An Empirical Study of Messaging Passing Concurrency in Go Projects

Nicolas Dilley
School of Computing
University of Kent
Canterbury, UK
nd315@kent.ac.uk

Julien Lange
School of Computing
University of Kent
Canterbury, UK
j.s.lange@kent.ac.uk

***Abstract*—Go is a popular programming language renowned for its good support for system programming and its channel-based message passing concurrency mechanism. These strengths have made it the language of choice of many platform software such as Docker and Kubernetes. In this paper, we analyse 865 Go projects from GitHub in order to understand how message passing concurrency is used in publicly available code. Our results include the following findings: (1) message passing primitives are used frequently and intensively, (2) concurrency-related features are generally clustered in specific parts of a Go project, (3) most projects use *synchronous* communication channels over asynchronous ones, and (4) most Go projects use simple concurrent thread topologies, which are however currently unsupported by existing static verification frameworks.**

***Index Terms*—Golang, message passing, static analysis, empirical study**

I. Introduction

Go is an open source programming language which was initiated by Google in 2009. Go is renowned for its good support for system programming and its channel based concurrency mechanism. It is advertised as "*an open source programming language that makes it easy to build simple, reliable, and efficient software*" [6]. 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 therefore 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*" [21]; 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 errors such as deadlock and thread starvation. 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 errors. Several research groups have recently worked towards filling this gap by developing a range of theories and tools intended to support developers in finding synchronisation bugs in Go programs, either statically (compile-time) or dynamically (runtime). Ng and Yoshida [19] first proposed a tool to statically detect global deadlock in Go programs using choreography synthesis [13]. Later, Stadtmüller et al. [25] proposed another static verification approach, based on forkable regular expressions, to detect global deadlocks. Lange et al. [11], [12] proposed two more advanced static verification frameworks which approximate Go programs with *behavioural types* [8] through their SSA intermediate representation. Various safety and liveness properties can be checked on behavioural types using bounded executions in [11] and exhaustive model checking in [12]. Midtgaard et al. [16] proposed a static verification approach based on abstract interpretation for detecting global deadlocks in a small subset of Go (without recursion). Sulzmann and Stadtmüller [26], [27] addressed the dynamic verification of Go programs. They proposed a trace-based method to analyse Go programs which only use synchronous channels in [26]; and an improved approach, supporting asynchronous channels and relying on vector clocks, is introduced in [27]. Both works require the code to be instrumented before the analysis.

Unsurprisingly, the static approaches mentioned above only provide partial support of the Go language. For instance, none of the static verification frameworks in [12], [16], [19], [25] can verify programs that spawn new threads within a `for` loop. The work in [11] only provides an unsound approximation for such programs. Additionally, these approaches have only been demonstrated on small Go programs or programs with fairly low usage of message passing primitives. Dynamic verification approaches instead may support a larger subset of the language since supporting additional features only requires further instrumentation. However, they are also impacted by intensive usage of message passing primitives. For instance, Sulzmann and Stadtmüller report up to 41% of tracing overhead for programs with high level of concurrency [27].

Our goal is to obtain a better understanding of how the message passing primitives of Go are used in practice by analysing publicly available Go projects. These observations can be used to guide research in the area of static or dynamic verification of message passing programs. Our study will allow 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.

We have implemented a tool-chain that analyses Go programs, which we have applied on 865 Go projects from GitHub. This paper presents the results of our study, which is structured around four research questions stemming from the point of view of the static verification of message passing concurrent programs.

RQ1: *How often are messaging passing operations used in Go projects?* The Go language natively offers a wide range of channel-based (message passing) primitives which differ significantly from traditional synchronisation mechanisms based on shared memory. This research question is about how frequently and intensively these primitives are used in practice. This is relevant to both static and dynamic verification since both are impacted by the number of message passing primitives occurring in a program. Static verification frameworks rely on checking properties of a model (e.g., behavioural type or forkable expression) whose size grows with the number of primitives used in the program. In dynamic verification frameworks, the code need to be instrumented around each primitive. Hence if more primitives are used, more data need to be recorded and analysed.

We have found that most Go projects use message passing mechanisms and use them intensively. However, the number of message passing operations per channel is relatively low, which suggests that programmers use simple protocols to synchronise threads over channels.

RQ2: *How is concurrency spread across Go projects?* Go is the main programming language of very large projects such as Docker and Kubernetes, with hundreds of thousands lines of code. Automatically verifying such projects (statically or dynamically) as a whole is generally unfeasible. This research question investigates whether Go projects may be divided into sequential and concurrent parts, and how significant these portions are. We have found that, even though most Go projects use message passing concurrency, only a limited part of their code-base contains concurrency-related 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 primitives are blocking by default. 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 channels bounds may not be known statically and (ii) the state space of the model grows exponentially with the capacity of the channel.

We have found that 61% 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 under 5). 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. An infinite or complex concurrent topology leads to an infinite state-space which renders techniques such as model checking prohibitively costly or impossible. This research question investigates whether complex concurrent topologies, which are currently not supported by static verification techniques, are used in practice.

We found that most projects contain programs for which it is not possible to determine the number of threads at compile-time. However, most projects use a finite number of channels.

Synopsis. In Section II, we present the main features of the message passing fragment of the Go programming language. In Section III, we describe our methodology, including our data selection and our Go program analyser. In Section IV, we present the results of our study, answering our four research questions. In Section V, we discuss the limitations of our study. We discuss related work in Section VI and give concluding remarks in Section VII. Our tool-chain [3] and experimental data [4] are available online.

II. Message Passing Concurrency in Go

Go is a statically typed imperative programming language with a particular emphasis on concurrency. Its main distinguishing features are lightweight threads (goroutines) and communication channels. The synchronisation mechanism over communication channels is inspired by theoretical models of concurrency such as Hoare's communicating sequential processes (CSP) [7], and reminiscent of Milner's calculus of communicating systems (CCS) [17] and π-calculus [18].

Go programs consists of packages (i.e., folders) which contain `.go` files. Each `.go` file contains a package declaration, a list of imports, a list of (package-scoped) variables, a list of type declarations, and a list of functions. We give a typical example of a Go program in Listing 1 which consists of two functions: `worker` and `main`. Function `worker` takes three parameters: an integer and two channels. Channel `x` is declared as a channel on which `worker` can only *send* integers, while channel `y` can only be used to *receive* integers. Channel direction annotations are enforced statically, but can be omitted. The body of `worker` consists of an infinite `for` loop containing a select statement offering two choices: either send an integer `j` on channel `x`, or receive a message from channel `y`. The function loops if it can send on `x`, or terminates if it can receive on `y`. The semantics of `select` statements is non-deterministic when more than one action is enabled.

Function `main` starts the program by creating two channels (Lines 10-11). It then spawns 30 concurrent instances of the `worker` function (or goroutine). The main thread then reads (and prints) 10 messages from channel `a`, see Lines 17-18, on which `workers` send messages. Once the main thread is done reading and printing, it `closes` channel `b` (on which the `workers` are listening). Closing a channel in Go has the effect

```
func worker(j int, x chan<- int, y <-chan int) {
  for {
    select {
           case x <-j:        // send
           case   <-y: return // receive
    }
  }
}
func main() {
  a := make(chan int)
  b := make(chan int)

  for i := 0; i < 30; i++ {
      go worker(i, a, b)
  }
  for i := 0; i < 10; i++ {
      k := <-a                  // receive
      fmt.Println(k)
  }
  close(b)
}
```

Listing 1. Concurrent workers.

of enabling any subsequent receive action on this channel (a read operation on a closed channel returns a default value, e.g., 0 for integers). Any attempt to invoke a close or send primitive on a closed channel triggers an exception and crashes the program. In the case of the program in Listing 1, closing channel `b` has the effect of terminating all `worker` goroutines.

We describe further message passing oriented constructs below. Communication channels are *synchronous* by default, i.e., both send and receive actions are blocking. It is possible to give a capacity at channel creation, e.g.,

```
ch := make(chan string, 256)
```

in which case send actions are not blocking until the (asynchronous) channel has reached its capacity (256 here).

Channels may be ranged over using the *range over channel* construct as in Listing 2. This program creates a buffered channel which can hold up to 10 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. Channels can only carry objects of the type declared at creation time. These can be simple (e.g., integer, boolean) or complex (e.g., structs) types, channels can be transmitted over channels too.

```
    msgs := make(chan int, 10)
    msgs <- 1
    msgs <- 3
    close(msgs)
    for m := range msgs { fmt.Println(m) }
```

Listing 2. Range over channel.

Select statements may include a (single) default case which is selected when no other case is enabled. Select statements with a default case are not blocking, see the example below.

```
select { case   <-x   : fmt.Println("received")
         case y <-42  : fmt.Println("sent")
         default :      fmt.Println("default") }
```

```
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 }
```

Listing 3. Concurrent prime sieve.

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.

Listing 3, adapted from [5], gives an example of a more complex concurrent program implementing a concurrent version of the Sieve of Eratosthenes (an algorithm to compute all prime numbers under a bound). 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 (resp. a channel) is spawned within a `for` loop, see Line 22 (resp. 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.

III. 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. Table I gives an overview of the total number of projects we have analysed.

TABLE I
GENERAL INFORMATION ABOUT THE PROJECTS

Total number of visited projects	900
Number of analysed projects	865
Number of message passing projects	661
Number of median-sized projects	32

We have visited 900 projects in total and thoroughly analysed 865 of them, totalling 35 million (physical) lines of code. Part of our analysis focuses on two sub-groups: 661 projects which contain at least one channel and 32 of similar sizes.

A. Data Selection

The goal of our study is to investigate how and how much developers use the message passing concurrency features of Go in all application domains, hence we have selected a wide range of projects that do not necessarily feature concurrency-related aspects. Figure 1 gives an overview of the selection procedure. First, we have selected the *900 most popular Go projects* on GitHub, according to the number of *stars* associated with these projects. The number of stars generally reflects how many people appreciate or are interested in a project [1]. The selection was made in August 2018 when the star counts of the selected projects ranged from 822 to 49765 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 which do not contain human-made applications, e.g., tutorials, textbooks, generated code, etc. 35 such projects were removed. For each of these 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`).

B. Program Analysis

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

- The channel creation primitive, `make(chan T)`, with or without a capacity, e.g., Lines 10 and 11 of Listing 1. We also record the capacity and the type `T` of each channel to determine whether it is asynchronous and/or whether the channel is used to carry other channels.
- The basic channel-based primitives: send, receive, and select. As well as the close primitive (e.g., Line 20 of Listing 1) and the range-over-channel statement (e.g., Line 5 of Listing 2).

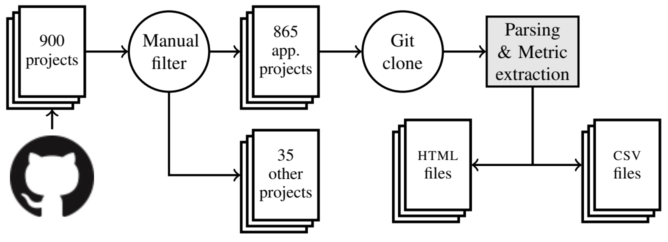


Fig. 1. Process of the empirical study.

- The spawning of a goroutine (e.g., Line 16 of Listing 3). We consider occurrences of goroutine and channel creations in `for` loops as special cases (e.g., Lines 22 and 21 of Listing 3).
- The aliasing (or assignment) of a channel within a `for` loop, as in Line 23 of Listing 3.
- The usage of channel direction annotations in formal parameters, as in Line 1 of Listing 1.

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

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 [4].

C. Project Sizes

To compare the level of intensity of message passing concurrency in projects of significantly different size and structure, we present some of our measurements relative to the number of *physical lines of code* (PLOC) using the CLOC command [2] (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 which contain at least one of the concurrency features described in Section III-B. 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.

IV. QUANTITATIVE ANALYSIS

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

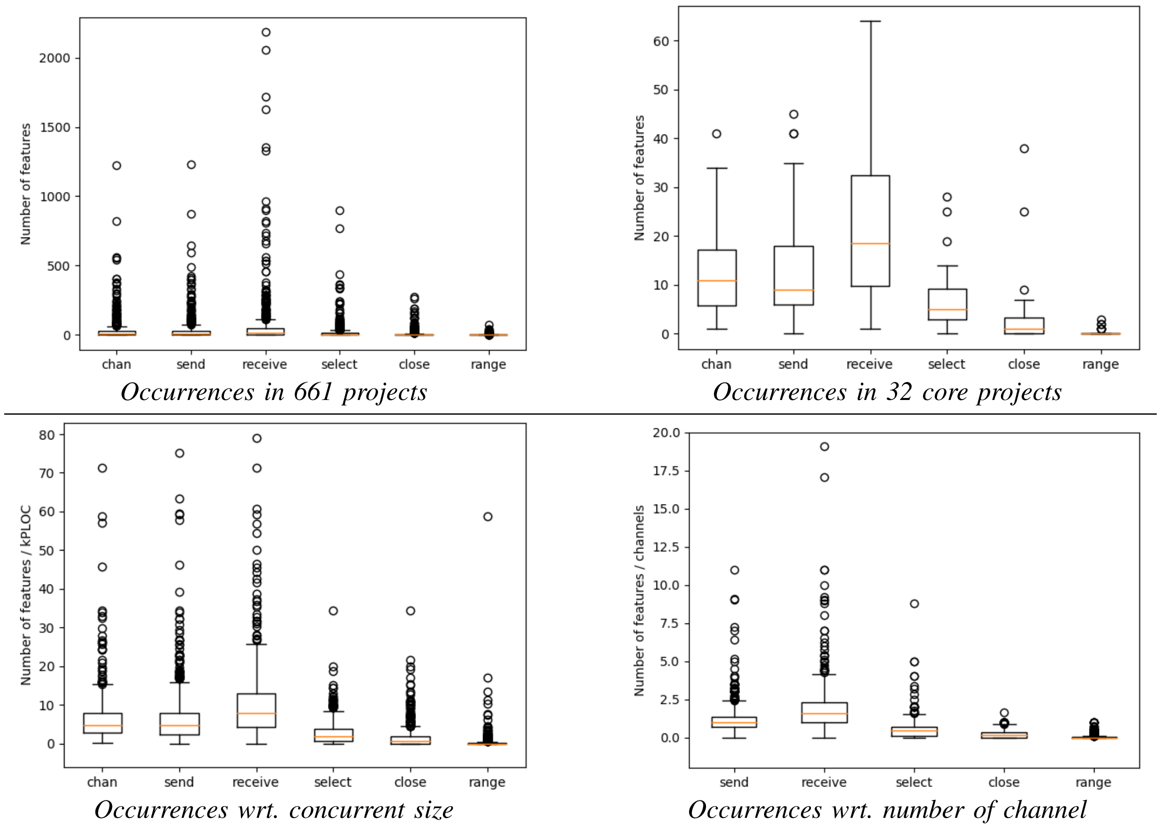


Fig. 2. Box plots for **RQ1:** *How often are messaging passing operations used in Go projects?*

TABLE II
PROJECTS USING MESSAGE PASSING (OUT OF 865 PROJECTS)

Feature	projects	proportion
chan	661	76%
send	617	71%
receive	674	78%
select	576	66%
close	402	46%
range	228	26%

RQ1: *How often are messaging passing operations used in Go projects?*

Our tool-chain is used to collect occurrences of Go's native message passing primitives. Table II summarises our findings wrt. occurrences of message passing operations in the 865 projects we have analysed. We note that 204 projects out of 865 ($\sim$24%) do not create any communication channels. We observe that send, receive, and select constructs appear in more than 66% of the projects. The receive primitive is the most frequently used message passing operation, with 78% of the projects containing at least one instance. This primitive is also used to model delays and timeouts, which explains why the number of projects with receive primitives 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.")
```

In the rest of this section, we focus on those 661 projects which contain at least one channel creation primitive. We present both absolute and relative measurements. To give two distinct perspectives on the relative occurrences of message passing primitives, we present results with respect to the *concurrent size* of projects (see Section III-C) and the number of occurrences of the *channel creation* primitive.

Absolute measurements: Figure 2 (top left) and Table III give the average, standard deviation and five-number summary of the number of occurrences of message passing primitives in the 661 projects which contain at least one channel creation. On average, the projects we have analysed contained 33.62 occurrences of a channel creation primitive (with a median of 9). The average number of occurrences of send (resp. receive) primitives is 36.37 (resp. 69.44) with a median of 10 (resp. 14). Select statements are the third most used synchronisation construct with an average of 20.25 selects (and a median of 3). This is followed by the close primitive with an average of 9.21 (and a median of 0.55). Table III also shows that the range over channel construct is not used intensively. On average, the projects we have analysed contained only 1.45 such constructs (with a median of 0). Table IV studies 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 $\sim$2 branches on average. Over the 13403 select statements we have analysed, 4116 (30%) included a default branch.

TABLE III
ABSOLUTE OCCURRENCES IN 661 PROJECTS.

Features	mean	std	min	25%	50%	75%	max
chan	33.62	86.17	1	3	9	27	1225
send	36.37	90.79	0	2	10	31	1229
receive	69.44	198.76	0	4	14	48	2183
select	20.25	63.94	0	1	3	14	901
close	8.98	26.13	0	0	1	5	275
range	1.44	4.81	0	0	0	1	72

TABLE IV
NUMBER OF BRANCHES IN SELECT STATEMENTS

	mean	std	min	25%	50%	75%	max
branches	2.15	0.82	0.00	2.00	2.00	2.00	41.00

The top 3 projects in terms of absolute numbers of channel-oriented features are `juju` (concurrent size = 86 kPLOC), `cockroach` (concurrent size = 146 kPLOC), and `go` (concurrent size = 122 kPLOC). The `juju` project holds the well-known cloud infrastructure management framework. This project contains the highest number of receive primitives (2183). It contains 820 channel creations, 876 send primitives, and 901 select statements. It has a ratio of receive to channel creation primitives of 2.67 and a ratio of receive to send primitives of 2.50. These high ratios can be explained in part by the fact that `juju` has the highest number of select statements amongst the projects we have analysed. Select cases are generally guarded by receive primitives. This project contains 17 select statements with 5 or more branches (with one select having 19 branches). Given the nature of the software, it is not too surprising that it relies heavily on concurrency-related features, e.g., to monitor applications and respond to events. The `go` project contains the Go compiler, standard library, and runtime. This project includes a large number of concurrency-related features, i.e., 1225 channel creations, 1229 send primitives, 1719 receive primitives, and 340 select statements. It has the largest number of channel creation and send primitives. Finally, `cockroach`, a cloud-native SQL database, has 564 channel creation, 591 send, 1355 receive primitives, and 364 select statements. It is larger in terms of concurrent size than `juju` and `go`, but smaller in terms of overall number of physical lines of code (620k PLOC for `cockroach`, 635k for `juju`, and 1340k for `go`).

To visualise the usage of message passing primitives in absolute terms over similarly sized projects, we selected the projects whose size falls within 10% of the median *concurrent size* $|P|$ of all 661 projects. The median concurrent size of our sample is 1.8 kPLOC, hence the *core projects* consists of projects whose size is between 1.7 and 2.1 kPLOC. Figure 2 (top right) and Table V summarise our results. We observe that there are generally more receive primitives than channel creation and send primitives. Secondary constructs such as close and range over channels occur less frequently, on average.

Within these 32 core projects, `RxGo` (an API that provides support for reactive programming) is the project with the most channel creation and close primitives (41 and 38, respectively). It contains the second largest number of receive primitives (41). The project with the second highest number of channel creation and close primitives is `surgemq`. The `surgemq` project provides a high performance implementation of a messaging protocol (MQTT) for IoT devices. This project contains 2 send primitives, 64 receive primitives, and 34 channel creation primitives. It also contains the highest number of select statements (28), all of which have 2 branches.

TABLE V
ABSOLUTE OCCURRENCES IN 32 CORE PROJECTS.

Features	mean	std	min	25%	50%	75%	max
chan	13.62	10.35	1	5.75	11.00	17.25	41
send	14.06	12.49	0	6.00	9.00	18.00	45
receive	23.22	16.19	1	9.75	18.50	32.50	64
select	7.25	6.74	0	3.00	5.00	9.25	28
close	3.62	7.83	0	0.00	1.00	3.25	38
range	0.34	0.75	0	0.00	0.00	0.00	3

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 features. For each project P, we divide the number of occurrences of each message passing feature by $|P|$. Figure 2 (bottom left) and Table VI summarise our findings (the box plot is capped at y=80 for readability). On average, we observe that message passing primitives are used intensively in concurrency-related files. We find 6.34 channels for every 1000 physical lines of code (with a median of 4.69). The relative average number of occurrences for send and receive primitives is 6.65 and 10.31, respectively. The other primitives are used significantly less intensively.

Disregarding the small projects which contain very few features, the three projects with the highest number of channel-oriented primitives relative to their concurrent size are: `doozerd` (a consistent distributed data store), `go-memdb` (an in-memory database), and `anaconda` (a Go client library for the Twitter API). The `doozerd` project has the largest number of send primitives relative to its concurrent size (75.3 per kPLOC in concurrency-related files). The `go-memdb` project has the largest number of receive (123.7) and close (21.50) primitives relative to its concurrent size. Therefore, on average a receive (resp. close) primitive occurs almost every 8 (resp. 46) physical lines of code in concurrency-related files. Finally, the `anaconda` project contains the highest number of channel creation primitives relative to its concurrent size (56.4), i.e., a channel creation every 18 physical lines of code in concurrency-related files, on average. Theses numbers can be explained by the frequent occurrence of functions similar to the one below:

```
func (a TwitterApi) GetFriendships() (...) {
  responseCh := make(chan response)
  a.queryQueue <-query{...,responseCh}
  return <-responseCh  }
```

which sends a query to the Twitter API together with a channel on which the response should be sent.

TABLE VI
RELATIVE OCCURRENCES WRT. CONCURRENT SIZE IN 661 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

TABLE VII
RELATIVE OCCURRENCES WRT. CHANNELS IN 661 PROJECTS.

Features	mean	std	min	25%	50%	75%	max
send	1.26	2.92	0.00	0.67	1.00	1.36	71.40
receive	2.08	3.36	0.00	1.00	1.56	2.28	66.50
select	0.57	1.33	0.00	0.12	0.44	0.71	30.50
close	0.22	0.28	0.00	0.00	0.13	0.36	1.62
range	0.06	0.15	0.00	0.00	0.00	0.04	1.00

Measurements relative to number of channels: Our second relative measurements are made relative to the number of occurrences of channel creation primitives in each project. Hence, we divide the number of occurrences of each primitive such as send, receive, etc. by the number of occurrences of `make(chan T)`. This measurement gives us an approximation of the number of operations invoked on each channel. Figure 2 (bottom right) and Table VII summarise our results (the box plot is capped at y=20 for readability). On average, there are 1.26 send primitives per channel creation (with a median of 1); while there are 2.08 receive primitives per channel creation (with a median of 1.56). The slightly higher number of receive primitives can be explained by the fact that on average there is approximately a select for every other channel. In turn, select statements have more than two branches on average, see Table IV, and they are generally guarded by receive primitives.

Two projects stand out with respect to the number of channel-oriented primitives 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. The `node_exporter` project contains several instances of send primitives sending several (53) hard-coded variations of a struct. These two examples are extreme cases of the operation to channel ratio. However, as Figure 2 (bottom right) and Table VII show, 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.

***RQ2:** How is concurrency spread across Go projects?*

Go is renowned for its support for concurrent programming, but is it the case that most of the source code is concurrent? In this question, we study the proportion of a Go project which is related to concurrency. We consider three different measures

RQ1: We found that 76% of the projects we have analysed use communication channels. The receive primitive is the most commonly used operation. On average, the number of primitives per channel is low, suggesting that channels are used for simple synchronisation protocols.

TABLE VIII
PROPORTION OF CONCURRENCY IN 661 PROJECTS

Measure	mean	std	min	25%	50%	75%	max
size	31.05	23.83	0.03	11.97	25.20	45.32	100
package	44.22	28.67	0.93	21.92	37.50	55.56	100
file	20.11	17.67	0.34	8.00	15.33	26.32	100

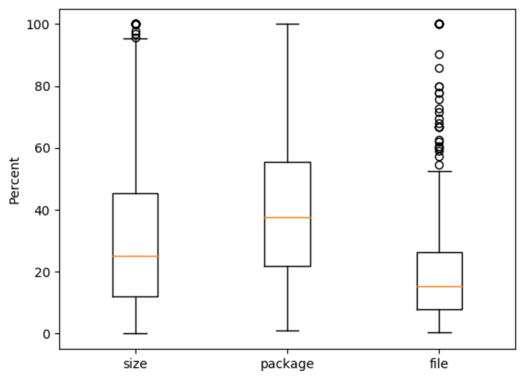
for project sizes: the number of physical lines of code, the number of packages, and the number of files. Figure 3 (left) and Table VIII summarise our results for the 661 projects which contain at least one channel creation primitive. The first line of the table gives the ratio of *concurrent size* $|P|$ to the total number of physical lines of code in projects. The table shows that, on average 31.05% of the size of projects is dedicated to concurrency (with a median of 25.20%). The second line of the table gives the ratio of number of *packages* featuring concurrency to the total number of packages. On average, 44.22% percent of a project's packages contain at least one concurrency feature (with a median of 37.50%). The third line of the table gives the ratio of number of *files* containing some concurrency features to the total number of files. On average, 20.11% percent of a project's files contain at least one concurrency feature (with a median of 15.33%). Figure 3 (right) and Table IX give the results of the same analysis on the 32 core projects described above. We note that both populations give similar results.

RQ2: We observed that, on average, just under half of the packages of the Go projects we analysed contain concurrency features, while around 20% of files contain concurrency-related features. We observed that concurrency-related files are generally larger than files containing only sequential code.

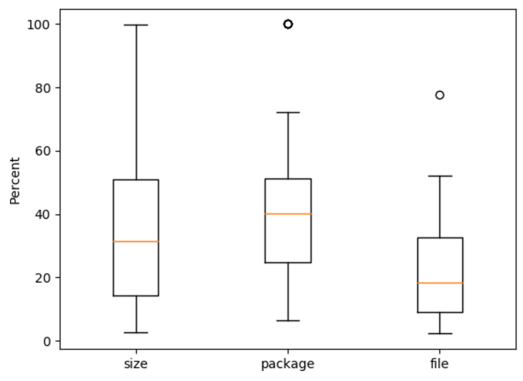
Tables VIII and IX suggest that files which contain concurrency-related features tend to be larger (wrt. PLOC) than files containing sequential code only. For instance, the `RxGo` project (included in the 32 core projects) has a concurrent size to overall project size ratio of 85%, while its ratio of number of concurrency-related files to overall number of files is 45%

TABLE IX
PROPORTION OF CONCURRENCY IN 32 CORE PROJECTS

Measure	mean	std	min	25%	50%	75%	max
size	36.57	26.03	2.69	14.30	31.38	50.85	99.70
package	45.16	28.51	6.41	24.79	40.03	51.14	100
file	23.29	18.63	2.23	9.01	18.38	32.59	77.78



Concurrency spread in 661 projects



Concurrency spread in 32 core projects

Fig. 3. Box plots for **RQ2:** *How is concurrency spread across Go projects?*

(9/20 files). This trend can be explained by the fact that files containing only sequential code are often used for declarative purposes only (e.g., to declare constants, global variables, and structs). Overall, our study suggests that concurrency-related code is usually clustered in a subset of the source code. Tables VIII and IX show that the percentage of files containing at least one concurrency-related feature over all files is close to 20%. For instance, the concurrency-related files to overall files ratio is 13% for `juju` (5730 features overall) and 9.1% for `go` (6772 features overall).

RQ3: *How common is the usage of* asynchronous *message passing in Go projects?*

Go offers two types of channels: synchronous (default) and asynchronous. Both send and receive operations are blocking on synchronous channels, while send operations are not blocking on asynchronous channels, as long as the channel has not reached its maximal capacity. 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 III to collect occurrences of channel creation primitives and record channel bounds, whenever possible. Because the capacity of a channel might only be known at runtime, we consider that some channels have an "unknown bound". Table X lists the number of occurrences of each type of channels. The projects we have analysed contain more than 22k channels. For a large majority (94%) of the channels, we were able to determine their bounds statically: either synchronous (61%) or a non-zero capacity known at compile time (33%), i.e., a hard-coded integer or a constant.

Table XI 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 5 is uncommon. We note that out of the 7229 asynchronous channels with statically known bounds, 3237 channels were located in test files (45%). A few projects use channels with very large capacity to simulate unbounded asynchrony. For instance, the

TABLE X
COMMUNICATION CHANNELS IN 661 PROJECTS

Type	occurrences	proportion
All channels	22226	100%
Channels with known bounds	20868	94%
Synchronous channels	13639	61%
Asynchronous channels (known)	7229	33%
Channels with unknown bounds	1358	6%

TABLE XI
KNOWN SIZES OF ASYNCHRONOUS CHANNELS

	mean	std	min	25%	50%	75%	max
size	1193.62	29838.20	1	1	1	5	10^6

project `gometalinter` contains four channels of size 10^6 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 `netstack` and `gonet` projects.

RQ3: We observed that synchronous channels are the most commonly used channels (61%). Whenever asynchronous channels are used, they are generally created with a *statically known* bound, which is less than or equal to 5 in 75% of the cases.

RQ4: *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.

For instance, the concurrent prime sieve program from Section II (Listing 3) has a complex concurrent topology because it creates an unknown number of goroutines which are linked by distinct channels.

a) Goroutine creation: The first part of Table XII summarises our analysis on the frequency of different patterns of goroutine creations in the 865 projects. The table shows

TABLE XII
FREQUENCY OF CONCURRENCY PATTERNS IN 865 PROJECTS

Feature	projects	proportion
`go`	711	82%
`go` in (any) `for`	500	58%
`go` in bounded `for`	172	20%
`go` in unknown `for`	474	55%
`chan` in (any) `for`	111	13%
`chan` in bounded `for`	19	2%
`chan` in unknown `for`	103	12%
channel aliasing in `for`	14	2%
channel in slice	31	4%
channel in map	8	1%
channel of channels	49	6%

TABLE XIII
KNOWN BOUNDS OF `FOR` LOOPS CONTAINING `GO`

	mean	std	min	25%	50%	75%	max
bound	280.53	1957.50	1	5	10	100	50000

that 82% of the projects we have analysed contain at least one thread creation (i.e., the keyword `go`) and 58% contain at least one occurrence of a thread creation within a `for` loop (i.e., `go` in (any) `for`). We distinguish between thread creation within a *bounded* `for` loop, as in Line 13 of Listing 1 and *unknown* `for` loop, as in Line 18 of Listing 3. 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. However, out of 918 occurrences of a creation of a goroutine within a bounded `for`, 788 of them were located in a `<file>_test.go` file (86%). Table XIII summarises the size of the bounds we have encountered and the top of Table XIV summarises the relative occurrences of patterns in projects which contain at least one occurrence of such a pattern.

b) Channel creation: The second part of Table XII gives the proportion of projects where channels are created within a `for` loop. The second part of Table XIV summarises the *relative* number of occurrences of these patterns (for which there are at least 30 occurrences) in projects which contain at least one occurrence of such a pattern. Observe that channel creation within a `for` loop is much less common that tread spawning. Again, we distinguish between channel creations within *bounded* `for` loops as these could be unfolded as part of a static analysis. Only 13% 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 (2%). 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 Listing 3). Channel aliasing can be used to create a potentially unbounded chain of linked threads as in the concurrent prime sieve program (Listing 3). We have manually analysed all occurrences of “channel aliasing in `for`” in our sample and found *no* occurrence resembling the pattern in Listing 3. In fact, our investigation revealed that most

TABLE XIV
RELATIVE OCCURRENCES WRT. CONCURRENT SIZE.

Patterns	mean	std	min	25%	50%	75%	max
`go`	9.08	7.76	0.16	4.4 9	7.14	11.32	71.43
`go` in (any) `for`	2.51	4.72	0.03	0.64	1.24	2.43	67.51
`go` in unknown `for`	2.29	4.74	0.03	0.53	1.11	2.11	67.51
`go` in bounded `for`	1.00	1.76	0.01	0.15	0.41	0.99	10.81
`chan` in (any) `for`	0.60	1.68	0.02	0.08	0.19	0.46	14.71
`chan` in unknown `for`	0.61	1.73	0.01	0.08	0.20	0.46	14.71

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 records contain channels).

c) 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. The last part of Table XII shows that only 4% (resp. 1%) of the projects we have analysed use slices (resp. maps) to directly store channels. The last line of Table XII shows that very few projects (6%) 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 [18].

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 1 of Listing 1. Channel direction annotations restrict the concurrent topologies: they enforce channels to be unidirectional. We found that in 45% of the cases channel formal parameters had a specified direction.

RQ4: 58% of the projects we have analysed include thread creations within `for` loops, a pattern which is not (soundly) supported by existing static verification frameworks. Most projects (87%) use a *bounded* number of communication channels.

V. LIMITATIONS OF THE STUDY

The main limitations of our study are related to data selection and metric extraction. Extracting data from GitHub involves the risks of including repositories which contain personal or inactive projects, or are used as free storage [9]. For this study, we are interested in *any* code written as part of a Go program, hence inactive or personal projects do no pose a particular problem. Repositories used as free storage are unlikely to attract more than 800 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 channels 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 send 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 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 [3], [4].

VI. Related Work

To the best of our knowledge, this paper is the first empirical study of programming in Go, consequently we consider a wider context of related work. Several studies have investigated the usage of concurrency constructs in different programming languages, using publicly available source code. Marinescu [15] 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. [30], [31] 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. [23], [29] 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 [20] 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 et al. [28] 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. Landam et al. [10] 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. [24] 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. [14] 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 et al. [22] 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.

VII. Conclusions & Future Work

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. We have discovered that concurrency-related features are generally located in a limited parts of Go projects, which contrasts with existing verification approaches which consider programs as a whole. This suggests that static analyses dedicated to concurrency may be done in a modular way on smaller parts of projects. The most important challenge for future static verification of message passing Go programs concerns functions which spawn a statically unknown number of goroutines. We have shown that this patterns appears frequently in Go projects, and therefore should be supported by future verification frameworks. Other potentially un-tractable topologies involving an unbounded number of channels or channels carrying other channels are much less common.

We plan to extend our survey to compare the usage of message passing in languages such as Go, Rust, and Erlang which all *natively* provide message passing facilities. Additionally, we would like to study whether programming with message passing concurrency is more or less error-prone than programming with, e.g., locks or barriers.

Acknowledgements.

We thank Nicholas Ng, Stefan Marr, Bernardo Toninho, and the referees for comments on earlier versions of this work.

References

[1] Hudson Borges and Marco Tulio Valente. What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129, 2018.

[2] Al Danial. Count lines of code. https://github.com/AlDanial/cloc.

[3] Nicolas Dilley and Julien Lange. Go project analyser. http://github.com/nicolasdilley/gocurrency_tool, 2018.

[4] Nicolas Dilley and Julien Lange. Survey data. http://www.cs.kent.ac.uk/~jl703/go-survey, 2018.

[5] Golang. The Go playground — a concurrent prime sieve. https://play.golang.org/p/9U22NfrXeq.

[6] Golang. The Go programming language. https://golang.org/, 2018.

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

[8] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016.

[9] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela E. Damian. The promises and perils of mining GitHub. In *MSR 2014*, pages 92–101, 2014.

[10] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. Challenges for static analysis of Java reflection: literature review and empirical study. In *ICSE 2017*, pages 507–518, 2017.

[11] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off Go: liveness and safety for channel-based programming. In *POPL 2017*, pages 748–761, 2017.

[12] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in go using behavioural types. In *ICSE 2018*, pages 1137–1148, 2018.

[13] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *POPL 2015*, pages 221–232, 2015.

[14] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS 2008*, pages 329–339, 2008.

[15] Cristina Marinescu. An empirical investigation on MPI open source applications. In *EASE 2014*, pages 20:1–20:4, 2014.

[16] Jan Midtgaard, Flemming Nielson, and Hanne Riis Nielson. Process-local static analysis of synchronous processes. In *SAS 2018*, pages 284–305, 2018.

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

[18] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[19] Nicholas Ng and Nobuko Yoshida. Static deadlock detection for concurrent Go by global session graph synthesis. In *CC 2016*, pages 174–184, 2016.

[20] Semih Okur and Danny Dig. How do developers use parallel libraries? In *FSE 2012*, page 54, 2012.

[21] Rob Pike. Go proverbs. https://www.youtube.com/watch?v=PAAkCSZUG1c, 11 2015.

[22] Gustavo Pinto, Weslley Torres, and Fernando Castor. A study on the most popular questions about concurrent programming. In *PLATEAU 2015*, pages 39–46, 2015.

[23] Gustavo Pinto, Weslley Torres, Benito Fernandes, Fernando Castor Filho, and Roberto Souto Maior de Barros. A large-scale study on the usage of Java's concurrent programming constructs. *Journal of Systems and Software*, 106:59–81, 2015.

[24] Amir Saboury, Pooya Musavi, Foutse Khomh, and Giulio Antoniol. An empirical study of code smells in JavaScript projects. In *SANER 2017*, pages 294–305, 2017.

[25] Kai Stadtmüller, Martin Sulzmann, and Peter Thiemann. Static trace-based deadlock analysis for synchronous Mini-Go. In *APLAS 2016*, pages 116–136, 2016.

[26] Martin Sulzmann and Kai Stadtmüller. Trace-based run-time analysis of message-passing go programs. In *HVC 2017*, pages 83–98, 2017.

[27] Martin Sulzmann and Kai Stadtmüller. Two-phase dynamic analysis of message-passing go programs based on vector clocks. In *PPDP 2018*, pages 22:1–22:13, 2018.

[28] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why do Scala developers mix the actor model with other concurrency models? In *ECOOP 2013*, pages 302–326, 2013.

[29] Weslley Torres, Gustavo Pinto, Benito Fernandes, João Paulo Oliveira, Filipe Alencar Ximenes, and Fernando Castor. Are Java programmers transitioning to multicore?: a large scale study of Java FLOSS. In *SPLASH 2011*, pages 123–128, 2011.

[30] Di Wu, Lin Chen, Yuming Zhou, and Baowen Xu. An empirical study on C++ concurrency constructs. In *ESEM 2015*, pages 257–266, 2015.

[31] Di Wu, Lin Chen, Yuming Zhou, and Baowen Xu. An extensive empirical study on C++ concurrency constructs. *Information & Software Technology*, 76:1–18, 2016.