA on the same set of 125 Github projects used in Chapter 3. For each project, we cloned it locally and generated models for each of its packages. Note that, the compilation unit in Go is the package (i.e., a directory), hence this is an important level of granularity

Table 22: Project sizes and verification run-times.

Project sizes	mean	std	25%	50%	75%	max
models per project	50.41	84.04	4.5	19	57.5	499
models per package	3.75	11.17	1	1	3	295
parameters per models	2.08	2.35	1	1	2	42
states per valuation	8118.73	166k	18	43	132	8.94 mil
Run-time						
per project	4m20s	8m3s	19s	58s	4m	52m11s
per model	4.67s	32.54s	2.50s	2.72s	3.22s	22m25s
per valuation	3.42s	1.56s	2.57s	3.06s	3.87s	18s

Table 23: Verification scores for generated models.

All scores	mean	std	25%	50%	75%	max
model deadlock ϕ_{MD}	0.031	0.17	0	0	0	1
channel safety ϕ_{CS}	0.0019	0.043	0	0	0	1
mutex safety ϕ_{MS}	0.0015	0.037	0	0	0	1
waitgroup safety ϕ_{WS}	0.0012	0.031	0	0	0	1
Strictly positive scores (occurrences)						
model deadlock (179)	0.88	0.29	1	1	1	1
channel safety (11)	0.92	0.23	1	1	1	1
mutex safety (9)	0.96	0.1	0.66	1	1	1
waitgroup safety (8)	0.85	0.25	0.76	1	1	1

for GOMELA too (as it is built atop the Go parser). Each model was then verified, wrt. the properties described in Section 4.3.1, using our automated approach with $S = \{0, 1, 3\}$.

Our measurements for scalability are in Table 22. The key factors that affect the run-time of GOMELA on a given project is the number of models it generates and how many parameters these projects have. Recall that GOMELA only generates a model when it detects at least one concurrency primitive. Table 22 (top) shows that most projects and packages only give rise to a couple of models, but some produce more than one hundred models. If a model has k parameters, it may require up to $|S|^k$, but the table also shows that 75% of these models have at most 2 concurrency parameters (hence they require at most $|S|^2$ verifications). The last row shows the number of states as reported by SPIN when verifying it. This metric is representative of the number of goroutines and the number of concurrency operations contained in the source program. Overall GOMELA generated 5327 models, out of which 27 were not valid Promela (due to limitations of our aliasing analysis) and were rejected by SPIN, 32 contained >5 concurrency parameters and 118 contained at least one valuation that took >30 seconds to verify (we omitted these in the next phase).

Table 22 (bottom) shows the run-time for the remaining 5150 models we verified. More than 75% of projects can be verified in under 4 minutes, and a model valuation is verified in under 3.42s on average (we set a timeout of 30s per valuation). This suggests that our approach does scale to real-world Go.

Table 23 (top) shows the score computed for the 5150 models. As expected from mature projects on GitHub, GOMELA reports few concurrency bugs. This suggests that it has a reasonable false alarm rate. Table 23 (bottom) focuses on the models for which at least one valuation violated a property. We can see that the average score of all properties is close to one (>0.85) which means that, on average, all evaluations report an error no matter what the values given to

```
1  var x int = 0
2
3  func TestRacex() {
4    c := make(chan struct{})
5    go func() {
6      x = 1
7      close(c)
8    }()
9
10   if x == -1 {
11     close(c)
12   }
13   <-c
14 }
```

Figure 55: Simplified example from Golang (2021d) which GOMELA falsely reports as having a double close error due to data dependency. `x` is never equal to -1.

communication parameters.

To determine how accurate these reports are, and in particular what the rate of false alarms is, we have *manually* analysed 10 random programs out of 179 reported deadlocks and all other 28 safety errors detected by GOMELA. This amounts to a total of 38 *models* with at least one strictly positive score. By *manual analysis*, we mean that the authors looked at the source code of the program to determine whether or not the error reported by GOMELA can indeed happen at run time (i.e., there is an execution that leads to a goroutine leak or a safety error). In every case, we took a conservative approach, i.e., we report a true positive only if were are convinced the bug is indeed reachable. Dilley and Lange (2022e,d) contains a list of all bugs reported and the results of our manual analysis.

Out of 10 reported model deadlocks we analysed, 6 were real bugs and 4 were false alarms (due to higher-order functions and tricky aliasing). Out of all 11 channel safety errors, 10 were real bugs while one was a false alarm due to how we over-approximate conditional statements. Figure 55 shows a simplified example of the false alarm raised by GOMELA in the Golang (2021a) project where, clearly, the channel will never be closed twice due to `x` never being equal to -1. Out of all 9 mutex errors, one was a true positive (a case similar to *unlock-unlocked*, see

```
1
2     go func() {
3       var wg sync.WaitGroup
4       wg.Add(SHARD_COUNT)
5
6       for range m {
7         var shard sync.RWMutex
8         go func() {
9           shard.RLock()
10          for range items {
11            ch <- "..."
12          }
13          shard.RUnlock()
14          wg.Done()
15        }()
16      }
17      wg.Wait()
18      close(ch)
19    }()
20
21    for range ch {
22      // (...)
23    }
24  }
```

Figure 56: Example of a tricky false alarm, adapted from Snail007 (2022).

Figure 38) but the others were false alarms (all due to higher-order functions). Finally, the 8 reports of waitgroup errors were real bugs.

We conclude this section by commenting on Figure 56 which shows a simplified version of a function for which our tool detected two safety errors and one possible deadlock, which all revealed to be false alarms. Figure 56 is adapted from GoProxy an HTTP proxy project on GitHub, with 11.8k stars and 20 contributors. Figure 56 is a simplification of a function that returns the keys of a (custom) ConcurrentMap which consists of a `slice` of "sharded" slices Snail007 (2022). We made several modification to obtain a more self-contained program, but did not modify its concurrent behaviour besides the creation of `Mutex shard` at Line 7. In the original code, a mutex is attached each element in the slice `m` — this modification has no consequence wrt. the program Figure 56. GOMELA produces a very accurate Promela model of Figure 56, with three concurrency parameters: `SHARD_COUNT`, `count` and `items`. We observe the following:

1. If `len(m)` < `SHARD_COUNT`, the counter of `wg` will not be decremented enough

and the program will get stuck at line 17 (blocking bug).

2. If `len(m)` > `SHARD_COUNT` and `len(items)` = 0, the counter of `wg` may reach a negative number hence triggering an error: `panic: sync: negative WaitGroup counter`.

3. If `len(m)` > `SHARD_COUNT` and `len(items)` > 0, the program may trigger either a negative WaitGroup counter or a send on close error, i.e., `panic: send on closed channel`. In the latter case, the `Wait` at Line 17 succeeds while some goroutines spawned at Line 8 are still running.

While all these are real bugs in the program from Figure 56, they do not occur in the original program Snail007 (2022) as the slice `m` is always initialised with exactly `SHARD_COUNT` elements. Note that when `SHARD_COUNT` = `len(m)`, the three bugs described above are not reachable. We argue that GOMELA's report is still relevant in this case at it highlights a brittle dependency between constants and variables that can be easily solved, e.g., by replacing Line 4 with `wg.Add(len(m))`. With this fix, the dependency is made explicit and GOMELA will not report errors.

Figure 56 and its source in Snail007 (2022) also highlight the work required to assess whether reports are indeed a false alarm.

RQ3: We have found that GOMELA scales well to real-world programs and can verify each program in under 4min20s on average. From the manual analysis that we have performed on the bugs found by GOMELA, we have found that the majority of bugs reported were real bugs and that most of the false alarms were due to unsupported features such as high-order functions.

5.4 Conclusion

We have devised a set of 298 benchmarks which aims at evaluating the supported functionalities and the applicability of our approach to real-world Go programs.

We have then compared our tool to recent static checkers namely, GCatch and Godel2, on this set of benchmarks and reported on the results. We have found that Gomela found 63% of bugs while GCatch and Godel2 found less than 27% of bugs. In addition, Gomela supports more programs than Godel2 and does not miss as many bugs as GCatch. After that, we ran Gomela on 125 Github projects to see if it could scale to much larger Go program. Overall, we have found that Gomela scales well to real-world Go code. Gomela verifies the majority of projects in under 1 minute per project and a single model in less than 5 seconds on average. Remarkably, most false alarms we discovered were due to limitations of our front-end rather than over-approximation in our verification approach.

Chapter 6

Conclusions and Future Directions

6.1 Overview of the main contributions.

Empirical analysis of concurrent Go programs. To gain knowledge of how Go developers use concurrency in Go projects, we have performed an empirical analysis on 125 Github projects. This helped us evaluate how much of these projects were supported by previous work on the static verification of Go programs. We have found that most projects spawned goroutine in `for`-loops (85%), used waitgroups (76%) and mutexes (58%) which were not fully supported by previous work. In addition, we found that the number of concurrency operations per concurrency primitive was low (<2) which allowed us to build an effective partitioning approach.

Static verification of concurrency in Go programs. Following the insights gained in the empirical analysis, we developed a novel verification approach using bounded model checking of parameterised behavioural types which over-approximate their programs. Our approach builds on the technique introduced

in Lange et al. (2017) to support programs that spawn a parameterised number of goroutines or channel capacities, as well as waitgroups and mutexes. These extensions increase the applicability of the approach to real-world programs which often contain these features as reported in our empirical analysis in Chapter 3. One of the main challenges with software verification is the difficulty to scale to large programs. To increase the scalability of our approach, we partition the programs into function declarations that do not take concurrency primitives as parameters and use the well-known model-checker SPIN to verify the resulting models. This allows us to verify whole Github projects in under 4 minutes on average.

A platform to synthesise programs. To evaluate the subset of Go as well as which bugs are supported by static checkers of concurrent Go programs, we devised a technique to synthesise buggy programs from parameterised contexts. These contexts are instantiated with various concurrency primitives, code snippets and bounds to produce buggy benchmarks. This platform was used to produce a total of 220.

Evaluation and comparison of Gomela against state-of-the-art. We have implemented our approach into a tool, called Gomela, and evaluated it on 298 benchmarks aimed at assessing the functionalities supported by Gomela and the applicability of the approach to real-world projects. In addition, we have also run this set of benchmarks on Godel2 and GCatch to compare our results with state-of-the-art tools in the verification of concurrent Go programs and found that our approach supports a larger set of benchmarks than the competition. We have also fed Gomela with the same 125 Github projects than in the empirical analysis and found that our approach scales well to real-world Go code. Another challenge with software verification is proper feedback. When manually analysing the results of Gomela, we discovered that all false alarms were due to limitations of our front-end rather than over-approximation in our verification approach.

6.2 Future work

In this section, we discuss the opportunities for future work that could stem from the approaches introduced in earlier chapters.

Empirical analysis. In the shorter term, we plan to extend our empirical analysis from Chapter 3 to pursue an inter-procedural analysis on the usage of goroutines and channels in Go projects to better reflect the number of goroutines and channels created in `for`-loops. In the longer term, we plan to extend the analysis to compare the usage of message passing in languages such as Go, Rust, and Erlang which all provide natively message-passing facilities. In addition, we would like to study the usage of additional concurrency primitives that are offered by the Go `sync` package such as `sync.Cond` and `sync.Once`.

Benchmark suite. We plan to adapt the set of benchmarks devised in Chapter 5 to other programming languages such as Erlang and Rust which would test the applicability and scalability of static and dynamic checkers in those languages. In addition, we plan to also include non-buggy programs. These non-buggy programs will be aimed at evaluating the false alarm rate of these checkers.

Static verification of Go programs. We also plan to improve our static verification approach introduced in Chapter 4 by developing: (*i*) scalable techniques to deal with virtual method calls, (*ii*) heuristics to deal with models with a high number of parameters, and (*iii*) techniques to help automate the identification of false alarms. Recall that we have found, via our manual analysis, that false alarms were generally due to the presence of high-order functions and usage of structures. Furthermore, if a technique to distinguish false alarms existed, techniques to mitigate false alarms such as counterexample-guided abstraction by Clarke et al. (2000) could be applied to reduce the number of false alarms. In the shorter term, we

also would like to improve our front-end analysis so that more Go programs can be analysed without manual modifications. We also plan to add support for the blocking bug that can arise when sequentially calling `Rlock()` twice on the same RWMutex as described at the end of Section 5.2. In the longer term, we plan to use our tool to detect concurrency errors and suggest repairs for large code-bases. In addition, we plan to apply our verification approach on other programming languages that supports channel-based concurrency such as Rust.

Bibliography

Beego (2021a). Beego github project. `https://github.com/beego/beego`.

Beego (2021b). Example of multiple unlock() and lock() per mutex declaration. `https://github.com/beego/beego/blob/develop/server/web/session/sess_file_test.go`.

Bocchi, L., Yang, W. and Yoshida, N. (2014). Timed multiparty session types. In P. Baldan and D. Gorla, eds., *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, *Lecture Notes in Computer Science*, vol. 8704, Springer, pp. 419–434.

Borges, H. and Valente, M. T. (2018). What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146, pp. 112–129.

Brzozowski, J. A. (1964). Derivatives of regular expressions. *J ACM*, 11(4), pp. 481–494.

Cassar, I., Francalanza, A., Aceto, L. and Ingólfsdóttir, A. (2017). A survey of runtime monitoring instrumentation techniques. In A. Francalanza and G. J. Pace, eds., *Proceedings Second International Workshop on Pre- and Post-Deployment Verification Techniques, PrePost@iFM 2017, Torino, Italy, 19 September 2017*, *EPTCS*, vol. 254, pp. 15–28.

Castro-Perez, D., Hu, R., Jongmans, S., Ng, N. and Yoshida, N. (2019). Distributed programming using role-parametric session types in go: statically-typed endpoint apis for dynamically-instantiated communication structures. *Proc ACM Program Lang*, 3(POPL), pp. 29:1–29:30.

Chabbi, M. and Ramanathan, M. K. (2022). A study of real-world data races in golang. *CoRR*, abs/2204.00764, `2204.00764`.

Chi, G. (2021a). Chi github project. `https://github.com/go-chi/chi`.

Chi, G. (2021b). throttle-test code. `https://github.com/go-chi/chi/blob/25eb15cdd4f644896eac2ac05d4e3e932f34a188/middleware/throttle_test.go`.

Clarke, D. G., Potter, J. M. and Noble, J. (1998). Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, New York, NY, USA: Association for Computing Machinery, OOPSLA '98, p. 48–64.

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y. and Veith, H. (2000). Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, eds., *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, *Lecture Notes in Computer Science*, vol. 1855, Springer, pp. 154–169.

Clearfield, C. and Lofchie, S. (2013). Nasdaq and the Facebook IPO. `http://www.system-logic.com/commentary/2013/06/04/NASDAQ-and-the-Facebook-IPO-part-1-The-Race-Condition.html`, accessed: 2022-03-02.

Coppo, M., Dezani-Ciancaglini, M., Yoshida, N. and Padovani, L. (2016). Global progress for dynamically interleaved multiparty sessions. *Math Struct Comput Sci*, 26(2), pp. 238–302.

Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1), pp. 46–55.

Danial, A. (2021). Count lines of code. `https://github.com/AlDanial/cloc`.

Deniélou, P.-M. and Yoshida, N. (2011). Dynamic multirole session types. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA: Association for Computing Machinery, POPL '11, p. 435–446.

Dilley, N. and Lange, J. (2019). An empirical study of messaging passing concurrency in go projects. In X. Wang, D. Lo and E. Shihab, eds., *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, IEEE, pp. 377–387.

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

Dilley, N. and Lange, J. (2021a). Automated verification of go programs via bounded model checking. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, IEEE, pp. 1016–1027.

Dilley, N. and Lange, J. (2021b). Automated verification of go programs via bounded model checking (artifact). `https://doi.org/10.5281/zenodo.5222045`, accessed: 2021-10-17.

Dilley, N. and Lange, J. (2021c). Gomela. `https://github.com/nicolasdilley/gomela-ase21/`.

Dilley, N. and Lange, J. (2021d). Survey data. https://github.com/nicolasdilley/gocurrency_tool/tree/b11c6be9dbeb12565657ed839dbbccdef19df2eb/analyser/results/html.

Dilley, N. and Lange, J. (2022a). Benchmark-maker. https://github.com/nicolasdilley/benchmark-maker.

Dilley, N. and Lange, J. (2022b). Go project analyser. http://github.com/nicolasdilley/gocurrency_tool.

Dilley, N. and Lange, J. (2022c). Gomela. http://github.com/nicolasdilley/gomela.

Dilley, N. and Lange, J. (2022d). Manual analysis of chapter 5 rq3. https://github.com/nicolasdilley/Gomela/blob/master/results/manual-analysis.csv.

Dilley, N. and Lange, J. (2022e). Verification results of chapter 5 rq3. https://github.com/nicolasdilley/Gomela/blob/master/results/scores.csv.

Docker-slim (2021). Docker-slim github project. https://github.com/docker-slim/docker-slim.

Drone (2021). Drone github project. https://github.com/drone/drone.

Elastic (2021). Run. https://github.com/elastic/beats/blob/49e3857fd412ff95d4374118e6e447aff9ebb619/x-pack/elastic-agent/pkg/composable/controller.go#L98.

Etcd-io (2021). Etcd github project. https://github.com/etcd-io/etcd.

Forum, M. P. I. (1993). CORPORATE the MPI forum - MPI: a message passing interface. In B. Borchers and D. Crawford, eds., *Proceedings Supercomputing '93, Portland, Oregon, USA, November 15-19, 1993*, ACM, pp. 878–883.

Francalanza, A. and Seychell, A. (2015). Synthesising correct concurrent runtime monitors. *Formal Methods Syst Des*, 46(3), pp. 226–261.

Gabet, J. and Yoshida, N. (2020). Static race detection and mutex safety and liveness for go programs. In R. Hirschfeld and T. Pape, eds., *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, *LIPIcs*, vol. 166, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 4:1–4:30.

Geist, A. et al. (1996). MPI-2: extending the message-passing interface. In L. Bougé, P. Fraigniaud, A. Mignotte and Y. Robert, eds., *Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Lyon, France, August 26-29, 1996, Proceedings, Volume I*, *Lecture Notes in Computer Science*, vol. 1123, Springer, pp. 128–135.

Go-chi (2021). Chi code. `https://github.com/go-chi/chi/blob/fb48a47641af106c7eae2f09864e7fecd54237bf/middleware/throttle.go#L43`.

Golang (2018). The Go programming language. `https://golang.org/`.

Golang (2021a). Go github project. `https://github.com/golang/go`.

Golang (2021b). The Go playground — a concurrent prime sieve. `https://play.golang.org/p/9U22NfrXeq`, accessed: 2021-01-20.

Golang (2021c). testconcurrentreadwritereqbody code. `https://github.com/golang/go/blob/1b09d430678d4a6f73b2443463d11f75851aba8a/src/net/http/clientserver_test.go#L735`.

Golang (2021d). Testraceissue12664_2 code. `https://github.com/golang/go/blob/2580d0e08d5e9f979b943758d3c49877fb2324cb/src/runtime/race/testdata/issue12664_test.go#L35`.

Google (2020). Findall code. `https://github.com/google/gops/blob/6fb0d860e5fa50629405d9e77e255cd32795967e/goprocess/gp.go#L29`.

Google (2021). preload code. `https://github.com/google/trillian/blob/c92fa63aaa6c133eb8383f2727524421bea420c4/storage/cache/subtree_cache.go#L108`.

Google (2022a). Go sync package documentation. `https://pkg.go.dev/sync#WaitGroup`.

Google (2022b). Go sync package documentation. `https://pkg.go.dev/sync`.

Graham, R. L., Woodall, T. S. and Squyres, J. M. (2005). Open MPI: A flexible high performance MPI. In R. Wyrzykowski, J. J. Dongarra, N. Meyer and J. Wasniewski, eds., *Parallel Processing and Applied Mathematics, 6th International Conference, PPAM 2005, Poznan, Poland, September 11-14, 2005, Revised Selected Papers*, *Lecture Notes in Computer Science*, vol. 3911, Springer, pp. 228–239.

Groote, J. F. and Mousavi, M. R. (2014). *Modeling and Analysis of Communicating Systems*. MIT Press.

Gropp, W. (2002). MPICH2: A new start for MPI implementations. In D. Kranzlmüller, P. Kacsuk, J. J. Dongarra and J. Volkert, eds., *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 9th European PVM/MPI Users' Group Meeting, Linz, Austria, September 29 - October 2, 2002, Proceedings*, *Lecture Notes in Computer Science*, vol. 2474, Springer, p. 7.

Havelund, K. and Pressburger, T. (2000). Model checking JAVA programs using JAVA pathfinder. *Int J Softw Tools Technol Transf*, 2(4), pp. 366–381.

Hewitt, C., Bishop, P. and Steiger, R. (1973). A universal modular actor formalism

for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., IJCAI'73, p. 235–245.

Hoare, C. A. R. (1978). Communicating sequential processes. *Commun ACM*, 21(8), pp. 666–677.

Holzmann, G. J. (1997). The model checker SPIN. *IEEE Trans Software Eng*, 23(5), pp. 279–295.

Honda, K. (1993). Types for dyadic interaction. In E. Best, ed., *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, *Lecture Notes in Computer Science*, vol. 715, Springer, pp. 509–523.

Honda, K., Vasconcelos, V. T. and Kubo, M. (1998). Language primitives and type discipline for structured communication-based programming. In C. Hankin, ed., *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, *Lecture Notes in Computer Science*, vol. 1381, Springer, pp. 122–138.

Honda, K., Yoshida, N. and Carbone, M. (2008). Multiparty asynchronous session types. In G. C. Necula and P. Wadler, eds., *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, ACM, pp. 273–284.

Hüttel, H. et al. (2016). Foundations of session types and behavioural contracts. *ACM Comput Surv*, 49(1), pp. 3:1–3:36.

Jhala, R. and Majumdar, R. (2009). Software model checking. *ACM Comput Surv*, 41(4), pp. 21:1–21:54.

Johnson, B., Song, Y., Murphy-Hill, E. R. and Bowdidge, R. W. (2013). Why don't software developers use static analysis tools to find bugs? In D. Notkin, B. H. C. Cheng and K. Pohl, eds., *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, IEEE Computer Society, pp. 672–681.

K3s-io (2021a). Generatenodemap code. `https://github.com/k3s-io/k3s/blob/cfe7e0c73483038d9e9242ae95da1a7c466f2891/pkg/agent/tunnel/tunnel.go#L165`.

K3s-io (2021b). K3s github project. `https://github.com/k3s-io/k3s`.

Kalliamvakou, E. et al. (2014). The promises and perils of mining github. In P. T. Devanbu, S. Kim and M. Pinzger, eds., *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, ACM, pp. 92–101.

Kozen, D. (1983). Results on the propositional mu-calculus. *Theor Comput Sci*, 27, pp. 333–354.

Kubernetes (2021a). Generatenodemap code. `https://github.com/kubernetes/kubernetes/blob/d70ee902fddc682863a3cc4f0d8eac0223ebf70b/test/e2e/storage/vsphere/nodemapper.go#L62`.

Kubernetes (2021b). Kubernetes (k8s). `https://github.com/kubernetes/kubernetes`.

Labstack (2021a). Echo github project. `https://github.com/labstack/echo`.

Labstack (2021b). Testechostart code. `https://github.com/labstack/echo/blob/4b88e25e49537dacca73903ccd243f734fdbbe9c/echo_test.go#L765`.

Landman, D., Serebrenik, A. and Vinju, J. J. (2017). Challenges for static analysis of Java reflection: literature review and empirical study. In *ICSE 2017*, pp. 507–518.

Lange, J., Tuosto, E. and Yoshida, N. (2015). From communicating machines to graphical choreographies. In S. K. Rajamani and D. Walker, eds., *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, ACM, pp. 221–232.

Lange, J., Ng, N., Toninho, B. and Yoshida, N. (2017). Fencing off go: liveness and safety for channel-based programming. In G. Castagna and A. D. Gordon, eds., *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, ACM, pp. 748–761.

Lange, J., Ng, N., Toninho, B. and Yoshida, N. (2018). A static verification framework for message passing in go using behavioural types. In M. Chaudron, I. Crnkovic, M. Chechik and M. Harman, eds., *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, ACM, pp. 1137–1148.

Larsen, K. G., Pettersson, P. and Yi, W. (1997). Uppaal in a nutshell. *Int J Softw Tools Technol Transf*, 1(1–2), p. 134–152.

Liu, Z., Zhu, S., Qin, B., Chen, H. and Song, L. (2021). Automatically detecting and fixing concurrency bugs in go software systems. In T. Sherwood, E. D. Berger and C. Kozyrakis, eds., *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, ACM, pp. 616–629.

Liu, Z., Xia, S., Liang, Y., Song, L. and Hu, H. (2022). Who goes first? detecting go concurrency bugs via message reordering. In B. Falsafi, M. Ferdman, S. Lu and T. F. Wenisch, eds., *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, ACM, pp. 888–902.

Lu, S., Park, S., Seo, E. and Zhou, Y. (2008). Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS 2008*, pp. 329–339.

Magee, J. and Kramer, J. (1999). *Concurrency - state models and Java programs*. Wiley.

Marinescu, C. (2014). An empirical investigation on MPI open source applications. In *EASE 2014*, pp. 20:1–20:4.

Midtgaard, J., Nielson, F. and Nielson, H. R. (2018). Process-local static analysis of synchronous processes. In A. Podelski, ed., *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, *Lecture Notes in Computer Science*, vol. 11002, Springer, pp. 284–305.

Milner, R. (1989). *Communication and concurrency*. PHI Series in computer science, Prentice Hall.

Milner, R., Parrow, J. and Walker, D. (1992). A calculus of mobile processes, I. *Inf Comput*, 100(1), pp. 1–40.

Ng, N. and Yoshida, N. (2016). Static deadlock detection for concurrent Go by global session graph synthesis. In *CC 2016*, pp. 174–184.

Okur, S. and Dig, D. (2012). How do developers use parallel libraries? In *FSE 2012*, p. 54.

Pepitone, J. (2012). Facebook ipo: What the https://money.cnn.com/2012/05/23/technology/facebook-ipo-what-went-wrong/index.htm, accessed: 2022-03-02.

Photoprism (2021a). Example of multiple unlock() per lock(). https://github.com/photoprism/photoprism/blob/c520cb4ee49c19075f3d782afb8506ed88115c91/internal/api/websocket.go#L94.

Photoprism (2021b). Photoprism github project. https://github.com/photoprism/photoprism.

Pike, R. (2012). Go concurrency patterns. https://talks.golang.org/2012/concurrency.slide#9.

Pike, R. (2015). Go proverbs. https://www.youtube.com/watch?v=PAAkCSZUG1c.

Pingcap (2021). Tidb github project. https://github.com/golang/tidb.

Pinto, G., Torres, W. and Castor, F. (2015). A study on the most popular questions about concurrent programming. In *PLATEAU 2015*, pp. 39–46.

Pinto, G., Torres, W., Fernandes, B., Filho, F. C. and de Barros, R. S. M. (2015). A large-scale study on the usage of Java's concurrent programming constructs. *Journal of Systems and Software*, 106, pp. 59–81.

Saboury, A., Musavi, P., Khomh, F. and Antoniol, G. (2017). An empirical study of code smells in JavaScript projects. In *SANER 2017*, pp. 294–305.

Siebenmann, C. (2020-09). Even in Go, concurrency is still not easy (with an example). https://utcc.utoronto.ca/~cks/space/blog/programming/GoConcurrencyStillNotEasy.

Sirupsen (2021a). Logrus code. `https://github.com/sirupsen/logrus/blob/ac6e35b4c213b54a2086b831179b9c324f519695/logrus_test.go#L585`.

Sirupsen (2021b). Logrus github project. `https://github.com/sirupsen/logrus`.

Snail007 (2022). Keys code. `https://github.com/snail007/goproxy/blob/7e0406bdb90960fa0c0d9c89a770ef206c4c02d8/utils/map.go#L243`.

Stadtmüller, K., Sulzmann, M. and Thiemann, P. (2016). Static trace-based deadlock analysis for synchronous mini-go. In A. Igarashi, ed., *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, *Lecture Notes in Computer Science*, vol. 10017, pp. 116–136.

Sulzmann, M. and Stadtmüller, K. (2017). Trace-based run-time analysis of message-passing go programs. In O. Strichman and R. Tzoref-Brill, eds., *Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings*, *Lecture Notes in Computer Science*, vol. 10629, Springer, pp. 83–98.

Sulzmann, M. and Stadtmüller, K. (2018). Two-phase dynamic analysis of message-passing go programs based on vector clocks. In D. Sabel and P. Thiemann, eds., *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018*, ACM, pp. 22:1–22:13.

Taheri, S. and Gopalakrishnan, G. (2021). Automated dynamic concurrency analysis for go. *CoRR*, abs/2105.11064, `2105.11064`.

Tasharofi, S., Dinges, P. and Johnson, R. E. (2013). Why do Scala developers mix the actor model with other concurrency models? In *ECOOP 2013*, pp. 302–326.

team, T. G. (2017). Go survey results. `blog.golang.org/survey[year]-results`.

Torres, W. et al. (2011). Are Java programmers transitioning to multicore?: a large scale study of Java FLOSS. In *SPLASH 2011*, pp. 123–128.

Tu, T., Liu, X., Song, L. and Zhang, Y. (2019). Understanding real-world concurrency bugs in go. In I. Bahar, M. Herlihy, E. Witchel and A. R. Lebeck, eds., *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, ACM, pp. 865–878.

Turing, A. M. (1937). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1), pp. 230–265, `https://academic.oup.com/plms/article-pdf/s2-42/1/230/4317544/s2-42-1-230.pdf`.

Vetter, J. S. and de Supinski, B. R. (2000). Dynamic software testing of MPI applications with umpire. In J. Donnelley, ed., *Proceedings Supercomputing 2000, November 4-10, 2000, Dallas, Texas, USA. IEEE Computer Society, CD-ROM*, IEEE Computer Society, p. 51.

Vitessio (2021). Vitess github project. `https://github.com/vitessio/vitess`.

Wu, D., Chen, L., Zhou, Y. and Xu, B. (2015). An empirical study on C++ concurrency constructs. In *ESEM 2015*, pp. 257–266.

Wu, D., Chen, L., Zhou, Y. and Xu, B. (2016). An extensive empirical study on C++ concurrency constructs. *Information & Software Technology*, 76, pp. 1–18.

Yuan, T. (2022). kubernetes62464. `https://github.com/timmyyuan/gobench/blob/b9eac4dc18cc5328e8ff57a3fa677975027fd73f/gobench/goker/blocking/kubernetes/62464/kubernetes62464_test.go`.

Yuan, T. et al. (2021). Gobench: A benchmark suite of real-world go concurrency bugs. In J. W. Lee, M. L. Soffa and A. Zaks, eds., *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, IEEE, pp. 187–199.

Yuan, X. and Yang, J. (2020). *Effective Concurrency Testing for Distributed Systems*, New York, NY, USA: Association for Computing Machinery. p. 1141–1156.

Zaks, A. and Joshi, R. (2008). Verifying multi-threaded C programs with SPIN. In K. Havelund, R. Majumdar and J. Palsberg, eds., *Model Checking Software, 15th International SPIN Workshop, Los Angeles, CA, USA, August 10-12, 2008, Proceedings*, *Lecture Notes in Computer Science*, vol. 5156, Springer, pp. 325–342.