Automated Verification of Go Programs via Bounded Model Checking

Nicolas Dilley
Department of Computing
University of Kent
Canterbury, United Kingdom
nd315@kent.ac.uk

Julien Lange
Department of Computer Science
Royal Holloway, University of London
Egham, United Kingdom
julien.lange@rhul.ac.uk

***Abstract*—The Go programming language offers a wide range of primitives to coordinate lightweight threads, e.g., channels, waitgroups, and mutexes — all of which may cause concurrency bugs. Static checkers that guarantee the absence of bugs are essential to help programmers avoid these costly errors before their code is executed. However existing tools either miss too many bugs or cannot handle large programs. To address these limitations, we propose a static checker for Go programs which 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 a detailed presentation of the extraction algorithm from Go programs to 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.**

***Index Terms*—Go, concurrency, static verification, behavioural types, model checking**

I. Introduction

Developing concurrent software is particularly difficult because concurrency-related bugs are often difficult to detect. Typically these bugs 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).

Message-passing concurrency, as supported natively by the Go programming language, offers a higher level of abstraction than traditional shared memory-based concurrency. However bugs are still a common problem in message-passing software [31] and it is thus essential to develop reliable *static* checkers that can rule out these bugs. Static verification has the advantage of ruling out bugs for *every* execution and to do this *before* the code is executed and released.

Amongst recent works on static checkers for Go, we distinguish those that aim for soundness, e.g., using a behavioural type approach [5], [7], [20], [21], [25], and those that aim for completeness, e.g., GCatch [22]. Existing 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. Additionally, it is not easy to predict the type of bugs it misses.

In this work, we build on the behavioural types approach first formalised in [20], where models that represent the communication structure (behavioural types) are extracted from Go programs. These models over-approximate their programs and can be verified using an off-the-shelf model checker.

Our work builds on our earlier prototype [5] which uses Promela as a behaviour type language. Our extension 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) To deal with large codebases, we divide Go programs into independently verifiable components. This allows us to verify a large project like Kubernetes [19] (more than 3 million LoC) in 26 minutes. Most projects are checked under 4 minutes.

(3) Our approach supports programs that coordinate over the three main concurrency primitives (channels, waitgroups, mutexes), and 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 a bug our tool discovered in the wild with Example 1.

Example 1. The program in Figure 1 is adapted from code found on the GitHub repository of `trillian` [11], a verifiable data store developed at Google. At line 10, function `preload()` spawns $|$`trees`$|$ worker goroutines which 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

```
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 }
```

Fig. 1: Example of a blocking bug, adapted from [11].

(`wg`). When all goroutines have invoked `wg.Done()`, operation `wg.Wait()` succeeds, and channel `ch` is `close`d.

After spawning $|\texttt{trees}|+1$ 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 1 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 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.

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` is called often with the "wrong" arguments, this would affect significantly the memory footprint of the program.

Programs like Example 1 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 1. 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 that helps distinguishing false alarms from real bugs.

Contributions § III describes a novel algorithm to extract *parameterised* behavioural types from Go code. § IV describes a bounded model checking technique to verify these behavioural types. We have implemented these algorithms in a tool, GOMELA [6], outlined in § V. In § VI, we give a thorough experimental evaluation of GOMELA against related work [7], [22]. Out of 78 real-world buggy programs, we show that GOMELA detects 46 bugs (at least 1.54× more than other tools). Additionally, we demonstrate the scalability of our approach using 99 most starred Go repositories from GitHub.

II. Preliminaries

We review key aspects of the Go programming language, and give a brief overview of Spin and Promela (the model checker and language we use to verify Go programs).

A. 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 which 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. The standard library also offers two popular concurrency primitives: waitgroups and mutexes.

Figure 2 gives an overview of the control-flow constructs and concurrency operations of interest 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). 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.

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 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 $ch <- e$ sends e on ch, while $<- ch$ receives from ch. Invoking **`close`**(ch) closes ch. Receiving on a closed

	$f(\overline{a})$	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
○	**for** x $:=$ **range** ch $\{\overline{s}\}$	Iteration over channel ch
	$ch :=$ **make**(**chan** T, e)	Declare a chan. with capacity e
○●	ch <- e	Send e over ch
○	<- ch	Receive on channel ch
●	**close**(ch)	Close 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

Fig. 2: Key statements in Go, some may be blocking (○) and/or may trigger a run-time error (●).

channel is non-blocking, but sending on, or closing, a closed channel triggers a runtime error. 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. A **select** statement may contain a default branch which is executed if all other branches are blocked.

Instruction **var** wg sync.WaitGroup 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. A waitgroup whose counter becomes negative triggers a runtime error.

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.

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 2. 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 1.

Safety bugs occur when an operation is unexpectedly invoked on a concurrency primitive (and triggers a run-time error), e.g., sending on a closed channel causes a run-time error. Operations that may trigger a run-time error are marked with

```
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 }
```

Fig. 3: Negative counter bug, adapted from [18].

● in Figure 2. Observe that all three concurrency primitives we consider can trigger such errors.

Example 2. Figure 3, adapted from kubernetes [18], 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 may trigger a run-time error ("panic: sync: negative WaitGroup counter").

Focus of this work: For the sake of space and clarity, the core of this paper focuses on the subset of Go identified by Figure 2. We explain how our tool handles additional features (e.g., structs, methods, and anonymous functions) in § V.

Our aim is to develop a technique that is sound for a well-understood subset of Go. The language features our approach does *not* currently support are: programs that recursively spawn goroutines or create concurrency primitives, virtual method calls, higher-order functions, mutable/mutated concurrency primitives (e.g., channel variables that are reassigned), and collections containing concurrency primitives.

B. Promela as a behavioural type language

The crux of our technique is to over-approximate Go programs with behavioural types [20] — 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. (The model extraction we present in § III can easily be adapted to other modelling languages that feature processes communicating over channels.) (ii) While the work in [20] and its extensions [7], [21] 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 [14] 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. Promela processes are declared using **proctype** $f(\overline{p})\{\overline{\mathtt{S}}\}$ where f is the name of the process, $\overline{p}$ is a list of (typed) parameters, and $\overline{\mathtt{S}}$ is a list of Promela statements.

Promela statements and expressions include basic Boolean and arithmetic expressions, as well as declarations and instantiations of variables and structures. 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 $\langle e\rangle$ of a send/receive when it is not relevant.

Promela provides two types of control-flow constructs: loops and branching. Loop **for** $n..m$ $\{\overline{\mathtt{S}}\}$ repeats statements $\overline{\mathtt{S}}$ for $m{-}n{+}1$ iterations. Branching constructs encode (possibly non-deterministic) choices between several behaviours. Promela uses **if** and **do** constructs to encode choices, but we use a graphical (automata/statechart-like) notation in this paper for the sake of readability. The branches of a choice may be guarded or not. A guarded branch is labelled with $[g]$ or $[g]\alpha/\mathtt{S}$ where g is a guard (Boolean expression), α is either a send or receive action, and $\mathtt{S}$ is a Promela instruction. A guarded branch is fireable when its guard g holds and a matching event for α is available (if α is specified). Upon firing, instruction $\mathtt{S}$ is executed. Unguarded branches (labelled with τ) can fire (silently) at any point.

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*, e.g., they cannot spawn infinitely many processes.

III. Extracting Parameterised Models

We describe our approach to extract several models from 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 points to a model that can be 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 detect some partial deadlocks and 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 consists 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.

A. 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:

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

The first two cases deal with integer literals and variables. The third case 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 list 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 $s = ch := \mathbf{make}(\mathbf{chan}\ \mathtt{T}, e)$, then $\mathcal{E}_S(s) \triangleq \mathcal{E}_E(e)$, i.e., the parameters that set the capacity of the channel.
2) If $s = wg.\mathtt{Add}(e)$, 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 $s = \mathbf{for}\ \mathtt{i} := e_1; e_2; r\ \{\overline{s}\}$, we apply heuristics depending on the shape of the loop to identify a variable that can represent the number of iterations. For instance, if $s = \mathbf{for}\ \mathtt{i} := 0; i{<}e; i{+}{+}\ \{\overline{s}\}$, then $\mathcal{E}_S(s) \triangleq \mathcal{E}_E(e) \cup \mathcal{E}_S(\overline{s})$ when e is either a variable, a constant, or an integer literal.
4) If $s = \mathbf{for}\ \mathtt{i,x} := \mathbf{range}\ l\ \{\overline{s}\}$, 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 $s = \mathbf{go}\ f(\overline{a})$, we first extract the concurrency parameter of f. Assume we have $\mathbf{func}\ f(\overline{x\ \mathtt{T}})\ \mathtt{T}\ \{\overline{s}\}$ 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 $\mathcal{E}_E(s)$ recursively following the abstract syntax tree, e.g., if $s = \mathbf{if}\ e\ \mathbf{then}\ \overline{s_1}\ \mathbf{else}\ \overline{s_2}$, then $\mathcal{E}_S(s) \triangleq \mathcal{E}_S(s_1) \cup \mathcal{E}_S(s_2)$.

We say that a for loop is *dynamic* if its body syntactically (and inter-procedurally) contains a statement of the form **go** $f(\overline{a})$. 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 construct models that can be verified

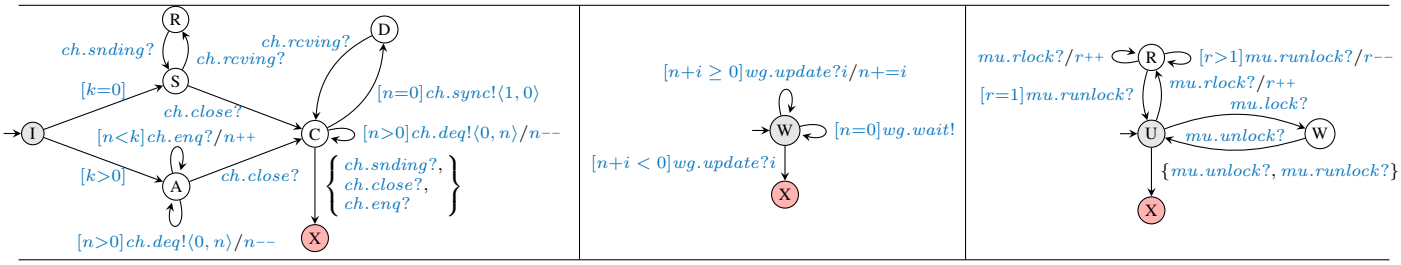


Fig. 4: Primitive processes for channels (left), waitgroups (middle), and mutexes (right).

effectively. Instantiating optional parameters is not necessary, but helps discard false alarms (see § IV).

Example 3. Let $\overline{s}$ be the body of `preload()` from Figure 1. We extract three concurrency parameters from these statements, i.e., $\mathcal{E}_S(\overline{s}) = \{n, trees, runtime.NumCPU\}$. 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).

B. 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 use several Promela channels and variables stored in a structured data type. The declaration of a concurrency primitive in Go is translated to spawning the corresponding primitive process in Promela.

a) Channels: Figure 4 (left) 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 § III-D). 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 § III-D, we show that all function processes are always ready to interact with either the synchronous or asynchronous version of a channel process.

b) Waitgroups: Figure 4 (middle) 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.

c) Mutex: Figure 4 (right) 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.

C. 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 1 will produce one Promela model, whose entry point corresponds to function `preload()`. 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,

```
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)
    // ...
}
```

Fig. 5: Blocking vs. concurrent function calls in Go (top) and their models in Promela (bottom).

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 definitions of function processes include only parameters of their corresponding Go functions that pertain to concurrency. Given a function signature `func ExFunction(..., x T,...)` parameter `x` is abstracted away if `x` is not a concurrency parameter of `ExFunction` 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 5 (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: (i) a `ChannelProcess` (`ch1`) structure which implements the channel process discussed in § III-B; (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 § III-D and § III-E — 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 takes 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, otherwise we spawn a receiving process which will garbage collect the send messages, e.g., `run (ret2?0)`. We illustrate this aspect with the translation of calls to `InnerFunc` in Figure 5.

D. 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 6 (top half) 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 an asynchronous version of ch. As a consequence, both send and receive are modelled as composite Promela operations. A send operation (ch <– e) is 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 (<– 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 actions 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).

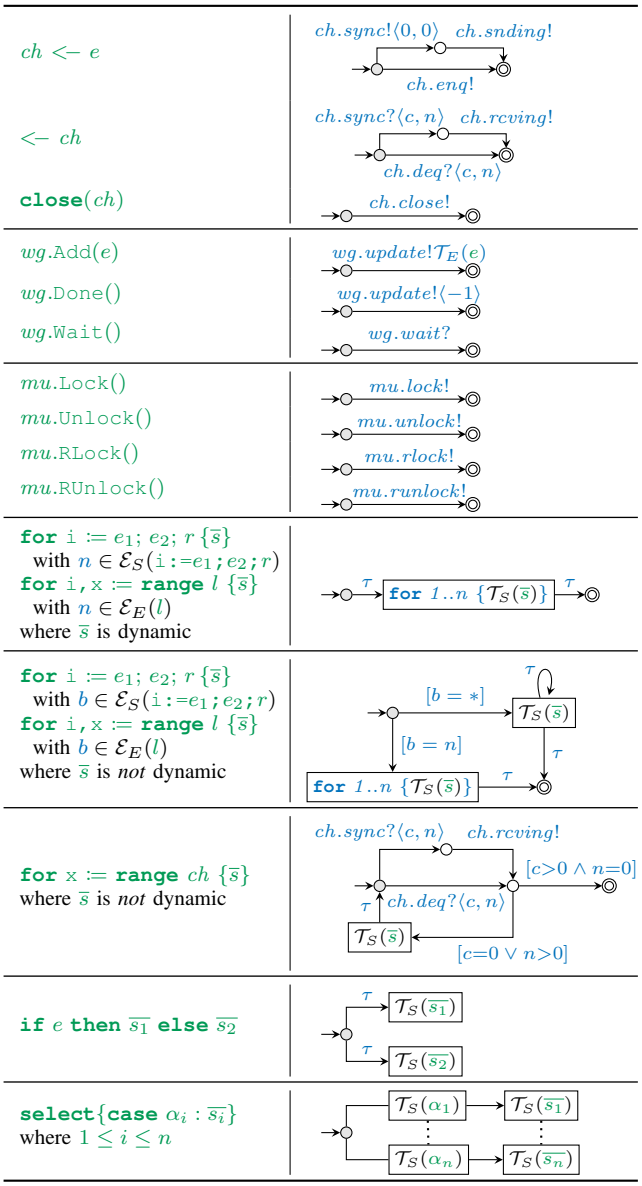


Fig. 6: Overview of the translation function $\mathcal{T}_S(s)$.

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.

E. Control flow and branching constructs

The control flow and branching constructs of Go are over-approximated naturally by Promela's own constructs. Figure 6 (bottom half) 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 § III-A. 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 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 second line of Figure 6). 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). 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/receive operations, we use the construction given in the first two lines of Figure 6. Any branch of the **select** block that is `default` or guarded by a timeout is mapped to an unguarded branch (τ).

Note that in general the translation of loops, if-then-else, and select statements *over*-approximates the behaviour of a Go program, e.g., while only one branch of a Go if-then-else may be taken, both branches are fireable in its generated model.

IV. Verifying models

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

A. 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 6.

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 § II-A. 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. $M \models \phi_{\text{MD}}$ (model deadlock) holds whenever no execution in M leads to a situation where *all* goroutine processes are stuck. Since

```
function verify(S, φ, M)
    x̄ ← mandatory(M)
    ȳ ← optional(M)
    V ← {(ū, v̄) | ū ∈ S^|x̄|, v̄ ∈ ({∗} ∪ S)^|ȳ|}
    while V ≠ ∅ do
        (ū, v̄) ← pickMax(V)
        b ← M[x̄ := ū, ȳ := v̄] ⊨ φ
        if b ∨ ∗ ∉ v̄ then
            Δ(ū · v̄) ← b
            V ← V \ {(ū, w̄) ∈ V | v̄ ⪰ w̄}
        otherwise
            V ← V \ {(ū, v̄)}
    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. $\succeq$.

several models may be extracted from a given program, ϕ_{MD} can identify some partial deadlocks in the source program. $M \models \phi_{\text{CS}}$ (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). $M \models \phi_{\text{WS}}$ (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). $M \models \phi_{\text{MS}}$(mutex safety) holds whenever no execution in M leads to a situation where a mutex process reaches its error state (i.e., an unlocking 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 $P \not\models \phi$.

B. 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 $M[x_1 := 1, x_2 := 2]$.

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 score to indicate a confidence level wrt. results of the verification. The higher the score, the more likely it is that we discovered a real bug.

Example 4. Given the program in Figure 1 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 5. Consider the (correct) program in Figure 7, 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 is 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.

V. Implementation

We have implemented our approach in a tool (GOMELA) which extracts models from Go code, and uses Spin as a backend to automatically verify models up-to user provided

```
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 }
```

Fig. 7: Program with an optional concurrency parameter.

bounds. GOMELA analyses a Go codebase package by package, and relies on Go's `ast` library for the front-end parsing. Unlike [7], [22], our analysis is done on the surface language, rather than its lower-level representation (SSA). In addition to the language constructs listed in Figure 2, our tool handles structures, methods, and anonymous functions.

Go structures that contain concurrency primitives are analysed inter-procedurally and flattened into a list of their primitives. We do not support dynamic data-structures (e.g., linked lists) nor struct embedding.

A method m on struct `S`, **`func`** $(x\ \texttt{S})\ m(\overline{y\ \texttt{T}})\ \texttt{T}\ \{\overline{s}\}$, 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). 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.

GOMELA supports **`break`** statements and **`defer`** statements (when they do not occur under conditionals/loops). Both constructs are implemented using Promela's `goto` instructions.

We have implemented heuristics to speed up the verification. Notably we omit loops that do not contain concurrency-related operations and we model-check all properties (see IV-A) at once instead of one-by-one, c.f. Algorithm 1.

VI. Empirical Evaluation

We structure our evaluation into two research questions that aim at evaluating the real-world usability of our approach.

RQ1: *How does* GOMELA *compare to the state-of-the-art?* To answer this question we compare our tool against GCatch and Godel2 on a dataset of 78 buggy Go programs.

RQ2: *How does* GOMELA *scale to real-world programs?* To answer this question we analysed 99 most starred Go projects on GitHub with GOMELA. We measure the number/parameters of models generated and the verification run-times. We also report on the error scores obtained for four properties.

RQ1: How does GOMELA *compare to the state-of-the-art?*

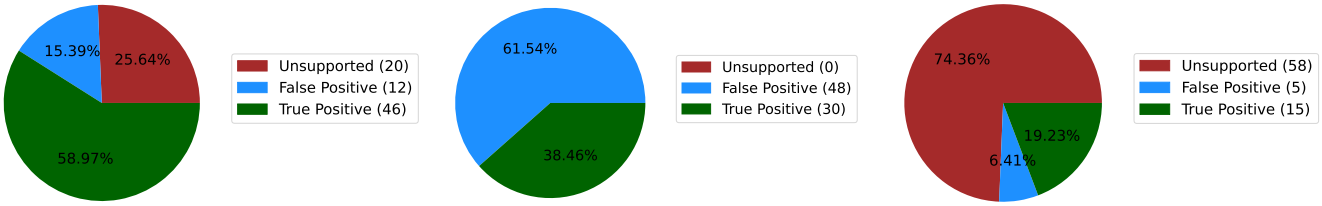
Here we focus on programs which contain bugs related to channels, waitgroups, or mutexes. We have collected a set of buggy programs, that consists of blocking examples from [7], [33], and six additional programs with intricate concurrency patterns (all benchmarks are available online [6], [9]). These programs range from 12 to 298 LoC (mean: 83, median: 70). We evaluate GOMELA on this set, against the state-of-the-art static checkers: GCatch [22] and Godel2 [7]. We omit benchmarks from [7] that contain data-races or do not contain any bugs. We focus on the benchmarks from [33] 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 [33]. 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 [33].

Figures 8 and 9 give the results of our experiments. For GOMELA, a program is reported as a *true positive* if at least one verification returns false, and *false positive* if all verifications return true (no bug found), all properties are checked with $S = \{0, 1, 3\}$. For GCatch and Godel2, a *true positive* is a correctly identified buggy program, while a *false positive* corresponds to a missed bug. For each tool, an example is *unsupported* if the tool aborts or crashes. 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 supports for real-world examples.

Figure 9 shows how the three tools perform wrt. execution time for true positives. From left to right, the first group of examples are all four benchmarks that contain a blocking bug from [7]. The second group of programs include: the program in Figure 3; a program with a blocking range-over-channel, see Figure 10 (right); a program (`FindAll()`) adapted from a bug report in Google's `gops` project [10], [27]; a program with a circular dependency and a bounded for-loop, see Figure 10 (left); a program invoking function `preload()` from Figure 1; and a buggy version of Figure 7 where `receiver` receives `x+1` times.

The third group consists of the 68 blocking programs from the GoKer benchmark suite [33], 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 limitations of the front-end of GOMELA and Godel2. Our modifications consists in inlining concurrency primitive declarations, inlining higher-order functions, and replacing virtual method calls by static ones.

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



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

Fig. 8: Proportion of true/false positives in 78 buggy programs, *unsupported* indicates that the tool has aborted or crashed.

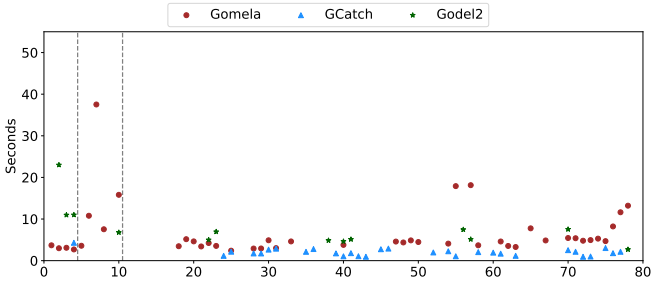


Fig. 9: Run-times for true positives in 78 buggy programs.

Figure 9. 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 7.

RQ2: How does GOMELA *scale to real-world programs?*

To evaluate the scalability of our approach we ran GOMELA on 99 most starred Go projects on GitHub. For each project, we cloned it locally and generated models for each of its packages. Each model was then verified, wrt. the properties described in § IV-A, using our automated approach with $S = \{0, 1, 3\}$.

Our measurements for scalability are in Table I. 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. GOMELA only generates a model when it detects at least one concurrency primitive. Table I (top) shows that most projects and packages only give rise to a single model, but some produce more than one hundred models. The table also shows that 75% of these models have at most 3 concurrency parameters (hence they require at most $|S|^3$ 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 concurrency operations contained in the source program. Overall GOMELA generated 3175 models, out of which 99 were not valid Promela (due to limitations of our aliasing analysis) and were rejected by Spin, 22 contained >5 concurrency parameters and 70 contained at least one valuation that took >30 seconds to verify (we omitted these in the next phase).

Table I (bottom) shows the run-time for the remaining 2984 models we verified. More than 75% of projects can be verified under 3m26s, and a model valuation is verified under 3.60s on average (we set a timeout of 30s per valuation). This suggests that our approach does scale to real-world Go.

TABLE I: Project sizes and verification run-times.

Project sizes	mean	std	25%	50%	75%	max
models per project	42.91	64.24	5	19	50	339
models per package	3.83	8.27	1	1	3	159
parameters per models	2.18	2.29	1	1	3	42
states per valuation	7970	156k	17	45	151	7.8 mil
Run-time						
per project	3m48s	7m5s	20s	1m4s	3m26s	35m5s
per model	4.56s	28.78s	2.56s	2.80s	3.33s	19m9s
per valuation	3.60s	1.90s	2.62s	3.11s	3.87s	19s

TABLE II: Verification scores for generated models.

All scores	mean	std	25%	50%	75%	max
model deadlock ϕ_{MD}	0.025	0.15	0	0	0	1
channel safety ϕ_{CS}	0	0	0	0	0	0
mutex safety ϕ_{MS}	0.002	0.048	0	0	0	1
waitgroup safety ϕ_{WS}	0.0006	0.019	0	0	0	0.667
Strictly positive scores (occurrences)						
model deadlock (81)	0.89	0.20	0.75	1	1	1
channel safety (0)	0	0	0	0	0	0
mutex safety (9)	0.869	0.15	0.75	1	1	1
waitgroup safety (3)	0.59	0.104	0.45	0.67	0.67	0.67

Table II (top) shows the score computed for the 2984 models. As expected from mature projects on GitHub, GOMELA reports few concurrency bugs; this suggests that it has a reasonable false alarm rate. Table II (bottom) focuses on the models for which at least one valuation violated a property. We manually verified a randomly selected sample of these models. 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 9 mutex errors, all were false alarms (all due to higher-order functions). Finally, the 3 reports of waitgroup error were real bugs.

VII. Related work

Static checkers are available as part of the Go ecosystem, e.g., the native `go vet` command, `staticcheck` [15], and tools based on Go's language server [12], but all these tools rely on syntactical checks (linters) that cannot detect concurrency bugs that require a semantic analysis.

The Go library also includes a *runtime* global deadlock detector and a *runtime* data-race detector. Sulzmann and

```go
func parseFiles(files []os.File) {
  ch := make(chan string)
  for file := range files {
    go parseFile(ch, file)
  }
  for v := range ch { // blocked
    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
}
```

Fig. 10: Examples of buggy programs caught by GOMELA, missed by GCatch [22], and unsupported by Godel2 [7].

Stadtmüller [29], [30] have proposed techniques for run-time analyses of Go programs based on vector clocks.

GCatch [22] combines several *static* bug detectors and includes a novel detector for blocking bugs (BMOC). This detector relies on approximating possible executions and uses an SMT-solver to determine whether they lead to blocking bugs. GCatch is accompanied by GFix which aims at repairing bugs involving at most two goroutines and a single channel. 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 fed into GCatch.

Several works propose static checkers based on techniques from programming languages theory, e.g., behavioural types [7], [20], [21], [25], abstract interpretation [24], and forkable regular expressions [28]. None of these support programs with concurrency parameters. The work by Gabet and Yoshida [7] is the latest in the behavioural type series. It detects channel and mutex-related bugs and data-races, but it has limited applicability. Our work builds on our earlier prototype [5] which uses Promela as a behaviour type language. In this work, we have extended it with an *inter*-procedural extraction of concurrency parameters, support for more concurrency primitives (waitgroup and mutex), support for structures and methods, an automated verification algorithm (Algorithm 1), and an evaluation of our approach on real-world Go code.

There is a growing body of surveys of Go programs and tools. Dilley and Lange [4] show that channels are used often and intensively in Go projects. Tu et al. [31] show that message-passing concurrency is as error-prone as traditional shared-memory concurrency. Yuan et al. [33] propose a benchmark suite of concurrency bugs in Go on which they compare the performance of goleak [32] (a *run-time* goroutine leak detector), go-deadlock [26] (a *run-time* deadlock detectors for locks), and Gong [20] (a *static* checker).

Software model checking has a long history, see [16] for a comprehensive survey. While other general purpose model checkers have support for inter-process communication [3], [23], we found that Spin is closest to Go and thus allows a nearly one-to-one translation from Go. Spin has notably been used to verify multi-threaded Java programs [13] and multi-threaded C programs [34]. Our approach is related to other specialised bounded model checkers for C, C++, and Java, e.g., CBMC [1], [17], ESBMC [8], and JBMC [2]. These checkers use symbolic execution techniques (e.g., unrolling loops k times) to detect low-level bugs.

VIII. Conclusions and Future Work

We described a novel approach to verifying Go programs using bounded model checking of parameterised behavioural types. This approach scales well to real-world Go code, and outperforms the competition on 78 buggy programs. Remarkably, all false alarms we discovered were due to limitations of our front-end rather than over-approximation in our verification approach.

The key remaining challenges are to develop scalable techniques to deal with virtual method calls, to develop heuristics to deal with models with high number of parameters, and to develop techniques to help automate the identification 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.

Acknowledgment

We thank Rumyana Neykova, Tiago Cogumbreiro, and the anonymous reviewers for their insightful feedback on earlier versions of this work.

References

[1] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[2] L. C. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtík. JBMC: A bounded model checking tool for verifying java bytecode. In *CAV (1)*, volume 10981 of *Lecture Notes in Computer Science*, pages 183–190. Springer, 2018.

[3] S. Cranen, J. F. Groote, J. J. A. Keiren, F. P. M. Stappers, E. P. de Vink, W. Wesselink, and T. A. C. Willemse. An overview of the mCRL2 toolset and its recent advances. In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2013.

[4] N. Dilley and J. Lange. An empirical study of messaging passing concurrency in Go projects. In *SANER*, pages 377–387. IEEE, 2019.

[5] N. Dilley and J. Lange. Bounded verification of message-passing concurrency in Go using promela and spin. In *PLACES@ETAPS*, volume 314 of *EPTCS*, pages 34–45, 2020.

[6] N. Dilley and J. Lange. Gomela. https://github.com/nicolasdilley/gomela-ase21/, 2021.

[7] J. Gabet and N. Yoshida. Static race detection and mutex safety and liveness for Go programs (artifact). *Dagstuhl Artifacts Ser.*, 6(2):12:1–12:3, 2020.

[8] M. Y. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole. ESBMC 5.0: an industrial-strength C model checker. In *ASE*, pages 888–891. ACM, 2018.

[9] *Automated Verification of Go Programs via Bounded Model Checking (Artifact)*. Zenodo, 2021.

[10] Google. Findall code. https://github.com/google/gops/blob/6fb0d860e5fa50629405d9e77e255cd32795967e/goprocess/gp.go#L29, 2020.

[11] Google. preload code. https://github.com/google/trillian/blob/c92fa63aaa6c133eb8383f2727524421bea420c4/storage/cache/subtree_cache.go#L108, 2021.

[12] `gopls`, the Go language server. https://github.com/golang/tools/tree/master/gopls.

[13] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA pathfinder. *STTT*, 2(4):366–381, 2000.

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

[15] D. Honnef. Staticcheck. https://staticcheck.io/.

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

[17] D. Kroening and M. Tautschnig. CBMC - C bounded model checker - (competition contribution). In *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 389–391. Springer, 2014.

[18] Kubernetes. Generatenodemap code. https://github.com/kubernetes/kubernetes/blob/d70ee902fddc682863a3cc4f0d8eac0223ebf70b/test/e2e/storage/vsphere/nodemapper.go#L62, 2021.

[19] Kubernetes. Kubernetes (k8s). https://github.com/kubernetes/kubernetes, 2021.

[20] J. Lange, N. Ng, B. Toninho, and N. Yoshida. Fencing off Go: liveness and safety for channel-based programming. In *POPL*, pages 748–761. ACM, 2017.

[21] J. Lange, N. Ng, B. Toninho, and N. Yoshida. A static verification framework for message passing in Go using behavioural types. In *ICSE*, pages 1137–1148. ACM, 2018.

[22] Z. Liu, S. Zhu, B. Qin, H. Chen, and L. Song. Automatically detecting and fixing concurrency bugs in Go software systems. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 616–629, New York, NY, USA, 2021. Association for Computing Machinery.

[23] J. Magee and J. Kramer. *Concurrency - state models and Java programs*. Wiley, 1999.

[24] J. Midtgaard, F. Nielson, and H. R. Nielson. Process-local static analysis of synchronous processes. In *SAS*, volume 11002 of *Lecture Notes in Computer Science*, pages 284–305. Springer, 2018.

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

[26] sasha s. `go-deadlock`. https://github.com/sasha-s/go-deadlock.

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

[28] K. Stadtmüller, M. Sulzmann, and P. Thiemann. Static trace-based deadlock analysis for synchronous mini-go. In *APLAS*, volume 10017 of *Lecture Notes in Computer Science*, pages 116–136, 2016.

[29] M. Sulzmann and K. Stadtmüller. Trace-based run-time analysis of message-passing Go programs. In *Haifa Verification Conference*, volume 10629 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2017.

[30] M. Sulzmann and K. Stadtmüller. Two-phase dynamic analysis of message-passing Go programs based on vector clocks. In *PPDP*, pages 22:1–22:13. ACM, 2018.

[31] T. Tu, X. Liu, L. Song, and Y. Zhang. Understanding real-world concurrency bugs in Go. In *ASPLOS*, pages 865–878. ACM, 2019.

[32] Uber. `goleak`. https://github.com/uber-go/goleak.

[33] T. Yuan, G. Li, J. Lu, C. Liu, L. Li, and J. Xue. Gobench: A benchmark suite of real-world Go concurrency bugs. In *CGO*, pages 187–199. IEEE, 2021.

[34] A. Zaks and R. Joshi. Verifying multi-threaded C programs with SPIN. In *SPIN*, volume 5156 of *Lecture Notes in Computer Science*, pages 325–342. Springer, 2008.