Bounded verification of message-passing concurrency in Go using Promela and Spin

Nicolas Dilley
University of Kent

Julien Lange
University of Kent

This paper describes a static verification framework for the message-passing fragment of the Go programming language. Our framework extracts models that over-approximate the message-passing behaviour of a program. These models, or behavioural types, are encoded in Promela, hence can be efficiently verified with Spin. We improve on previous works by verifying programs that include communication-related parameters that are unknown at compile-time, i.e., programs that spawn a parameterised number of threads or that create channels with a parameterised capacity. These programs are checked via a bounded verification approach with bounds provided by the user.

1 Introduction

Go is an increasingly popular programming language that is known for its lightweight threads (called *goroutines*) and native support for message-passing concurrency. Go programmers are encouraged to coordinate threads by exchanging messages over channels, rather than using shared memory protected by mutexes [13]. In a recent empirical survey [3], we have discovered that more than 70% of the most popular Go projects on GitHub use message-passing primitives. Additionally, Tu et al. [18] showed that message-passing based software is as liable to errors as other concurrent programming techniques. They also showed that Go concurrency bugs are hard to detect and have a long life time. This is reflected in a recent survey amongst Go programmers reporting that programmers often do not feel they are able to effectively repair bugs related to Go's concurrency features [17]. Concretely, message-passing concurrency bugs in Go fall in two categories: (*i*) blocking errors, where a goroutine is permanently waiting for a matching send/receive action and (*ii*) channel errors, where a goroutine attempts to close or send to a channel that is already closed.

The Go ecosystem provides little support for users to detect concurrency bugs. Its type system only ensures that each channel instance carries a single specified data type. While a *run-time* global deadlock detector is available, it is silently disabled by some libraries. To help programmers produce correct concurrent software, several authors have proposed techniques to verify Go programs both statically (at compile-time) [7, 8, 10, 12, 14] and dynamically (at run-time) [15, 16]. One of the more mature techniques for statically verifying Go programs is Godel [8] which relies on the similarity of Go's message-passing aspect to CCS [11]. Godel follows an approach based on behavioural types where Go programs are over-approximated by CCS-like processes, which in turns are model-checked, using mCRL2 [2] for safety and liveness properties. Because mCRL2 only deals with finite-state models, Godel has one key limitation: it does not support programs that spawn new threads in for-loops, e.g., the program in Figure 1 is not supported. This restriction limits the applicability of Godel to real-world code-bases. Indeed, 58% of the Go projects we studied in [3] feature thread-spawning in for-loops.

Figure 1 shows a typical Go program where several worker threads are concurrently sending data to the parent thread via channel `a`. Note that this program spawns $|\texttt{files}|$ threads and creates a channel

Submitted to:
PLACES 2020

```
1  func main() {
2    files := getFiles()                // decl. of getFiles() omitted
3    a := make(chan string, len(files)) // create bounded buffer 'a'
4    for i := 0; i < len(files); i++ {
5      go worker(a, files[i], i) //spawn worker()
6    }
7    for i := 0; i < len(files); i++ {
8      <-a // receive from 'a'
9    }
10 }
11 func worker(a chan int, f string, i int) {
12   a <- parseFile(f, i) // send data on 'a' (decl. of parseFile() omitted)
13 }
```

Figure 1: File processing example

whose capacity is |`files`|. The length of `files` is unknown at compile-time, hence this program cannot be checked for concurrency errors with existing static verification techniques for Go [7, 8, 10, 12, 14].

Our approach Our short-term objective is to improve the approach from [8] so that we can detect bugs in programs that feature communication-related parameters that are unknown at compile-time. We focus on two kinds of communication-related parameters: (*i*) those that determine the number of threads a program may spawn at run-time and (*ii*) those that determine the capacity of channels. For example, the number of threads and the capacity of channel `a` are unknown at compile-time in Figure 1. To fulfil our objective, we augment the behavioural types technique of [8] with (*i*) an *intra*-procedural analysis to identify unknown communication-related parameters, and (*ii*) a bounded verification wrt. these parameters. Concretely, we infer behavioural types from Go programs which may feature (undefined) communication-related parameters. If so, we ask the users to instantiate these parameters with bounds so that we can model-check the inferred behavioural types. The main challenges are to ask for user-provided bounds only when necessary and to ensure that these bounds are used consistently. We address these challenges by keeping track of variables that may be used in channel creation statements or for-loops that spawn threads.

Our long-term objective is to study automated *repair* of message-passing errors in Go. To anticipate for this next step, we deviate from [8] in several ways. (1) We infer behavioural types directly from Go source code instead of its lower-level (SSA) representation. (2) We use Promela and Spin instead of mCRL2 to encode and verify behavioural types. Promela has the advantage of being much closer to Go. It has an imperative Go-like syntax and natively supports synchronous and asynchronous channels. As a consequence, it will be easier to syntactically map an error in a Promela model to its source program. (3) We divide Go programs into independent partitions. This allows us to detect partial deadlocks and to identify the location of defects more precisely, while making our tool faster. Figure 2 gives an overview of our approach, which we have implemented in a tool called GOMELA [4].

Synopsis In § 2, we present a core subset of Go, called *MiniGo*, as well as typical bugs that we want to rule out. In § 3, we give a detailed algorithm to extract Promela models from Go programs, while keeping track of communication-related parameters. In § 4 we present our implementation and its empirical evaluation. We discuss related work and conclude in § 5.

2 *MiniGo* and message-passing concurrency errors

For the sake of presentation, we use a fragment of Go that is focused on its message-passing features and call it *MiniGo* — we describe how our tool deals with a larger subset of Go in Section 4. The syntax

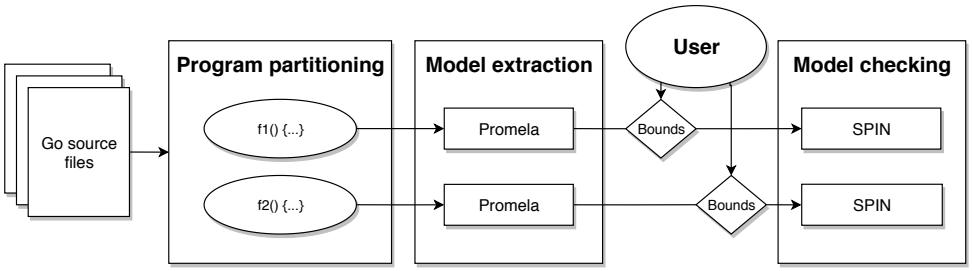


Figure 2: GOMELA workflow.

of *MiniGo* is given in Figure 3. We only discuss its semantics informally and refer to [7] for a formal account of the semantics of a variation of our language.

We use v to range over *non-channel* variables, ch to range over *channel* variables, x to range over any variables, e to range over expressions (excluding channel variables), a to range over expressions (possibly including channel variables), id to range over function names, and n to range over integer literals. We use r to range over mutators of for-loop indices. We use $\widetilde{a}$ to range over a list of expressions and overload the notation for lists of statements ($\tilde{s}$), etc. We write $\tilde{s}_1\tilde{s}_2$ for the concatenation of $\tilde{s}_1$ and $\tilde{s}_2$. We write $\texttt{chans}(\widetilde{a})$ (resp. $\texttt{chans}(\tilde{x})$) for the maximal sub-list of $\widetilde{a}$ (resp. $\tilde{x}$) that contains only channel variables.

A *MiniGo* program p consists of a list of function declarations $\tilde{d}$, possibly including a `main` function (the program's entry point). Each function declaration specifies a list of parameters $\tilde{x}$ (possibly including channel variables) and a function body $\tilde{s}$.

Statement ch `:= make(chan,`e`)` creates a new channel of capacity n, when e evaluates to n. If $n = 0$ then the channel is synchronous, otherwise it is asynchronous. Communication statements α interact with channels: $v \leftarrow ch$ receives a value from channel ch and binds it to variable v; while $ch \leftarrow e$ sends the evaluation of expression e on channel ch. Send actions are blocking when the channel is synchronous or has reached its maximal capacity. A channel can be closed with a $\texttt{close}(ch)$ operation. Any send or close action on a closed channel triggers a run-time error. Any receive action on a closed channel succeeds, if the channel is empty a default value is returned. Select statements $\texttt{select}\{\widetilde{c}\}$ are guarded choices: they block until one of the guarding communication operations succeeds; after which the corresponding case is executed. If multiple operations are available, one is chosen non-deterministically. Select statements may include a unique `default` branch, which is taken if all other branches are blocking.

A statement `go` $id(\widetilde{a})$ spawns a new goroutine, i.e., an instance of function $id(\widetilde{a})$ which is executed concurrently with its parent thread. *MiniGo* also includes standard constructs such as general sequencing, conditionals, for-loops, and assignment. For the sake of simplicity we model only the relevant parts of the language of expression (see definition of e). We assume that variable names are pairwise distinct. Additionally, as in [8], we assume that channel are not in e, that variables are immutable, and that recursive functions do not spawn goroutines (for-loops are more common than recursion in Go).

Message-passing errors in *MiniGo* In this work, we are interested in message-passing related bugs that *MiniGo* programs may encounter at run-time. We distinguish three types of such bugs. A **global deadlock** is a situation where at least one goroutine is waiting for a send or receive action to succeed, while *all* the other goroutines are either blocked or terminated. A **partial deadlock** is a situation where at least one goroutine is permanently stuck while waiting for a send or receive action to succeed. Go developers refer to partial deadlocks as *goroutine leaks* because such stuck goroutines never reach the end of their scope, and thus are never garbage-collected. A **channel safety error** is a situation where a send or close operation is triggered on a closed channel.

$$
\begin{array}{lcl}
p & := & \widetilde{d} \\
s & := & ch \texttt{ := make(chan,}e\texttt{)} \\
 & | & \alpha \mid \texttt{select}\{\widetilde{c}\} \mid \texttt{close}(ch) \\
 & | & id(\widetilde{a}) \mid \texttt{go } id(\widetilde{a}) \mid \{\tilde{s}\} \\
 & | & \texttt{if } e \texttt{ then } \tilde{s}_1 \texttt{ else } \tilde{s}_2 \mid \texttt{for } v\texttt{:=}e_1\texttt{;}e_2\texttt{;}r\{\tilde{s}_3\} \\
\alpha & := & v \leftarrow ch \mid ch \leftarrow e
\end{array}
\qquad
\begin{array}{lcl}
c & := & \texttt{case } \alpha\texttt{:}\tilde{s} \mid \texttt{default:}\tilde{s} \\
e & := & \texttt{true} \mid \texttt{false} \mid n \mid v \mid \ldots \\
a & := & ch \mid e \\
x & := & ch \mid v \\
d & := & \texttt{func } id(\tilde{x})\{\tilde{s}\} \\
r & := & v\texttt{++} \mid v\texttt{--} \mid \ldots
\end{array}
$$

Figure 3: Syntax of *MiniGo* (*ch* ranges over channel variables and *v* stands for non-channel variables).

3 Extracting Promela models from *MiniGo* programs

We adopt an approach based on behavioural types to produce a sound analysis of *MiniGo* for channel safety and global deadlock errors, following [7, 8]. Note that this approach is generally unsound wrt. liveness properties such as partial deadlock freedom without a termination checker, see [7, Section 5]. In this context, behavioural types are an over-approximation of the interactions between goroutines, i.e., they record send and receive actions, while abstracting away from the computational aspects. Typically, conditional statements are assigned behavioural types that correspond to non-deterministic choices in process calculi. In our work, behavioural types take the form of Promela models, which we extract from *MiniGo* source code. Remarkably, we keep track of some computational aspects when they affect the structure of the communication of the program, e.g., in Figure 1 we need to keep track of `len(files)`.

Given a *MiniGo* program p, we extract Promela models as follows. For each function declaration $\texttt{func } id(\tilde{x})\{\tilde{s}\}$ where $\tilde{x}$ does not contain any channel variables, we generate a model which consists of three parts: (1) a model entry point (`init` process in Promela) that contains the translation of $\tilde{s}$; (2) a list of process declarations (`proctype` in Promela), one for each distinct function call occurring (inter-procedurally) in $\tilde{s}$; (3) a set of monitor processes, one for each channel created in $\tilde{s}$. Each of these models correspond to a partition of a *MiniGo* program. Because these partitions do not have free channel variables, they are effectively independent. Hence, we can verify them independently by considering each function declaration without channel parameters as a program entry point. As a consequence, we obtain a more precise and wider analysis of code-bases, while reducing the computational cost of our analysis, comparing to [8]. In particular, we can detect some *partial* deadlocks in the program under considering by identifying global deadlocks in some of its partitions.

Hereafter, we take the following conventions: Promela strings generated by our algorithm are written in `typewriter blue`. *MiniGo* code is written in *italic*. Our approach is formalised through functions (in `typewriter black`) and algorithms (in **bold-red**) that manipulate *MiniGo* programs. Each identifier in *MiniGo* is translated to the equivalent string in Promela, e.g., *ch*1 in *MiniGo* is translated to `ch1`. For the sake of readability, we omit the concatenation operator between literal Promela strings and strings generated by translation functions.

Function declarations Given a function body $\tilde{s}$, for each distinct (blocking) function call $id(\widetilde{a})$ occurring inter-procedurally in $\tilde{s}$ such that $\texttt{chans}(\widetilde{a}) \neq []$, we define a Promela process (`proctype`) as follows:

$$\texttt{proctype } id\texttt{(chanParams(}id\texttt{),ch)\{TransStmts(}\varnothing\texttt{,body(}id\texttt{));ch!0\}}$$

where `ch` is a channel used to signal the termination of the function call (with `ch!0`).
For each distinct non-blocking function call $\texttt{go } id(\widetilde{a})$, such that $\texttt{chans}(\widetilde{a}) \neq []$, we define the process:

$$\texttt{proctype go\_}id\texttt{(chanParams(}id\texttt{))\{TransStmts(}\varnothing\texttt{,body(}id\texttt{))\}}$$

where $\texttt{chanParams}(id)$ (resp. $\texttt{body}(id)$) returns the *channel* parameters (resp. body) of function *id* and $\varnothing$ denotes the empty map. Observe that the non-channel parameters are abstracted away. We use

```
function TransStmts(Δ, s s̃)
    switch s :
        case ch ← e :  ch.in!0;ch.sending?state;TransStmts(Δ, s̃)
        case v ← ch :  ch.in?0;TransStmts(Δ, s̃)
        case close(ch) :  ch.closing?state;TransStmts(Δ, s̃)
        case if e then s̃1 else s̃2 :
            if
                ::  true ->   TransStmts(Δ, s̃1)
                ::  true ->   TransStmts(Δ, s̃2)
            fi; TransStmts(Δ, s̃)
        case select{c̃} :
            if
                TransStmts(Δ, c̃)
            fi; TransStmts(Δ, s̃)
        case case α : s̃1 : ::  TransStmts(Δ, α) -> TransStmts(Δ, s̃1)
        case default : {s̃1} : ::  true -> TransStmts(Δ, s̃1)
        case id(ã) :
            if chans(ã) ≠ [] then
                ch = [0] of {int}; run id(chans(ã), ch);ch?0; TransStmts(Δ, s̃)
            otherwise TransStmts(Δ, s̃)
        case go id(ã) :
            if chans(ã) ≠ [] then
                run go_id (chans(ã)); TransStmts(Δ, s̃)
            otherwise TransStmts(Δ, s̃)
        case for v :=e1 ;e2 ;r {s̃3} :
            let (Δ′, x, y) = lookup_Δ(v:=e;e;r) in
            if spawns(s̃3) ∨ Δ == Δ′ then
            for (i: x .. y ){TransStmts(Δ′, s̃3)}; TransStmts(Δ′, s̃)
            otherwise
              do    ::  true ->   TransStmts(Δ, s̃3)
                    ::  true ->   break;
              od;
            TransStmts(Δ, s̃)
        case ch := make(chan, e) :
            let (Δ′, _, y) = lookup_Δ(i := 0; i < e; i++) in
            Chandef ch;
            chan ch.in = [y] of {int};
            run chanmonitor(ch);
            TransStmts(Δ′, s̃)
```

Algorithm 1: Extracting Promela from *MiniGo* statements. We assume that `TransStmts`$(\Delta, [])$ returns the empty string.

`proctype` instead of `inline` definition as the latter cannot include declarations of new channels. Next, we define function `TransStmts` which translates *MiniGo* statements to Promela.

Algorithm 1 specifies how we extract a model from a list of *MiniGo* statements. We use b to range over the control statements of a for-loop, i.e., b ranges over triples of the form $(v := e_1 ; e_2 ; r)$.

Function `TransStmts` takes two parameters: (1) Δ maps expressions (corresponding to communication-related parameters) to Promela strings, and (2) a list of *MiniGo* statements.

Channel primitives For each *MiniGo* channel we generate a custom Promela structure, called `Chandef`, which contains three channels: `in` carries the exchanged messages, while `sending` (resp. `closing`) is used to monitor send (resp. closing) actions. A send statement is translated to a send statement in Promela (on channel `in`), followed by a receive statement on the corresponding channel monitor (on the `sending` channel, see below). A receive statement is translated to a Promela receive (on channel `in`). A close statement is translated to a Promela receive on channel `closing`.

$$\downarrow b = \begin{cases} e_1 & \text{if } b \text{ is } v := e_1;\ v < e_2;\ v\texttt{++} \\ e_2 & \text{if } b \text{ is } v := e_1;\ v > e_2;\ v\texttt{--} \\ \bot & \text{otherwise} \end{cases} \qquad \uparrow b = \begin{cases} e_2 & \text{if } b \text{ is } v := e_1;\ v < e_2;\ v\texttt{++} \\ e_1 & \text{if } b \text{ is } v := e_1;\ v > e_2;\ v\texttt{--} \\ \bot & \text{otherwise} \end{cases}$$

$$\texttt{lookup}_\Delta(b) = \begin{cases} (\Delta, x, y) & \text{if } \downarrow(b) = \bot \text{ or } \uparrow(b) = \bot, \text{ with } x \text{ and } y \text{ fresh} \\ (\Delta', \Delta'(\downarrow b), \Delta'(\uparrow b)) & \text{otherwise, where } \Delta' = \Delta[e \mapsto x \mid e \in \{\downarrow(b), \uparrow(b)\} \setminus dom(\Delta) \text{ with } x \text{ fresh}] \end{cases}$$

Figure 4: Auxiliary functions for Algorithm 1, where we assume $\Delta(n) = n$ for each integer n.

Conditionals An if-then-else statement is translated to an `if` block in Promela. It behaves as a non-deterministic internal choice (`true` is an always-enabled guard). A select statement is translated to a non-deterministic choice, using an `if` block where each non-default branch is guarded by a send or receive action. Default branches are translated to a branch that is always available.

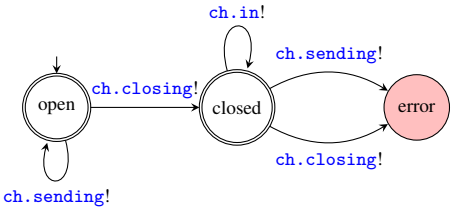
Control statements A blocking function call is translated to Promela code that spawns an instantiation of the corresponding Promela process (using the `run` keyword), then waits for it to terminate by waiting on fresh channel `ch`. For spawning function calls, i.e., `go` $id(\tilde{a})$, there are two cases. If the parameters include channels, the algorithm returns Promela code that spawns the corresponding process. Otherwise it omits the call entirely — as it will be checked in the model of an independent partition.

To translate `for` v`:=`e_1`;`e_2`;`$r\{\tilde{s}_3\}$, we need to first check (*i*) whether we can extract well-identified bounds that we consider as communication-related parameters and (*ii*) whether the loop contains (inter-procedurally) spawning function calls, i.e., $\texttt{spawns}(\tilde{s}_3)$ holds (whose straightforward definition is omitted). If $\texttt{spawns}(\tilde{s}_3)$ holds then the range of the loop needs to be finite. Hence, either we set-up Promela variables (that the user will instantiate) to define a range; or we are able to identify a static range (integer literal bounds). When the loop does not spawn new threads, we only use a finite Promela for-loop if all involved variables have been seen before ($\Delta{=}{=}\Delta'$, see below). In all other cases, we use a non-deterministic loop using a Promela `do` block which can be exited at any iteration with a `break` operation.

We keep track of re-usable communication-related parameters with the map Δ and use the function `lookup`, defined in Figure 4, to query it. Functions $\downarrow(b)$ and $\uparrow(b)$ respectively return the lower and upper bounds of for-loop control statement b. When the control statements of a for-loop are well-formed (they obey a recognisable pattern, i.e., $\downarrow b \neq \bot \wedge \uparrow b \neq \bot$), the `lookup` function returns a new map Δ' and the lower ($\Delta'(\downarrow b)$) and upper ($\Delta'(\uparrow b)$) bounds of the for-loop. Map Δ' augments Δ with mappings from newly identified *expressions* to fresh Promela variables.

Channel creation A channel creation statement is translated to the instantiation of a `Chandef` structure and the spawning of its `chanmonitor` process. We initialise the `in` channel of the `Chandef` structure with a capacity corresponding to its *MiniGo* equivalent. If the capacity is not a integer literal, the `lookup` function ensures that we either re-use Promela variables, or generate fresh ones. Note that channels `sending` and `closing` in `Chandef` are always synchronous.

Channel monitors To detect channel safety errors, we keep track of the state of *MiniGo* channels. We use Promela processes to monitor channel actions, i.e., send, receive, and close. As an optimisation, we only create such monitors when a `close` primitive appears in the program. Processes corresponding to goroutines interact with channel monitors via an instance `ch` of a `ChanDef` structure which contains three channels (`in`, `sending`, and `closing`). The automa-



```
1  #define len_files_0  15
2
3  typedef Chandef {
4    chan in = [0] of {int};
5    chan sending = [0] of {int};
6    chan closing = [0] of {bool};
7  }
8  init {
9    chan _ch0_in = [len_files_0] of {int};
10   Chandef _ch0;
11   _ch0.in = _ch0_in;
12   run chanMonitor(_ch0)
13
14   for(i : 1.. len_files_0) {
15     run mainworker(_ch0)
16   };
17   for(i : 1.. len_files_0) {
18     _ch0.in?0
19   };
20 }
21 proctype mainworker(Chandef a) {
22   a.in!0;
23   a.sending?0
24 }
25
26 proctype chanMonitor(Chandef ch) {
27   bool open = true;
28   do
29   :: true ->
30     if
31     :: open ->
32       if
33       :: ch.sending!open;
34       :: ch.closing!true ->
35         open = false
36       fi
37     :: else ->
38       if
39       :: ch.sending!open ->
40         assert(false)
41       :: ch.closing!true ->
42         assert(false)
43       :: ch.in!0;
44       fi
45     fi
46   od
47 }
```

Figure 5: Model extracted from Listing 1 with Algorithm 1

ton on the right represents the behaviour of this monitor (`chanMonitor`). Figure 5 (lines 26- 47) gives the corresponding Promela code. In *MiniGo* and Go, when a channel is *closed*, sending on it or closing it will raise an error, hence the transitions from state *closed* to state *error* in the automaton. Also, receive actions on closed channels always succeed, hence the self-loop on state *closed*.

Example 1. The model extracted from Figure 1 with Algorithm 1 is given in Figure 5. Lines 8-20 contain the translation of the `main` function. Note that a `chanMonitor` is spawned at line 12. The definition of this process is given in lines 26-47. The translation of the `worker` function is in lines 21-24.

Figure 5 contains one (unknown) communication-related parameter: `len(files)` which is used in the capacity of channel `a`, and in the loops at lines 4 and 7 of Figure 1. Observe that the communication-related parameter `len(files)` is set to 15 here (see line 1). Our implementation also allows user to specify such parameters as program (command line) arguments. Expression `len(files)` is first seen by Algorithm 1 in a channel creation statement. At this point, it adds `len(files)` $\mapsto$ `'len_files_0'` in map Δ. When the algorithm processes both loops, it invokes lookup_{Δ}`(i:=0;i<len(files);i++)` to obtain lower and upper bounds (0 and `'len_files_0'`, respectively). Note that the loop in line 4 is a spawning loop, hence it must be translated to a finite loop in Promela to obtain a finite model. The loop in line 7 is not spawning, but it is still translated to a finite loop because all its bounds are already in Δ.

4 Implementation and evaluation

Implementation We have implemented our approach in a tool called GOMELA [4]. Given a Go program, our tool extracts Promela models (as described in Section 3). If necessary, the user enters values (bounds) for the statically unknown communication-related parameters produced by Algorithm 1 (i.e., the Promela variables). For instance, the user provides a bound to instantiate `len(files)` in Figure 1. This value is then used as a bound in the for-loops at lines 4 and 7 as well as the capacity of channel `a`.

Table 1: Go programs verified by GOMELA and comparison with Godel [8]. All programs are available online [4].

#	Programs & `Partitions`	GOMELA |States|	GOMELA CS	GOMELA GD	GOMELA Infer (ms)	GOMELA Spin (ms)	Godel Time	Godel ψ_s	Godel ψ_g	Manual analysis CS	Manual analysis GD
1	file processing (|`files`|=15)	376880	✓	✓	85	1666	⊥	⊥	⊥	✓	✓
2	file processing v1 (|`files`|=15)	109	✓	✗	86	1057	⊥	⊥	⊥	✓	✗
3	file processing v2 (|`files`|=15)	110	✓	✗	85	1027	⊥	⊥	⊥	✓	✗
4	prod-cons (k=5,n=10,m=10)	4802785	✓	✓	85	31239	⊥	⊥	⊥	✓	✓
5	alt-bit	35	✓	✓	956	1391	460	✓	✓	✓	✓
6	concsys	96	-	-	768	8194	588	✓	✓	-	-
	- `ConcurrentSearch()`	49	✓	✓	-	1126	-	-	-	✓	✓
	- `ConcurrentSearchWC()`	26	✓	✗	-	1335	-	-	-	✓	✗
	- `First()`	3	✓	✓	-	1183	-	-	-	✓	✓
	- `ReplicaSearch()`	9	✓	✗	-	1326	-	-	-	✓	✗
	- `SequentialSearch()`	3	✓	✓	-	1073	-	-	-	✓	✓
	- `FakeSearch()`	3	✓	✓	-	1083	-	-	-	✓	✓
	- `main()`	3	✓	✓	-	1068	-	-	-	✓	✓
7	cond-recur	13	✓	✓	84	1278	623	✓	✓	✓	✓
8	dinephil	968	✓	✓	804	1478	859	✓	✓	✓	✓
9	dinephil5	71439	✓	✓	835	1801	8681	✓	✓	✓	✓
10	double-close	18	✗	-	85	1486	463	✗	✓	✗	✓
11	fanin-alt	34	✓	✗	769	1686	786	✓	✓	✓	✗
12	fanin	15	✓	✓	681	1495	621	✓	✓	✓	✓
13	fixed	14	-	-	740	2379	656	✓	✓	-	-
	- `Work()`	2	✓	✓	-	1117	-	-	-	✓	✓
	- `main()`	12	✓	✓	-	1262	-	-	-	✓	✓
14	forselect	21	✓	✓	755	1507	813	✓	✓	✓	✓
15	jobsched	48	-	-	781	2745	589	✓	✓	-	-
	- `main()`	45	✓	✓	-	1532	-	-	-	✓	✓
	- `morejob()`	3	✓	✓	-	1213	-	-	-	✓	✓
16	mismatch	12	-	-	714	2316	603	✓	✗	-	-
	- `Work()`	2	✓	✓	-	1044	-	-	-	✓	✓
	- `main()`	10	✓	✗	-	1272	-	-	-	✓	✗
17	sel	21	✓	✗	84	1719	326	✓	✗	✓	✗
18	selFixed	19	✓	✓	82	1330	572	✓	✓	✓	✓
19	philo	18	✓	✗	85	1362	537	✓	✗	✓	✗
20	starvephil	67	✓	✗	759	1343	836	✓	✗	✓	✗
21	nonlive	7	✓	✓	850	1194	550	✓	✓	✓	✓
22	nonlive v1	7	✓	✗†	850	1152	366	✓	✗†	✓	✓
23	prod-cons	61	✓	✓	87	1390	508	✓	✓	✓	✓
24	prod3-cons3	5746	✓	✓	643	1534	19963	✓	✓	✓	✓
25	prodconsclose	12185424	✓	✓	163	18672	25348	✓	✓	✓	✓
26	stuckmsg	5	✓	✓	86	1109	790	✓	✓	✓	✓
27	data-dependent	12	✓	✗†	86	1170	694	✓	✗†	✓	✓
	Column number	1	2	3	4	5	6	7	8	9	10

GOMELA uses Spin to check whether each model is free from channel safety errors and global deadlocks. Spin reports any global deadlock and any trace that leads to an `assert(false)` statements. We use the former to check for global deadlocks (GD) in a program's partitions, and the latter to check for channel safety errors (CS). Spin detects when the main process (`init`) terminates while another process is still running. We use this to detect some goroutine leaks, i.e., a particular case of partial deadlocks.

In addition to *MiniGo* statements, GOMELA deals with constants (used as communication-related parameters), anonymous functions, break statements, for range loops and switch statements. Occurrences of integer constants are replaced with their actual values. To deal with anonymous functions, GOMELA generates Promela corresponding function declarations (using fresh names) and its (unique) invocation. Go's `break` statements are translated as Promela break statements. For-ranges loop (`for range` $list\{\tilde{s}_1\}$) are treated as for-loops with control statements of the form: to `for` i`:=0;`$i < len(list)$`;i++`. Finally, `switch` statements are translated into n-ary internal choices (similar to an if-then-else).

Evaluation To evaluate our tool, we ran it on several benchmarks, including some from [8, Table 1]. The results of this evaluation are in Table 1. In Table 1, Column 1 gives the number of states in the model, as given by Spin. In Column 2 (resp. 3) a ✓-mark says that the corresponding program partition is channel-

safe (resp. free of global deadlock); a ✗-mark says that the property is violated. Column 4 (resp. 5) shows the time (milliseconds) taken to extract (resp. verify) the Promela model of a partition. The timing for programs are the sum of all of their partitions. Column 6 shows the time (milliseconds) taken by Godel [8] to verify channel safety (ψ_s, Column 7) and global deadlock (ψ_g, Column 8) properties in. A $\perp$-mark means that Godel does not support this program. A ✓-mark in Column 9 (resp. 10) says that it was not possible to find any channel errors (resp. global deadlocks) manually, ✗† highlights false alarms.

In Table 1, Program 1 is the example in Figure 1 where the user has set `len(files)` to 15. Programs 2 and 3 are variations of Program 1 with a global deadlock and a goroutine leak, respectively. Program 4 spawns `n` producers and `m` consumers that interact (repeatedly) over a channel with capacity `k`. Observe that these are not supported by Godel because they include thread-spawning in a for-loop. Programs 5 to 26 are taken from [8, Table 1]. We note that Godel runs marginally faster than GOMELA on programs with small models. This is due to Spin's start up time of $\sim$1s. For larger programs (more than 5000 states) GOMELA performs much better than Godel, see, e.g., Programs 9, 24, and 25. This suggests that our tool scales better than Godel on larger code-bases.

Below we comment on the errors identified in the programs from Table 1.

- Program 6 has two partitions that contain global deadlocks. In `ConcurrentSearchWC`, a function spawns three goroutines that send a message on a shared channel `c`. That function may terminate silently (via timeout) hence leaving three unmatched send actions in the goroutines. `ReplicaSearch` includes a similar pattern where the parent thread may terminate while leaving some goroutines permanently blocked.
- Program 10 contains a channel safety error where a channel is closed twice by two different threads. We note that Spin aborts the verification as soon as it finds such errors.
- Program 11 creates two producers and one consumer. The consumer may terminate silently in which case both producers are blocked permanently. Program 12 is similar to Program 11 except that the consumer never terminates, thus fixing the bug.
- Program 16 contains two partitions: `Work` consist of a non-terminating loop (without communications), while `main` contains a global deadlock due to a mismatch between the number of send and receive actions. Note that if we were modelling this program with one monolithic partition, we would not detect a global deadlock because `Work` never blocks.
- In Program 17 each goroutine may get stuck because of a mismatch between the number of send and receive actions.
- Program 19 is an encoding of the (starving) philosopher problem using only two philosophers, taken from [14]. Program 20 is another encoding of the (starving) dining philosophers from [8].
- Programs 21 and 22 are two variants of a program that contain two goroutines: one is waiting for a message, while the other contains a non-terminating for-loop. In Program 21, the non-terminating for-loop is followed by a (unreachable) matching send action. These programs show the limits of the behavioural types approach: they contain proper partial deadlocks. The problem in Programs 21 is only detected in Godel by using an additional termination checker.
- Program 27 is given in Figure 6. This program gives a typical example of a false alarm raised by our tool, and any existing approach based on behavioural types. Because if-then-else statements are translated to non-deterministic choices, our approach is unable to determine that the two conditional blocks are "synchronised" by the same invocation of function `f()`.

Limitations Our approach is applicable to *MiniGo*, extended with the syntactic constructs discussed above. Several key limitations need to be tackled to address the full Go language. We assume that variables are immutable, as a consequence we cannot soundly analyse programs that, e.g., mutate a list

```
1  func main() {
2    a := make(chan int)
3    go send(a)
4    if f() {  // decl. of f() elided
5      a <- 0
6    } else {}
7  }
8
```

```
9  func send(a chan int) {
10   if f() {
11     <-a
12   } else {}
13 }
14 /* f() is a deterministic function
      without side-effects */
```

Figure 6: Data-dependent choice (Program 27).

`files` in between using `len(files)` as a communication-related parameter. Go has object oriented-like features, such as structs, methods, and interfaces which we currently do not support. Virtual method calls (on interfaces) are particularly difficult to model. As in [7, 8, 10, 12, 14], we do not support channel passing (since we abstract away the data sent over channels). We note that our empirical survey [3] found that only 6% of projects used channels that carry channels.

5 Related work, conclusions and future work

Related work Spin and Promela have been used extensively in software verification. Notably Java PathFinder [5] translates Java programs to Promela models which are then verified for deadlocks and violations of user-provided assertions. Also, Zaks and Joshi [19] use Spin to verify multi-threaded C programs using their LLVM representation and custom virtual machine.

Several works focus on the verification of message-passing concurrency in Go [7, 8, 10, 12, 14, 15, 16]. Four papers studied static verification using behavioural models. Ng and Yoshida [12] proposed *dingohunter*, the first static global deadlock detection tool for Go. It relies on communicating finite-states machines [1] and multiparty compatibility [9]. Their work does not support asynchronous channels nor programs that spawn goroutines or create channels in loops or conditionals. Stadtmüller et al. introduced *gopherlyzer* [14] which detects global deadlocks using forkable regular expressions. This work does not support channel closures, asynchronous channels, nor goroutines spawned in loops. Lange et al. [7, 8] proposed Gong and Godel, whose approach serves as a basis for this work. Gong uses an *ad-hoc* checker which supports bounded verification of infinite-state models, but did not scale well. Instead, Godel uses mCRL2 [2] as a back-end. Because mCRL2's communication model is very different from Go's the encoding from behavioural types to mCRL2's language is very intricate, see [6]. Promela is a more convenient language for this purpose, but because Spin supports only LTL, while mCRL2 supports the μ-calculus, it is not possible to check the liveness property specified in [7, 8]. However, we can still identify some goroutine leaks by checking whether their corresponding models reach their end states.

Conclusions Our work builds on the approach in [8] and improves it to support statically unknown communication-related parameters via a bounded analysis. Our approach allows us to support programs that spawn a parameterised number of goroutines or channel capacities. Our evaluation shows that our tool scales well and produces models that can be easily understood and adjusted by programmers.

Future work Our short term plans are to support additional concurrency-related Go features, e.g., barriers (`WaitGroup`). We will also improve our algorithms to support more complex for-loops control statements, and to perform a fully *inter*-procedural analysis of communication-related parameters. In the longer term, we plan to use our tool to detect concurrency errors and suggest repairs for large code-bases. We plan to perform a large-scale empirical evaluation of this toolchain on the dataset identified in [3].

References

[1] Daniel Brand & Pitro Zafiropulo (1983): *On Communicating Finite-State Machines*. *J. ACM* 30(2), pp. 323–342, doi:10.1145/322374.322380.

[2] Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink & Tim A. C. Willemse (2013): *An Overview of the mCRL2 Toolset and Its Recent Advances*. In: *TACAS*, *Lecture Notes in Computer Science* 7795, Springer, pp. 199–213, doi:10.1007/978-3-642-36742-7_15.

[3] Nicolas Dilley & Julien Lange (2019): *An Empirical Study of Messaging Passing Concurrency in Go Projects*. In: *SANER*, IEEE, pp. 377–387, doi:10.1109/SANER.2019.8668036.

[4] Nicolas Dilley & Julien Lange (2020): *Gomela*. `http://github.com/nicolasdilley/gomela`.

[5] Klaus Havelund & Thomas Pressburger (2000): *Model Checking JAVA Programs using JAVA PathFinder*. *STTT* 2(4), pp. 366–381, doi:10.1007/s100090050043.

[6] Julien Lange, Nicholas Ng & Bernardo Toninho: *Godel Checker*. `https://bitbucket.org/MobilityReadingGroup/godel-checker`.

[7] Julien Lange, Nicholas Ng, Bernardo Toninho & Nobuko Yoshida (2017): *Fencing off Go: liveness and safety for channel-based programming*. In: *POPL*, ACM, pp. 748–761, doi:10.1145/3009837.3009847.

[8] Julien Lange, Nicholas Ng, Bernardo Toninho & Nobuko Yoshida (2018): *A static verification framework for message passing in Go using behavioural types*. In: *ICSE*, ACM, pp. 1137–1148, doi:10.1145/3180155.3180157.

[9] Julien Lange, Emilio Tuosto & Nobuko Yoshida (2015): *From Communicating Machines to Graphical Choreographies*. In: *POPL*, ACM, pp. 221–232, doi:10.1145/2676726.2676964.

[10] Jan Midtgaard, Flemming Nielson & Hanne Riis Nielson (2018): *Process-Local Static Analysis of Synchronous Processes*. In: *SAS*, *Lecture Notes in Computer Science* 11002, Springer, pp. 284–305, doi:10.1007/978-3-319-99725-4_18.

[11] Robin Milner (1984): *Lectures on a calculus for communicating systems*. In: *International Conference on Concurrency*, Springer, pp. 197–220, doi:10.1007/3-540-15670-4_10.

[12] Nicholas Ng & Nobuko Yoshida (2016): *Static deadlock detection for concurrent Go by global session graph synthesis*. In: *CC*, ACM, pp. 174–184, doi:10.1145/2892208.2892232.

[13] Rob Pike (2015): *Go Proverbs*. `https://www.youtube.com/watch?v=PAAkCSZUG1c`.

[14] Kai Stadtmüller, Martin Sulzmann & Peter Thiemann (2016): *Static Trace-Based Deadlock Analysis for Synchronous Mini-Go*. In: *APLAS*, *Lecture Notes in Computer Science* 10017, pp. 116–136, doi:10.1007/978-3-319-47958-3_7.

[15] Martin Sulzmann & Kai Stadtmüller (2017): *Trace-Based Run-Time Analysis of Message-Passing Go Programs*. In: *Haifa Verification Conference*, *Lecture Notes in Computer Science* 10629, Springer, pp. 83–98, doi:10.1007/978-3-319-70389-3_6.

[16] Martin Sulzmann & Kai Stadtmüller (2018): *Two-Phase Dynamic Analysis of Message-Passing Go Programs Based on Vector Clocks*. In: *PPDP*, ACM, pp. 22:1–22:13, doi:10.1145/3236950.3236959.

[17] The Go team (2016-17): *Go Survey Results*. `blog.golang.org/survey[year]-results`.

[18] Tengfei Tu, Xiaoyu Liu, Linhai Song & Yiying Zhang (2019): *Understanding Real-World Concurrency Bugs in Go*. In: *ASPLOS*, ACM, pp. 865–878, doi:10.1145/3297858.3304069.

[19] Anna Zaks & Rajeev Joshi (2008): *Verifying Multi-threaded C Programs with SPIN*. In: *SPIN*, *Lecture Notes in Computer Science* 5156, Springer, pp. 325–342, doi:10.1007/978-3-540-85114-1_22.