View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Kent Academic Repository

Kent Academic Repository

Full text document (pdf)

Citation for published version

Lange, Julien and Ng, Nicholas and Toninho, Bernardo and Yoshida, Nobuko (2018) A Static Verification Framework for Message Passing in Go using Behavioural Types. In: ICSE '18: 40th International Conference on Software Engineering, 27 May - 03 Jun 2018, Gothenburg, Sweden.

DOI

https://doi.org/10.1145/3180155.3180157

Link to record in KAR

http://kar.kent.ac.uk/65587/

Document Version

Publisher pdf

Copyright & reuse
Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research
The version in the Kent Academic Repository may differ from the final published version.
Users are advised to check **http://kar.kent.ac.uk** for the status of the paper. **Users should always cite the published version of record.**

Enquiries
For any further enquiries regarding the licence status of this document, please contact:
researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at **http://kar.kent.ac.uk/contact.html**

KAR Kent Academic Repository

University of Kent

A Static Verification Framework for Message Passing in Go using Behavioural Types

Julien Lange
University of Kent
j.s.lange@kent.ac.uk

Nicholas Ng
Imperial College London
nickng@imperial.ac.uk

Bernardo Toninho
Imperial College London
b.toninho@imperial.ac.uk

Nobuko Yoshida
Imperial College London
n.yoshida@imperial.ac.uk

ABSTRACT

The Go programming language has been heavily adopted in industry as a language that efficiently combines systems programming with concurrency. Go's concurrency primitives, inspired by process calculi such as CCS and CSP, feature channel-based communication and lightweight threads, providing a distinct means of structuring concurrent software. Despite its popularity, the Go programming ecosystem offers little to no support for guaranteeing the correctness of message-passing concurrent programs.

This work proposes a practical verification framework for message passing concurrency in Go by developing a robust static analysis that infers an abstract model of a program's communication behaviour in the form of a *behavioural type*, a powerful process calculi typing discipline. We make use of our analysis to deploy a model and termination checking based verification of the inferred behavioural type that is suitable for a range of safety and liveness properties of Go programs, providing several improvements over existing approaches. We evaluate our framework and its implementation on publicly available real-world Go code.

CCS CONCEPTS

• **Theory of computation → Verification by model checking**; **Type theory**; Process calculi; • **Software and its engineering → Model checking**; **Automated static analysis**; *Software verification*; *Concurrent programming languages*;

ACM Reference Format:
Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2018. A Static Verification Framework for Message Passing in Go using Behavioural Types. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180157

1 INTRODUCTION

Modern programming languages have evolved with the ever increasing need for highly available, interactive software services , providing programmers with frameworks that facilitate the development of such intricate communicating systems. Amongst these languages, the Go programming language created at Google in 2007 targets the development of concurrent software systems by integrating channel-based concurrency and lightweight threads as distinctive language features, greatly inspired by advances in formal languages for concurrency theory known as *process calculi* [43]. Go enables programmers to write statically-typed concurrent software, and has been used successfully in a range of industrial settings such as Uber [45] and Dropbox's infrastructure [15], the Docker [14] software container platform, the Kubernetes [27] cluster manager, among others [9, 41].

However, beyond its simple static type system, Go provides fairly few assurances on the correctness of concurrent code. At compile time, Go only enforces that messages exchanged via communication channels adhere to the declared channel payload types, providing no way of detecting common concurrency errors such as deadlocks or undelivered messages. At runtime, Go offers only a toy global deadlock detector. This is in sharp contrast with the rich body of work on process calculi-based verification, where a plethora of type-based and logic-based techniques enable reasoning about safety and liveness properties of interactive systems.

Given the foundations of Go's message-passing concurrency in process calculi, our work aims to bridge the divide between the foundations and programming practices by applying modern process calculi based verification techniques to real-world Go concurrent programming. Concretely, we propose a static verification framework for concurrency and message-passing communication using *concurrent behavioural types* [24], which have been developed extensively in concurrency theory since the early 90s.

To achieve this, we crucially address the substantial conceptual gap that exists between a formal mathematical language (a process calculus) and a general purpose programming language with concurrency features. Our approach analyses general Go source code and distills from programs behavioural types that serve as a faithful *model* of its message-passing concurrent behaviour. Our behavioural types consist of a simplified form of *concurrent processes* which are reminiscent of Concurrent Communicating Systems (CCS) [34] or Communicating Sequential Processes (CSP) [22] (which inspired the design of the Go language). Given such a formally grounded model, we may then apply a range of process calculi oriented verification techniques to Go. Specifically, we convert Go source code into a static single assignment (SSA) form which provides a fine-grained view of the concurrency primitives used in programs in a quasi-functional form [2], enabling our behavioural type inference. We then employ model checking and termination-checking techniques to automatically verify safety and liveness properties such as deadlock-freedom and communication safety.

A significant advantage of our approach over previous works [30, 36, 40] is that our inference procedure covers a much larger part of the Go language allowing for the *automatic* extraction of an accurate model of a program's concurrency-related features, resulting in a more precise analysis with reduced numbers of false alarms and

This work is licensed under a Creative Commons Attribution International 4.0 License.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5638-1/18/05.
https://doi.org/10.1145/3180155.3180157

```
1  func prod(ch chan int) {
2    for i := 0; i < 5; i++ {
3      ch <- i // Send i to ch
4    }
5    close(ch) // No further values accepted at ch
6  }
7  func cons(ch1, ch2 chan int) {
8    for {
9      select {
10     case x := <-ch1: print(x) // Either input from ch1
11     case x := <-ch2: print(x) // or input from ch2
12     }
13   }
14 }
15 func main() {
16   ch1, ch2 := make(chan int), make(chan int)
17   go prod(ch1)
18   go prod(ch2)
19   cons(ch1, ch1)
20 }
```

Figure 1: Partially deadlocked Producer-Consumer in Go.

undetected errors. Our integration with a general purpose model checker also enables us to modularly verify arbitrary safety and liveness properties, over the more single-minded nature of previously proposed techniques, as well as take advantage of advances in model checking to provide better performance scaling.

Concurrent Programming in Go. We provide an overview of the Go programming language with an emphasis on the challenges of concurrent programming and how our verification framework can check for common concurrency errors in programs.

Go is a language with message-passing concurrency features and lightweight threads (known as *goroutines*). The key feature of Go's concurrency primitives is the predominance of *channel-based* communication over shared memory based communication amongst threads. In Go, a channel consists of a (typed) buffer that can be used by an arbitrary number of threads for read and write operations. Channels are *synchronous* by default (i.e. blocking on reads and writes) but can be made asynchronous by specifying a buffer size during channel creation. Asynchronous channels provide non-blocking sends while the buffer is not full.

We introduce the Go programming language with the program in Figure 1 which implements a producer/consumer concurrent pattern with two producers and one consumer thread, communicating over a pair of synchronous channels. The producer code (lines 1-6) is written as a function that takes as a parameter a channel `ch` that can carry payloads of type `int`. A producer merely sends an integer value over the given communication channel `ch` (written `ch <- i` in Go, where `i` is the value to be sent) a predetermined number of times (encoded as a `for` loop) and then *closes* the channel, signalling that no further values are to be sent.

The consumer code (lines 7-14) is specified as a function taking two channels `ch1` and `ch2` (one per producer). The `cons` function consists of a common Go idiom known as a *for-select* loop: a potentially infinite loop (the parameterless `for` on line 8) containing a selective communication construct (line 9). The behaviour of `select` is such that the consumer waits for an input on either `ch1` or `ch2` (inputs in Go are written `<-ch`). Whenever communication is available, the appropriate `case` is selected. The consumer prints

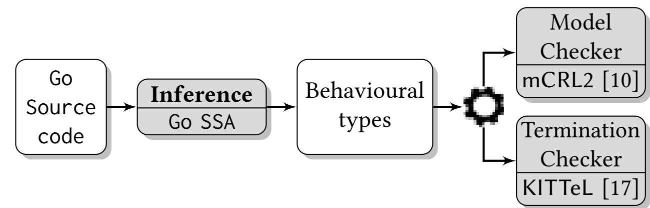

Figure 2: The Godel Checker [31] workflow.

the received integer from either producer. Despite the channels being closed by the producers, the consumer's inputs still succeed.

Finally, the program entry point (`main` in lines 15-20) sets up the producers and consumer by creating the two synchronous channels `ch1` and `ch2` (line 16), spawning two producers in parallel (achieved by the `go` prefix to the function calls, which creates a *goroutine* that runs in parallel with the main program executing the `prod` function) and then running the `cons` function.

Common concurrency errors in Go. We describe common errors in channel-based Go programming, as well as limitations of Go's built-in runtime detector:

Channel safety errors: Once a channel is closed, receive actions always succeed (receiving messages in-flight or a default value for the payload type), but all send and close actions performed on the channel raise a runtime error. Hence, channels should be closed at most once and no message should be sent on closed channels.

Global deadlocks: The Go *runtime* contains a *global* deadlock detector that signals a runtime error when *all* goroutines in a program are stuck (i.e. deadlocked). However it is often the case that when certain libraries are imported (such as the commonly used `net` library for networking) the global deadlock detector is silently *disabled* [5], i.e. all global deadlocks are just ignored.

Partial deadlocks: It is often the case that a program's communication cannot progress despite *some* of its goroutines not being stuck. This is known as a *partial* deadlock or as a failure of *liveness*. For instance, the `cons` function above is being executed with the *wrong* channels (`ch1` twice instead of `ch1` and `ch2`), due to a programmer error. Running the program results in a system that is not *live*, since the second producer is not interacting with the consumer – the outputs are never matched with their respective inputs. Since only a subset of the goroutines are stuck, these errors cannot be detected by the Go runtime.

A static verification framework for Go. This paper proposes a *static* analysis toolchain dubbed Godel Checker [31], which can automatically detect safety and liveness errors in real-world Go programs. The workflow is presented in Figure 2, consisting of three layers: given a Go program we first perform (1) *behavioural type inference* (detailed in § 3) that extracts a behavioural type model for the program (§ 2). In this stage, we use the SSA (static single assignment) package from the Go project and apply a control flow analysis to obtain behavioural types. We then apply (2) a *model checking* tool, `mCRL2` [10], to the extracted behavioural types (§ 4). This enables us to check types with a finite state-space (i.e. *finite control*) for the absence of global deadlocks, as well as several Go specific safety properties (including channel safety). Finally, to pinpoint potentially problematic loops and accurately detect partial

deadlocks, we augment our approach with a termination analysis for loops in the original Go source code using a term-rewriting based tool, KITTeL [17] (§ 5). We show benchmarks for (publicly available) Go programs and compare with existing tools (§ 6).

2 BEHAVIOURAL TYPES FOR GO

This section introduces *behavioural types* [24] for concurrent Go (which are infered as an abstract model of Go – see § 3) and their relationship with Go programs, following our previous work in [30].

Behavioural types are a typing discipline in which types express the possible actions of a program in a fine-grained way [24]. When applied to communication and concurrency, behavioural types act as an abstract specification of all communication actions that may be performed in a program. Moreover, behavioural types are an *executable* specification. They have a natural operational meaning and evolve throughout program execution.

The syntax of types is given below, it extends the types defined in [30] with general sequencing. The types abstract away data elements, singling out the concurrency specific features such as spawning of threads (i.e. *goroutines*), creation of communication channels, send and receive actions, and selective communication.

$$\begin{array}{rcl} \alpha & := & \overline{u} \mid u \mid \tau \qquad\qquad \boldsymbol{T} := \{\boldsymbol{t}_i(\tilde{y}_i) = T_i\}_{i\in I} \ \mathtt{in}\ S \\ T, S & := & \alpha; T \mid T; S \mid T \oplus S \mid \&\{\alpha_i; T_i\}_{i\in I} \mid (T \mid S) \mid \mathbf{0} \\ & \mid & (\mathtt{new}^n\ a); T \mid \mathtt{close}\, u; T \mid \boldsymbol{t}\langle\tilde{u}\rangle \mid \lfloor a \rfloor_k^n \mid a^\star \end{array}$$

Communication is specified with the α prefix, where α can be $\overline{u}$, denoting a send on channel u, u, denoting the dual receive action on channel u and τ, denoting a silent internal step (u is either a constant channel or a variable). $T; S$ models the sequential composition of T and S. The construct $\mathtt{close}\, u$ signals that channel u is to be *closed*. We represent conditional branching with the $T \oplus S$ construct, denoting a *non-deterministic* internal choice between T and S. Thus, our type level analogue of conditional branching does not depend on data but rather simulates the ability to take either branch of a conditional through a non-deterministic step.

We model Go's selective communication with $\&\{\alpha_i; T_i\}_{i\in I}$. The construct waits for the availability of one of the α_i communication actions. When some action α_j becomes available, it is executed and the communication evolves to behaviour T_j (discarding the other options). When more than one communication action is available, one is chosen non-deterministically. Since τ actions are always available to fire, we can use a τ action in a select construct to model timeouts or default behaviours when no other actions are available. The parallel composition construct $T \mid S$ denotes the parallel execution of T and S; the construct $\mathbf{0}$ denotes no behaviour.

Channels in Go are synchronous buffers by default, but may also be created with a *bound*, achieving asynchronous communication. Send actions are non-blocking until the number of messages in the buffer reaches the bound and, dually, receive actions on empty buffers are blocking until a message is available. $(\mathtt{new}^n\ a); T$ denotes the creation of a channel a (with a bound n) which can be used locally in T. If the bound n is set to 0, then a is a synchronous channel. We often write $(\mathtt{new}\, a); T$ for $(\mathtt{new}^0\ a); T$ and assume that the scope of a extends as far to the right as possible.

Construct $\boldsymbol{t}\langle\tilde{u}\rangle$ denotes a function call with parameter $\tilde{u}$. We often identify a list $\tilde{u}$ with its underlying set and write $x \in \tilde{u}$ if x is a element of $\tilde{u}$. We use the following two *runtime* constructs to define the semantics of types in § 4: $\lfloor a \rfloor_k^n$ represents a communication channel at a (where n is the maximum capacity of the queue and k is the current number of messages in it) and $a^\star$ represents a closed channel a.

The type of a program, sometimes written $\{\boldsymbol{t_i}(\tilde{y_i}) = T_i\}_{i\in I}$ in S, is a set of (potentially mutually recursive) definitions T_i with a distinguished program *entry point* S. Recursive definitions can be parameterised by communication channels and represent the *goroutines* that are executed in the program. For each type definition $\boldsymbol{t_i}(\tilde{y_i}) = T_i$ we assume that the free names of T_i are included in $\tilde{y_i}$.

Example 2.1 (Type for Consumer-Producer and their Properties). The behavioural type for the program of Figure 1 is given below.

$$\begin{array}{l} \{\ \boldsymbol{prod}(ch) = \overline{ch}; \boldsymbol{prod}\langle ch\rangle \oplus \mathtt{close}\ ch \\ \ \ \boldsymbol{cons}(ch1,\ ch2) = \&\{ch1; \boldsymbol{cons}\langle ch1,\ ch2\rangle,\ ch2; \boldsymbol{cons}\langle ch1,\ ch2\rangle\} \\ \ \ \boldsymbol{main}() = (\mathtt{new}\, ch1,\ ch2); (\boldsymbol{prod}\langle ch1\rangle \mid \boldsymbol{prod}\langle ch2\rangle \mid \boldsymbol{cons}\langle ch1,\ ch1\rangle)\} \\ \hfill \mathtt{in}\ \boldsymbol{main}\langle\rangle \end{array}$$

Definition $\boldsymbol{prod}(ch)$ specifies the type for the `Producer` function, while $\boldsymbol{cons}(ch1, ch2)$ stands for the type of the `Consumer` function, and $\boldsymbol{main}\langle\rangle$ is the type of the program entry point (i.e. the `main` function). Notice how the imperative control structures are transformed into recursive definitions and the data elements are erased. For instance, the type $\boldsymbol{prod}$ specifies the behaviour of performing an internal choice (denoted by the $\oplus$ construct) between sending on ch and recursing or closing the channel ch and terminating.

Given that behavioural types act as a form of executable specifications, it is natural to consider properties of *types* in terms of their executions, as well as their relationship with program properties.

2.1 Behavioural Properties of Types

The property of ***global deadlock-freedom*** (GDF) entails that if a communication action is available to fire, the type can always make progress, meaning that a type as a whole is never globally stuck. For instance, the type $\boldsymbol{main}$ in Example 2.1 satisfies GDF since the communication actions in subcomponents $\boldsymbol{prod}\langle ch1\rangle$ and $\boldsymbol{cons}\langle ch1, ch1\rangle$ can always make progress, despite the fact that actions in $\boldsymbol{prod}\langle ch2\rangle$ are always stuck.

The property of ***liveness***, also known as ***partial deadlock freedom***, is strictly stronger than GDF, given that every live type is also GDF. It states that *all* communications that can become enabled in a type can always eventually fire. For instance, replacing the call to $\boldsymbol{cons}\langle ch1, ch1\rangle$ with $\boldsymbol{cons}\langle ch1, ch2\rangle$ makes the type $\boldsymbol{main}\langle\rangle$ in Example 2.1 satisfy liveness. We note that in the presence of internal choice (i.e. conditional branching), liveness requires that communication actions in *both* branches must eventually succeed, but when facing external choice (i.e. the **`select`** construct), only branches that can be selected are required to eventually succeed. For instance, the following single-producer variant of $\boldsymbol{main}\langle\rangle$ also satisfies liveness even though the $ch2$ branch in the select construct can never be taken: $(\mathtt{new}\, ch1, ch2); (\boldsymbol{prod}\langle ch1\rangle \mid \boldsymbol{cons}\langle ch1, ch2\rangle)$.

In § 4, we formally define the above properties, as well as other properties that are verified in our work, in the modal μ-calculus.

```
1  func main() {
2    ch := make(chan int) // Create channel
3    go sendFn(ch)        // Run as goroutine
4    x := recvVal(ch)     // Ordinary func call
5    for i := 0; i < x; i++ {
6      print(i)
7    }
8    close(ch) // Close channel ch
9  }
10 func sendFn(c chan int) {
11   c <- 42 // Send on channel c
12 }
13 func recvVal(c chan int) int {
14   return <-c // Receive from channel c
15 }
```

Listing 1: A simple concurrent program in Go.

2.2 Relationship between Types and Programs

In our analysis, a conditional is abstracted as a non-deterministic choice between the two alternative behaviours present in the then and else branches. This coarse abstraction introduces a subtle interaction between non-terminating program behaviour and *data-dependent* communication wrt. liveness [30, § 5] (we note that this issue does not affect GDF or Channel Safety). For instance, consider the following Go program:

```
1  func send(n int, c chan int) {
2    if n%2 == 0 { // Conditional (send if n is even)
3      c <- n // Send to channel c
4    }
5    send(n, c)
6  }
7  func recv(c chan int) {
8    for { // Infinite loop
9      x := <-c // Receive from channel c
10   }
11 }
12 func main() {
13   c := make(chan int)
14   go send(3, c)
15   go recv(c)
16 }
```

The type for the function send (lines 1-6) is $\boldsymbol{send}(c) = \overline{c}; \boldsymbol{send}\langle c\rangle \oplus \boldsymbol{send}\langle c\rangle$. Similarly, the type for recv (line 9) is $\boldsymbol{recv}(c) = c; \boldsymbol{recv}\langle c\rangle$. The type for the program above is deemed live since the then branch of the conditional in send can always eventually be reached through recursion, ensuring that the inputs in recv are matched. However, in the program the then branch of the conditional can never be reached and so the inputs in recv cannot succeed.

This example is symptomatic of a mismatch between type and program liveness in the presence of *infinite* executions that flow through a conditional. Note that it is not the case that the simple existence of non-termination makes the liveness analysis unsound [30, § 5]. For instance, for the example in § 1, type liveness implies program liveness, despite the presence of non-termination, since there is no communication contingent on a data-dependent test.

In § 5, we address this issue by deploying a lightweight termination analysis of iterative behaviour in our framework.

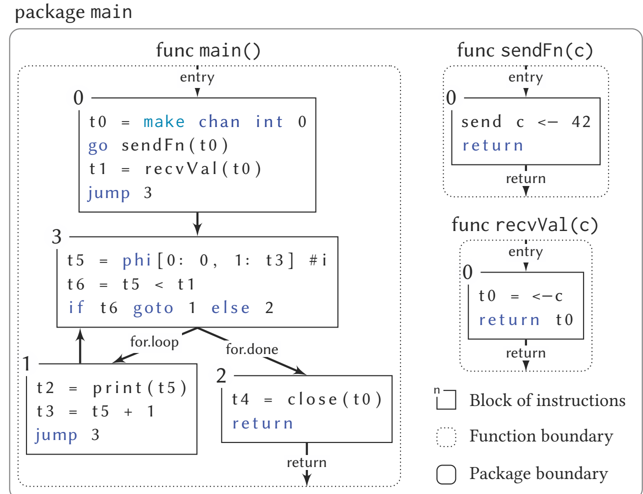

Figure 3: SSA IR built from Listing 1.

3 BEHAVIOURAL TYPE INFERENCE

We detail one of the main contributions of our work: the development and formalisation of a procedure that infers, whenever possible, behavioural types from Go source code. The inference consists of two key steps: (1) the conversion of Go source code to a static single assignment (SSA) intermediate representation (IR), using the ssa package from the Go standard library; and (2) the extraction of the communication structure as behavioural types from SSA blocks.

3.1 From Go source code to SSA IR

The ssa package [4] represents source code in SSA form and provides a high-level API for manipulating Go source code programatically. Go programs are organised as packages which consist of package variables and functions (also definable within bodies of functions). Each function is transformed into a list of *blocks* of SSA instructions, with one block marked as the function *entry point*.

Figure 3 gives a graphical view of the SSA representation of the program in Listing 1. Each of the three Go functions main(), sendFn(), and recvVal() becomes a set of blocks (graphically, a dotted box represents the scope of the function). The last instruction of a block is always a control flow instruction (if, jump, or return) connecting the block to its successors (if any). The successor relation is shown in the graph via *edges* connecting blocks within a function. There are also entry and return arrows for function entry and exit respectively, implicit from the SSA IR. Table 1 summarises the instructions of interest to our inference.

Communication instructions. Go's channel-based communication constructs are actual primitive language constructs. Thus, the key operations such as channel creation make(chan T), cf. line 2 from Listing 1, sending and receiving values over channels, ch <- value and <-ch respectively (lines 11 and 14), spawning of goroutines (go sendFn() in line 3), and closing channels, i.e. close

Table 1: Key SSA instructions used by our type inference.

`make chan T S`	Create channel of type T and size S
`local chan T`	Declare channel of type T
`ch <- v`	Send v to channel ch
`<-ch`	Receive from channel ch
`select b [<-t0,t1<-v]`	Non-deterministic select
`close(ch)`	Close channel ch
`jump 1`	Enter block 1
`if t0 goto 1 else 2`	If t0 then enter block 1 else block 2
`return`	Exit function
`F()`	Call function F
`go F()`	Spawn F as goroutine
`*t0 = t1`	Store t1 into address t0
$\texttt{phi}[\texttt{Blk}_i : \texttt{v}_i]_{i \in \text{InEdges}}$	Select v_i if predecessor block is Blk_i

```
1  func myselect(c chan int) {
2    select {
3    case msg := <-c:
4      print("received: ", msg)
5    case <-time.After(time.Second):
6      print("timeout: ready in 1s")
7    default:
8      print("default: always ready")
9    }
10 }
```

Listing 2: A select statement in Go.

operation (line 8), are all explicit in the SSA IR. As a result, identifying the channel operations that match with the corresponding behavioural types is straightforward.

Select instructions. Non-deterministic selective communication (`select`) also appears explicitly in the SSA representation but requires a more intricate representation. We illustrate the *structure* of the SSA representation of the select block from Listing 2 in Figure 4. Listing 2 shows a simple function `myselect(c)` which consists of a select construct featuring three cases: the first is guarded by a receive action on channel c, the second is guarded by a timeout, and the last is the default case (executed if none of the other cases are ready to be executed). The SSA IR of `myselect(c)` consists of 6 blocks. Block `0` is the entry point of the function, containing the SSA instruction for select. Note that the instruction `select nonblocking [<-c,<-t0]` contains only two cases. The default case is identified by the keyword `nonblocking`; if a select does not specify a default case its SSA representation becomes `select blocking [<-c,<-t0]`. We note that timeouts are implemented in Go as channels (e.g., t0) that receive a message after a predetermined time. This message is placed into the channel by the runtime and not by a user-level send. The statement `t2 = extract t1 #0` determines the index of the case which will be executed and stores it in t2. Block `0` ends with an if-then-else construct, which is the first of an if-then-else chain identifying which case of the select is to be executed depending on the value of t2. Blocks 2 and 4 represent the bodies of the first two cases, respectively. Block 5 contains the body of the default case (the default case is always the last block of the chain); if a select statement does not specify a default case, then this block contains a "panic" instruction which cannot be executed in normal circumstances. Finally, block 1 represents the code that follows the select statement.

Phi instructions. Another key SSA instruction is $\texttt{phi}[\texttt{Blk}_i : \texttt{v}_i]_{i \in \text{InEdges}}$ which is used to select between two or more variables when merging the control flow into one SSA block. An example of such an instruction is given in block 3 of Figure 3 where the instruction is used to select the value of variable t5 (the index of the for loop) depending on whether the predecessor of block 3 is block `0` or block 1. The former corresponds to the initialisation step

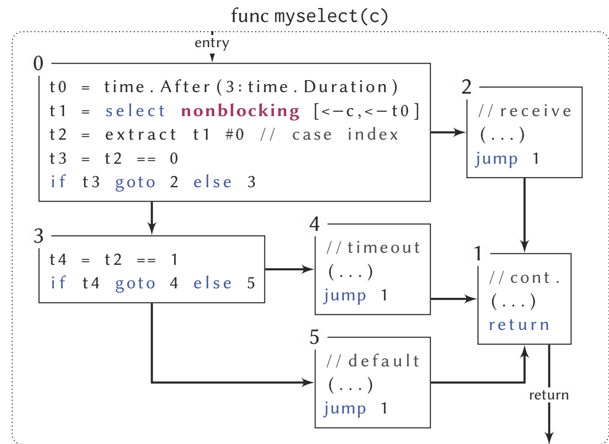

Figure 4: Simplified SSA IR built from Listing 2.

of the loop (in which case the index is `0`), the latter corresponds to an execution of the body of the loop.

Conventions. Given an SSA statement s, e.g., `t0 = make chan int 0`, the left-hand-side (LHS) is the part of the statement on the left of the = symbol (the *variable* t0). The key features of the SSA representation are that, within the scope of a function, all the LHS of the statements in the blocks of that function are pairwise distinct. Also, the static typing information is available for each statement. In addition, variables declared at the package level are initialised in a special `init()` function. Variables that are accessed by anonymous functions appear in the header of the SSA representation of that function as free variables.

3.2 Extracting type definition bodies

In Step (2) we *soundly* approximate, whenever possible, the communication behaviour of Go programs with the type language. First, for each SSA block n in each function $\texttt{fun}(\tilde{x})$, we generate a type signature of the form $\mathbf{fun_n}(\tilde{y}, \tilde{t}, \tilde{v})$ where:

- $\tilde{y}$ is a subsequence of $\tilde{x}$ where each y in $\tilde{y}$ is a *channel* parameter
- $\tilde{t}$ is a list of *channel* variables that appear in the LHS of the statements in the predecessors of block n and do not appear in the LHS in block n
- $\tilde{v}$ is a list of *global channel* variables (declared outside of function `fun`, e.g., package level variables).

We store all signatures in an environment Δ and write $\Delta(\texttt{fun}, n)$ for the signature of block n in function `fun`.

```
function genFunction(fun, n, k, ρ, Γ)
  switch s ← statement at line k do
    case t=make chan T S do
      genFunction(fun, n, k+1, ρ;(new^S t), Γ[t ↦ t])
    case t=local chan T do
      genFunction(fun, n, k+1, ρ, Γ[t ↦ ⊥])
    case t<-v or <-t or t'=<-t do
      genFunction(fun, n, k+1, ρ; mkPrefix_Γ(s), Γ)
    case close(t) do
      genFunction(fun, n, k+1, ρ; close Γ(t), Γ)
    case return do return ρ; 0
    case jump i do return ρ; mkJump_Γ(fun, i)
    case if _ goto i else j do
      return ρ; (mkJump_Γ(fun, i) ⊕ mkJump_Γ(fun, j))
    case select b [g_1, ..., g_j] do
      ρ_c ← mkJump_Γ(fun, n+1)
      for i in [1, ..., j] do
        ρ_i ← mkPrefix_Γ(g_i)
        ρ'_i ← mkJump_Γ(fun, n+2*i)
      if b = nonblocking then
        ρ_d ← mkJump_Γ(fun, n+1+2*j)
        return &{ρ_i; ρ'_i; ρ_c}_{i∈{1,...,j}} ∪ {τ; ρ_d; ρ_c}
      else return &{ρ_i; ρ'_i; ρ_c}_{i∈{1,...,j}}
    case F(x̃) or t=F(x̃) do
      if t is a channel then abort
      else genFunction(fun, n, k+1, ρ; mkCall_Γ(F, x̃), Γ)
    case go F(x̃) do
      ρ' ← genFunction(fun, n, k+1, ∘, Γ)
      return ρ; (mkCall_Γ(F, x̃) | ρ')
    case *t0 = t1 or t0 = *t1 do
      if t1 is a channel then
        genFunction(fun, n, k+1, ρ, Γ[t0 ↦ Γ(t1)])
      else genFunction(fun, n, k+1, ρ, Γ)
    case phi[Blk_i : v_i]_{i∈InEdges} do
      if ∃i ∈ InEdges : v_i is a channel then abort
      else genFunction(fun, n, k+1, ρ, Γ)
    otherwise do genFunction(fun, n, k+1, ρ, Γ)
```

Algorithm 1: Pseudo-code of genFunction

3.2.1 Core procedure: genFunction. We present the core algorithm, the genFunction procedure, which generates a type abstraction from an SSA block. The procedure takes five parameters: fun, the name of the function being considered; n, the identifier of the block; k, the line number within block n; ρ, the type we have constructed thus far; and Γ, a context which maps each channel variable name to its *representative.*

The context Γ is crucial in our development as Go allows channels to be aliased (i.e., several variables may contain a reference to the same channel) and channel variables to be overwritten, e.g., a channel variable may refer to different channels at different point of the execution of a program, or may be declared and only initialised at a later point. We keep track of aliased channels by assigning a unique *representative* to each newly created channel. We write $\Gamma[\mathtt{t} \mapsto \mathtt{t}']$ for the context Γ where the mapping from t is updated to t′. We assume that $\Gamma[\mathtt{t} \mapsto \mathtt{t}']$ is *undefined* if $\mathtt{t} \in \mathrm{dom}(\Gamma)$ and $\Gamma(\mathtt{t}) \neq \bot$ in order to *disallow* channel overwriting.

Algorithm for genFunction. Our algorithm relies on auxiliary (partial) functions for the translations from statements to types:

$mkPrefix_\Gamma(s)$	send/receive actions and select guards
$mkJump_\Gamma(\mathtt{fun}, j)$	jump statements
$mkCall_\Gamma(\mathtt{fun}, \tilde{x})$	function calls

Each function uses context Γ to generate communication actions and type definition calls, respectively.

Algorithm 1 gives the implementation of genFunction which iterates over the lines of block n in function fun and makes a case analysis depending on the structure of the statement s found at line k. The procedure returns a behavioural type or *aborts* whenever an invocation to auxiliary functions is undefined or when the algorithm reaches an "abort" statement, since in these cases we cannot guarantee a sound approximation of program behaviour. In particular, the algorithm aborts if a channel variable is overwritten (a new channel is assigned to it).

Channel creation/declaration. If s is a channel *creation* statement, variable t becomes the representative name for this channel and we update the environment with $\Gamma[\mathtt{t} \mapsto \mathtt{t}]$. The SSA representation guarantees that t is unique in function fun. We create the corresponding new channel type construct and recursively call genFunction over the next line. If s is a channel *declaration* statement, we update the environment with $\Gamma[\mathtt{t} \mapsto \bot]$. Note that t can only be used after it is initialised.

Send/receive. If s is a send or receive statement, we translate it to a type construct with a call to $mkPrefix_\Gamma(s)$, defined below:

$$mkPrefix_\Gamma(s) = \begin{cases} \overline{u} & \text{if } s = \mathtt{t<-v} \text{ and } \Gamma(\mathtt{t}) = u \\ u & \text{if } s \in \{\mathtt{<-t}, \mathtt{t'=<-t}\} \text{ and } \Gamma(\mathtt{t}) = u \\ \tau & \text{if } s = \mathtt{<-t} \text{ and t is a } \textit{timeout} \text{ channel} \end{cases}$$

Timeout channels are dedicated channels created at compile time to encode timeouts, they are never added in the context Γ.

Close is mapped to its respective type primitive, via context lookup.

Return. We return the type built so far appended with the termination construct.

Jump. We translate a jump statement into a type function call through the auxiliary function defined below, which uses the globally available signature environment (Δ).

$$mkJump_\Gamma(\mathtt{fun}, j) = \begin{cases} \boldsymbol{fun_j}\langle \tilde{y}, \Gamma(\tilde{t}), \tilde{v}\rangle & \text{if } \Delta(\mathtt{fun}, j) = \boldsymbol{fun_j}(\tilde{y}, \tilde{t}, \tilde{v}) \\ & \quad \text{and } \forall t \in \tilde{t} : \Gamma(t) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

$mkJump_\Gamma(\mathtt{fun}, j)$ models a jump to another block within the same enclosing function hence there is no need to rename the parameters nor the "global" variables from the function signature (since they are fixed within the scope of the function). Instead, the internally declared variables are replaced by their representatives using Γ. The function $mkJump_\Gamma(\mathtt{fun}, j)$ is undefined if any of the $\tilde{t}$ arguments maps to an uninitialised channel in order to guarantee that these cannot be overwritten in the definition of fun. The $\tilde{y}$ and $\tilde{v}$ arguments are assumed to be initialised by the parent function.

Conditional constructs are also translated straightforwardly using $\oplus$ and type definition call.

Select. If s is a select construct then s is followed by a chain of blocks linked by if-then-else statements, which encode the branching structure of the select, as explained in § 3.1. The jump to the

function genBody(fun, n)
  $\boldsymbol{fun_n}(\tilde{y}, \tilde{t}, \tilde{v}) \leftarrow \Delta(\texttt{fun}, n)$
  $\Gamma \leftarrow [x \mapsto x]_{x \in \tilde{y}, \tilde{t}, \tilde{v}}$
  return genFunction(fun, n, 0, $\circ$, Γ)
function genEquations()
  return $\{\Delta(\texttt{fun}, n) = \texttt{genBody}(\texttt{fun}, n) \mid (\texttt{fun}, n) \in \mathrm{dom}(\Delta)\}$
  in $\boldsymbol{main_0}\langle\rangle$

Algorithm 2: Pseudo-code of the overall algorithm.

continuation of the select statement is stored in ρ_c, while the guard and body of each case is stored into ρ_i and ρ'_i, respectively. If the select statement contains a default case (b = **nonblocking**), we additionally translate the last block of the chain into a type function call. The guard and body of each case is then appended with the type function call corresponding to the continuation and all the components are packaged into an external choice construct.

Function calls. If s is a function call, we create a corresponding type definition call using the auxiliary function $mkCall_\Gamma(\texttt{fun}, \tilde{x})$, which defined as follows:

$$mkCall_\Gamma(\texttt{fun}, \tilde{x}) = \begin{cases} \boldsymbol{fun_0}\langle\Gamma(\tilde{x}), \tilde{v}\rangle & \text{if } \Delta(\texttt{fun}, 0) = \boldsymbol{fun_0}(\tilde{y}, \tilde{v}) \\ & \text{and } \forall x \in \tilde{x} \, : \, \Gamma(x) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

Note that the arguments are replaced by their representatives according to Γ and the function is undefined if any of them refers to an uninitialised channel.

Goroutines. If s spawns a new goroutine, we proceed similarly to the function call case but place the call in parallel with its continuation, which is computed through a call to `genFunction`, starting with an empty behavioural type (denoted by $\circ$).

Aliasing. If s stores a *channel* variable into another, we update the context with $\Gamma[\texttt{t0} \mapsto \Gamma(\texttt{t1})]$ (which is undefined if $\Gamma(\texttt{t0}) \neq \bot$).

Phi. If s is a phi statement, we proceed only if it does not overwrite channels – we discuss how to lift this restriction in § 5.

We skip all other statements as they do not pertain to communication or concurrency.

3.2.2 Top-level procedure: `genEquations`*.* Finally, we generate the body of type definitions using the `genFunction` procedure and thus obtain a set of recursive equations as required. Algorithm 2 gives the overall generation process. We iterate over each type signature (and therefore each SSA block) to generate a type implementation starting with a context Γ which is initialised to the identity function for each of the parameters. The algorithm returns a set of (possibly mutually recursive) type definitions, whose entry point is the program entry point, e.g., $\boldsymbol{main_0}\langle\rangle$.

Example 3.1. Consider the Go program from Listing 1 and its SSA representation in Figure 3. The set of type definitions infered from this example is given below.

$$\begin{aligned}
&\boldsymbol{main_0}() = (\mathsf{new}\, t0); (\boldsymbol{sendFn_0}\langle t0\rangle \mid \boldsymbol{recvVal_0}\langle t0\rangle; \boldsymbol{main_3}\langle t0\rangle)\\
&\boldsymbol{main_1}(t0) = \boldsymbol{main_3}\langle t0\rangle\\
&\boldsymbol{main_2}(t0) = \mathsf{close}\, t0; \mathbf{0}\\
&\boldsymbol{main_3}(t0) = \boldsymbol{main_1}\langle t0\rangle \oplus \boldsymbol{main_2}\langle t0\rangle\\
&\boldsymbol{sendFn_0}(c) = \overline{c}; \mathbf{0}\\
&\boldsymbol{recvVal_0}(c) = c; \mathbf{0}
\end{aligned}$$

Note that the spawning of the goroutine `sendFn(ch)` becomes a parallel composition of the main thread with $\boldsymbol{sendFn_0}\langle t0\rangle$.

Example 3.2. Consider the Go program from Listing 2 and its SSA representation in Figure 4. Its inferred type definitions are given below.

$$\begin{aligned}
\boldsymbol{myselect_0}(c) &= \&\{c; \boldsymbol{myselect_2}\langle c\rangle; \boldsymbol{myselect_1}\langle c\rangle,\\
&\qquad \tau; \boldsymbol{myselect_4}\langle c\rangle; \boldsymbol{myselect_1}\langle c\rangle,\\
&\qquad \tau; \boldsymbol{myselect_5}\langle c\rangle; \boldsymbol{myselect_1}\langle c\rangle\}\\
\boldsymbol{myselect_i}(c) &= \mathbf{0} \qquad \text{for } \boldsymbol{i} \in \{1, 2, 4, 5\}\\
\boldsymbol{myselect_3}(c) &= \boldsymbol{myselect_4}\langle c\rangle \oplus \boldsymbol{myselect_5}\langle c\rangle
\end{aligned}$$

The type's entry point is $\boldsymbol{myselect_0}(c)$ and $\boldsymbol{myselect_3}(t0)$ is unused. Note how each branch of the select consists of the sequential composition of a guard, a type function call corresponding to the body of the branch, and a call to the continuation $\boldsymbol{myselect_1}\langle t0\rangle$.

4 MODEL CHECKING BEHAVIOURAL TYPES

We present our model checking based analysis of the *finite control* fragment of behavioural types. We proceed in three steps: (1) we generate a (finite) labelled transition system (LTS) for the types from a set of operational semantics rules; (2) we define properties of the states of the LTS in terms of the immediate actions behavioural types can take; and (3) we give safety and liveness properties expressed in the modal μ-calculus [28].

The notion of finite control has several definitions in the literature [8, 11] but is generally understood to refer to having *finitely* many reachable states (possibly up-to some equivalence relation). Here we adopt the definition of finite control used by the `mCRL2` toolchain [10]: types cannot feature parallel composition or channel creation operators under recursion, which is a sufficient condition to guarantee a finite state space. For instance, types of the form $\boldsymbol{t}(\tilde{x}) = \boldsymbol{t}\langle\tilde{x}\rangle \mid T$ or $\boldsymbol{t}(x) = (\mathsf{new}\, a); \boldsymbol{t}\langle a\rangle$ are *not* finite control as the former generates infinitely many instances of type $\boldsymbol{t}(\tilde{x})$ while the latter generates infinitely many channels.

Semantics of types. Before proceeding to Step (1), i.e., the generation of a labelled transition system (LTS) from behavioural types, we introduce the semantics of types. The semantics follows standard definitions from CCS and CSP, accounting for the constructs that are specific to the Go programming language. The labels, ranged over by α and β, have the form:

$$\alpha, \beta \coloneqq \overline{a} \mid a \mid \tau \mid \tau_a \mid \mathsf{clo}\, a \mid \overline{\mathsf{clo}}\, a \mid a^\star \mid {}^\bullet a \mid a^\bullet$$

and their meaning is given in Table 2.

We assume types are in *normal form*, with all channel creations at the outermost top level. In a finite control setting we can always soundly rewrite types to satisfy this normal form using the equivalences defined in Figure 6. Thus, a program's type is always of the form:

$$\{\boldsymbol{t}_i(\tilde{y}_i) = T_i\}_{i \in I} \;\mathsf{in}\; (\mathsf{new}^{n_0}\, a_0) \ldots (\mathsf{new}^{n_k}\, a_k); \boldsymbol{t}_0\langle\rangle$$

where the several T_i contain no channel creations. We also make use of the following transition which initialises all the channels accordingly and write $\mathcal{A}$ for the set of all initialised channels:

$$\begin{aligned}
&\{\boldsymbol{t}_i(\tilde{y}_i) = T_i\}_{i \in I} \;\mathsf{in}\; (\mathsf{new}^{n_0}\, a_0) \ldots (\mathsf{new}^{n_k}\, a_k); \boldsymbol{t}_0\langle\rangle\\
&\xrightarrow{\tau} \quad \{\boldsymbol{t}_i(\tilde{y}_i) = T_i\}_{i \in I} \;\mathsf{in}\; (\boldsymbol{t}_0\langle\rangle \mid \lfloor a_0 \rfloor_0^{n_0} \mid \ldots \mid \lfloor a_k \rfloor_0^{n_k})
\end{aligned}$$

$$\overline{a};T \xrightarrow{\overline{a}} T \qquad a;T \xrightarrow{a} T \qquad \tau;T \xrightarrow{\tau} T$$

$$\texttt{close}\,a;T \xrightarrow{\texttt{clo}\,a} T \qquad \lfloor a \rfloor_k^n \xrightarrow{\overline{\texttt{clo}\,a}} a^\star \qquad a^\star \xrightarrow{a^\star} a^\star$$

$$\frac{i \in \{1,2\}}{T_1 \oplus T_2 \xrightarrow{\tau} T_i} \qquad \frac{\alpha_j;T_j \xrightarrow{\alpha_j} T_j \quad j \in I}{\&\{\alpha_i;T_i\}_{i\in I} \xrightarrow{\alpha_j} T_j}$$

$$\frac{T \xrightarrow{\alpha} T'}{T \mid S \xrightarrow{\alpha} T' \mid S} \qquad \frac{T \xrightarrow{\alpha} T'}{T;S \xrightarrow{\alpha} T';S} \qquad \mathbf{0};S \xrightarrow{\tau} S$$

$$\frac{\alpha \in \{\overline{a}, a^\star, a^\bullet\} \quad T \xrightarrow{\alpha} T' \quad S \xrightarrow{\beta} S' \quad \beta \in \{{}^\bullet a, a\}}{T \mid S \xrightarrow{\tau_a} T' \mid S'}$$

$$\frac{T \equiv_\alpha T' \quad T \xrightarrow{\alpha} T''}{T' \xrightarrow{\alpha} T''} \qquad \frac{T\{\tilde{a}/\tilde{x}\} \xrightarrow{\alpha} T' \quad \boldsymbol{t}(\tilde{x}) = T}{\boldsymbol{t}\langle\tilde{a}\rangle \xrightarrow{\alpha} T'}$$

$$\frac{T \xrightarrow{\texttt{clo}\,a} T' \quad S \xrightarrow{\overline{\texttt{clo}\,a}} S'}{T \mid S \xrightarrow{\tau} T' \mid S'} \qquad \frac{k < n}{\lfloor a \rfloor_k^n \xrightarrow{{}^\bullet a} \lfloor a \rfloor_{k+1}^n} \qquad \frac{k \geq 1}{\lfloor a \rfloor_k^n \xrightarrow{a^\bullet} \lfloor a \rfloor_{k-1}^n}$$

Figure 5: Semantics of types.

$$\mathbf{0};T \equiv T \qquad T \mid S \equiv S \mid T \qquad T \mid (T' \mid S) \equiv (T \mid T') \mid S$$
$$T \mid \mathbf{0} \equiv T \qquad T \equiv_\alpha T' \Rightarrow T \equiv T'$$
$$(\mathsf{new}^n\,a);(\mathsf{new}^m\,b);T \equiv (\mathsf{new}^m\,b);(\mathsf{new}^n\,a);T$$
$$(\mathsf{new}^n\,a);\mathbf{0} \equiv \mathbf{0} \qquad (\mathsf{new}^n\,a);a^\star \equiv \mathbf{0} \qquad (\mathsf{new}^n\,a);\lfloor a \rfloor_k^n \equiv \mathbf{0}$$
$$T \mid (\mathsf{new}^n\,a)S \equiv (\mathsf{new}^n\,a)(T \mid S) \quad (a \notin \mathsf{fn}(T))$$

Figure 6: Structural congruence for types.

We give the semantic rules for behavioural types in Figure 5, adapted from [30], where $T \xrightarrow{\alpha} T'$ denotes that T reduces to T' by producing α, according to the rules in Figure 5. In the first line, the rules respectively model send, receive and silent actions. In the second line, the rules respectively model close actions, the closure of channel a and a closed buffer sending default values. In the third line, the rules respectively model a silent transition representing an *internal* choice and an *external* choice. The fourth line gives the standard rules for parallel and general sequencing. The rule in the fifth line models the synchronisation between a type or buffer firing a send-like action (i.e., a send action, a closed buffer, or a non-empty asynchronous buffer) and a receive action or a non-full buffer. The sixth line gives standard rules to deal with α-equivalence and unfolding of definition calls. In the seventh line, the rules respectively model the synchronisation of a type and a buffer a to effectively close a, and the action of adding (resp. removing) an element in (resp. from) a buffer, where n is the capacity of the queue and k is the number of messages currently stored in the queue. We have omitted symmetric rules for parallel and synchronisations.

In Step (1), given a *finite control* type in normal form, we construct a *finite* labelled transition system which represents *all* possible executions of $\boldsymbol{t}_0\langle\rangle$, i.e., the entry point type under all the name restrictions. The LTS of $\boldsymbol{t}_0\langle\rangle$ is a tuple $\mathcal{T} = (\mathcal{S}, \boldsymbol{t}_0\langle\rangle, \rightarrow, \mathbb{A})$ such that $\mathcal{S}$ is a set of states implicitly labelled by behavioural type terms (we often identify labels and states), $\boldsymbol{t}_0\langle\rangle \in \mathcal{S}$ is the initial state,

Table 2: (Predicate) labels

$\overline{a}$ / a	send / receive on channel a
τ_a	synchronisation over a
τ	silent action
$\texttt{clo}\,a$ / $\overline{\texttt{clo}\,a}$	request to close a / closing a
$a^\star$	channel a is closed
${}^\bullet a$ / $a^\bullet$	push / pop on buffer a
$\tilde{o}$	waiting to synchronise over the actions in $\tilde{o}$

$$a;T \downarrow_a \quad \texttt{close}\,a;T \downarrow_{\texttt{clo}\,a} \quad \frac{\forall i \in \{1,\ldots,n\} : \alpha_i \downarrow_{o_i}}{\&\{\alpha_i;T\}_{i\in\{1,\ldots,n\}} \downarrow_{\{o_1\ldots o_n\}}}$$
$$\overline{a};T \downarrow_{\overline{a}} \qquad a^\star \downarrow_{a^\star}$$

$$\frac{T \downarrow_o}{T;T' \downarrow_o} \qquad \frac{T \downarrow_a \quad T' \downarrow_{\overline{a}} \text{ or } T' \downarrow_{a^\star}}{T \mid T' \downarrow_{\tau_a}} \qquad \frac{T\{\tilde{a}/\tilde{x}\} \downarrow_o \quad \boldsymbol{t}(\tilde{x}) = T}{\boldsymbol{t}\langle\tilde{a}\rangle \downarrow_o}$$

$$\frac{T \downarrow_a \quad \alpha_i \downarrow_{\overline{a}}}{T \mid \&\{\alpha_i;S_i\}_{i\in I} \downarrow_{\tau_a}} \qquad \frac{T \downarrow_{\overline{a}} \text{ or } T \downarrow_{a^\star} \quad \alpha_i \downarrow_a}{T \mid \&\{\alpha_i;S_i\}_{i\in I} \downarrow_{\tau_a}}$$

$$\frac{k < n}{\lfloor a \rfloor_k^n \downarrow_{{}^\bullet a}} \qquad \frac{k \geq 1}{\lfloor a \rfloor_k^n \downarrow_{a^\bullet}} \qquad \frac{T \downarrow_{\overline{a}} \quad T' \downarrow_{{}^\bullet a}}{T \mid T' \downarrow_{\tau_a}} \qquad \frac{T \downarrow_{a^\bullet} \quad \alpha_i \downarrow_a}{T \mid \&\{\alpha_i;S_i\}_{i\in I} \downarrow_{\tau_a}}$$

$$\frac{T \downarrow_o}{T \mid T' \downarrow_o} \qquad \frac{T \downarrow_o \quad a \notin \mathsf{fn}(o)}{(\mathsf{new}^n\,a);T \downarrow_o} \qquad \frac{T \downarrow_o \quad T \equiv T'}{T \downarrow_o}$$

Figure 7: Barb predicates for types.

$\mathbb{A} \subseteq \{\tau\} \cup \{\tau_a \mid a \in \mathcal{A}\}$ is the set of labels, and $\rightarrow \subseteq \mathcal{S} \times \mathbb{A} \times \mathcal{S}$ is the transition relation $T \xrightarrow{\alpha} T'$ where the label α can be either a silent move, i.e., τ, or a synchronisation over a channel, e.g., τ_a.

Properties of behavioural type states. In Step (2), we define predicates over the state labels of the LTS defined above. This allows us to analyse what actions a given state (or type) can fire immediately. Concretely, we define a family of predicates of the form $T \downarrow_o$ or $T \downarrow_{\tilde{o}}$ which holds if T is ready to fire action o or one of the actions in $\tilde{o}$, with $o, o_i \in \{a, \overline{a}, \tau_a, \texttt{clo}\,a, a^\star, {}^\bullet a, a^\bullet\}$. Table 2 explains the meaning of each label and Figure 7 gives the defining rules of the predicates $T \downarrow_o$ and $T \downarrow_{\tilde{o}}$. Essentially, $T \downarrow_o$ if T is immediately ready to fire action o (with $o \neq \tau$) and $T \downarrow_{\tilde{o}}$ if T contains an external choice which does not feature a branch guarded by τ (i.e., $\tau \notin \tilde{o}$). We have, e.g., $\neg(\tau;T \downarrow_o)$ for any o and $\neg(\&\{\tau;T_1, \overline{a};T_2\} \downarrow_{\tilde{o}})$ for any $\tilde{o}$, which is an important subtlety for defining accurate safety and liveness properties.

Liveness and safety properties. In Step (3), we encode liveness and channel properties (including those discussed in § 2.1) in the μ-calculus [28] extended with the atomic propositions on state labels defined in Step (2).

A μ-calculus formula ϕ is interpreted on a pointed LTS, i.e., an LTS with a starting state T, we write $T \models_{\mathcal{T}} \phi$ if T satisfies ϕ in the LTS $\mathcal{T}$. Namely, formula $\top$ holds for every T (while $\bot$ never holds). The construct $[\alpha]\phi$ is a *modal* operator that is satisfied if, for each α-derivative T' of T (i.e. T' is reachable from T by performing action α), the formula ϕ holds in T'. The dual modality is $\langle\alpha\rangle\phi$ which holds if there is an α-derivative T' of T such that ϕ holds in T'. Construct $\nu\mathbf{x}.\,\phi$ (resp. $\mu\mathbf{x}.\,\phi$) is the standard *greatest* (resp.

$$\Psi(\phi) \stackrel{\text{def}}{=} \nu\mathbf{x}.\,(\phi \wedge [\mathbb{A}]\mathbf{x}) \qquad \text{[Always]}$$
$$\Phi(\phi) \stackrel{\text{def}}{=} \mu\mathbf{y}.\,(\phi \vee \langle\mathbb{A}\rangle\mathbf{y}) \qquad \text{[Eventually]}$$

$$\psi_t \stackrel{\text{def}}{=} \langle\mathbb{A}\rangle\top \qquad \text{[No terminal]}$$
$$\psi_c \stackrel{\text{def}}{=} \mu\mathbf{y}.\,[\mathbb{A}]\mathbf{y} \qquad \text{[No cycle]}$$
$$\psi_g \stackrel{\text{def}}{=} (\textstyle\bigwedge_{a\in\mathcal{A}} \downarrow_a \vee \downarrow_{\overline{a}}) \Longrightarrow \langle\mathbb{A}\rangle\top \qquad \text{[No global deadlock]}$$
$$\psi_{l_a} \stackrel{\text{def}}{=} (\textstyle\bigwedge_{a\in\mathcal{A}} \downarrow_a \vee \downarrow_{\overline{a}}) \Longrightarrow \Phi(\langle\tau_a\rangle\top) \qquad \text{[Liveness (a)]}$$
$$\psi_{l_b} \stackrel{\text{def}}{=} (\textstyle\bigwedge_{\tilde{a}\in\mathcal{P}(\mathcal{A})} \downarrow_{\tilde{a}}) \Longrightarrow \Phi(\langle\{\tau_a \mid a \in \tilde{a}\}\rangle\top) \qquad \text{[Liveness (b)]}$$
$$\psi_s \stackrel{\text{def}}{=} (\textstyle\bigwedge_{a\in\mathcal{A}} \downarrow_{a^\star}) \Longrightarrow \neg(\downarrow_{\overline{a}} \vee \downarrow_{\mathtt{clo}\,a}) \qquad \text{[Channel safety]}$$
$$\psi_e \stackrel{\text{def}}{=} (\textstyle\bigwedge_{a\in\mathcal{A}} \downarrow_{a^\bullet}) \Longrightarrow \Phi(\langle\tau_a\rangle\top) \qquad \text{[Eventual reception]}$$

Figure 8: μ-calculus formulae.

smallest) fixpoint operator (binding $\mathbf{x}$ in ϕ). The atomic proposition $\downarrow_o$ (resp. $\downarrow_{\tilde{o}}$) holds iff $T \downarrow_o$ (resp. $T \downarrow_{\tilde{o}}$). Given a set of actions $A \subseteq \mathbb{A}$, we write $[A]\phi$ for $\bigwedge_{\alpha\in A}[\alpha]\phi$ and $\langle A\rangle\phi$ for $\bigvee_{\alpha\in A}\langle\alpha\rangle\phi$.

We now describe several properties which can be verified using the model checker `mCRL2` [10]. Below we refer to the formulae given in Figure 8. Given a μ-calculus formula ϕ, formula $\Psi(\phi)$ holds if ϕ holds for all reachable states, while formula $\Phi(\phi)$ holds if ϕ holds in some reachable state. Formula $\Psi(\psi_t)$ holds if ***no terminal state*** is reachable in $\mathcal{T}$, i.e., the behavioural types only exhibit infinite executions; dually formula ψ_c holds if there are ***no cycles*** in $\mathcal{T}$, this property is useful as it implies that liveness of types corresponds with liveness of programs (cf. § 2.2). Formula $\Psi(\psi_g)$ models the ***global deadlock-freedom*** property discussed in § 2.1, i.e., the formula holds if for each state T in $\mathcal{T}$ if T is ready to execute a send or receive action, then T has a successor. Formula $\Psi\left(\psi_{l_a} \wedge \psi_{l_b}\right)$ models the ***liveness*** property (cf. § 2.1). It holds if for all state T in $\mathcal{T}$ (*i*) if T is ready to send/receive on a, there is always eventually a synchronisation on a (cf. ψ_{l_a}), and (*ii*) if T has a select construct, which does not contain a τ-branch, there is always eventually a synchronisation over one of the channels guarding the construct (cf. ψ_{l_b}). Formula $\Psi(\psi_s)$ models the ***channel safety*** property, i.e., no send nor close action is executed on a channel that is already closed. Formula $\Psi(\psi_e)$ models ***eventual reception***, which guarantees that when a channel is not empty, the head of its buffer can eventually be consumed.

5 IMPLEMENTATION

We present the `Godel Checker` toolchain of Figure 2 which consists of two core components: an inference tool and a type verifier.

Inference tool. The type inference tool implements the core algorithms described in § 3, with additional adjustments to support analysis of real world Go programs, which we discuss below.

Uninitialised channels: Uninitialised channels (or nil channels) can be used in Go, but they always block on communication. To model this behaviour, we prefix any communication on an uninitialised channel with a (new a) construct (with a fresh).

Composite data structures: Our tool supports channels that are stored in `struct`s by *flattening* such constructs into several channels. We only support structures that store finitely many channels (e.g., arrays or linked-list of channels are not supported).

Uniform representation of functions: A uniform representation of callable objects is used as an abstraction when obtaining the type signature of an SSA block. This allows us to support return values and closures by uniformly converting return values and closure binding as additional function parameters.

Channels in phi instructions: We support SSA instructions to merge control flow (`phi`) when they refer to channels by adding a parameter to the type definition of its enclosing block and modifying function calls accordingly.

Type verifier. The type verifier transforms the inferred behavioural types into an LTS and properties into μ-calculus formulae following the methodology in § 4 for the `mCRL2` model checker, and also into input for the `KiTTeL` termination analyser.

Model checking: Once a behavioural type has been inferred from Go source code, we translate it straightforwardly to the `mCRL2` language [19]. Before generating the μ-calculus formulae described in § 4, we analyse the model so to build the smallest formulae possible. Finally, we run the `mCRL2` model checker for each formula and return the result to the user.

Termination checking: To address the mismatch between types and programs detailed in § 2.2 we deploy a termination analysis of loops, using the `KITTeL` termination analyser [16]. The tool targets C programs and is based on integer term rewriting. The choice of this particular analyser amounts to the syntax of Go being close to C, its usability and performance.

The analysis takes advantage of the inference procedure of § 3 to collect the locations and parameters of loops in a given program, which are then checked for termination. Our analysis checks that the loop parameters are enough to make each loop eventually terminate, regardless of the non-loop code within the loop itself. This enables us to pinpoint program locations where liveness of types may not entail the analogue property in the program – if the termination analysis identifies the program as terminating, the liveness properties on types and programs coincide [30, § 5].

The analysis generates all loops in the original Go program as a set of C functions, *ignoring* all other Go statements. Each C function (and thus, each loop) is then individually checked for termination. Since loops can be nested, our analysis takes this into account by replicating the nesting in the generated C functions. For instance, for the following code snippet,

```
1  func f(n int) {
2    for i := 0; i < n; i++ {
3      for j := 0; j < 10; j++ { ... }
4    }
5  }
```

our tool generates a single C function `f` containing the two loops. Statically unknown values in loop parameters (e.g. the parameter `n` of function `f` above) are generated as parameters of the respective C functions. This forces the termination checker to verify termination *for all* possible values of the unknown parameter. Such values can appear due to usages of function arguments, values contained in dynamic data structures or communicated data.

Our analysis relies on the following: loops in Go programs generate types with conditional branching combined with recursion; most programs use traditional imperative control flow features such as for loops, for-range loops (i.e. loops over a fixed finite data structure) and for-select loops (i.e. an infinite loop with a `select`

Table 3: Go programs verified by our framework and comparison with existing static deadlock detection tools.

			Godel Checker									dingo-hunter [36]		gopherlyzer [40]		GoInfer/Gong [30]		
	Programs	LoC	# states	ψ_g	ψ_l	ψ_s	ψ_e	Infer	Live	Live+CS	Term	Live	Time	DF	Time	Live	CS	Time
1	`mismatch` [36]	29	53	×	×	✓	✓	620.7	996.8	996.7	✓	×	639.4	×	3956.4	×	✓	616.8
2	`fixed` [36]	27	16	✓	✓	✓	✓	624.4	996.5	996.3	✓	✓	603.1	✓	3166.3	✓	✓	601.0
3	`fanin` [36, 39]	41	39	✓	✓	✓	✓	631.1	996.2	996.2	✓	✓	608.9	✓	19.8	✓	✓	696.7
4	`sieve` [30, 36]	43	∞		*n/a*			-	-	-	*n/a*	*n/a*	-	*n/a*	-	✓	✓	778.3
5	`philo` [40]	41	65	×	×	✓	✓	6.1	996.5	996.6	✓	×	34.2	×	27.0	×	✓	16.8
6	`dinephil3` [13, 33]	55	3838	✓	✓	✓	✓	645.2	996.4	996.3	✓	*n/a*	-	*n/a*	-	✓	✓	13.2 min
7	`starvephil3`	47	3151	×	×	✓	✓	628.2	996.5	996.5	✓	*n/a*	-	*n/a*	-	×	✓	3.5 min
8	`sel` [40]	22	103	×	×	✓	✓	4.2	996.7	996.6	✓	×	15.3	×	13.0	×	✓	50.5
9	`selFixed` [40]	22	20	✓	✓	✓	✓	4.0	996.3	996.4	✓	✓	14.9	✓	3168.3	✓	✓	13.1
10	`jobsched` [30]	43	43	✓	✓	✓	✓	632.7	996.7	1996.1	✓	*n/a*	-	✓	4753.6	✓	✓	635.2
11	`forselect` [30]	42	26	✓	✓	✓	✓	623.3	996.4	996.3	✓	✓	611.8	*n/a*	-	✓	✓	618.6
12	`cond-recur` [30]	37	12	✓	✓	✓	✓	4.0	996.2	996.2	✓	✓	9.4	*n/a*	-	✓	✓	14.7
13	`concsys` [42]	118	15	×	×	✓	✓	549.7	996.5	996.4	✓	*n/a*	-	×	5278.6	×	✓	521.3
14	`alt-bit` [30, 35]	70	112	✓	✓	✓	✓	634.4	996.3	996.3	✓	*n/a*	-	*n/a*	-	✓	✓	916.8
15	`prod-cons`	28	106	✓	×	✓	✓	4.1	996.4	1996.2	✓	×	10.1	×	30.1	×	✓	21.8
16	`nonlive`	16	8	✓	✓	✓	✓	630.1	996.6	996.5	timeout	⊗	613.6	*n/a*	-	⊗	✓	613.8
17	`double-close`	15	17	✓	✓	×	✓	3.5	996.6	1996.6	✓	⊠	8.7	⊠	11.8	✓	×	9.1
18	`stuckmsg`	8	4	✓	✓	✓	×	3.5	996.6	996.6	✓	*n/a*	-	*n/a*	-	✓	✓	7.6
19	`dinephil5`	61	~1M	✓	✓	✓	✓	626.5	41.2 sec	41.4 sec	✓	*n/a*	-	*n/a*	-	timeout		>48 hrs
20	`prod3-cons3`	40	57493	✓	✓	✓	✓	465.1	40.9 sec	40.9 sec	✓	*n/a*	-	*n/a*	-	timeout		>48 hrs
21	`async-prod-cons`	33	164897	✓	✓	✓	✓	4.3	47.7 sec	89.4 sec	✓	*n/a*	-	*n/a*	-	timeout		>48 hrs
22	`astranet` [26]	~18k	1160	✓	✓	✓	✓	2512.5	70.4 sec	75.0 sec	✓	*n/a*	-	*n/a*	-	*n/a*		-
	Column		4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

CS: Channel Safe, Term: Termination check, DF: Deadlock-free, timeout: Termination check timeout (likely does not terminate), ⊠: False Alarm, ⊗: Undetected liveness error.

that can break the loop – the `Consumer` function of Figure 1) instead of recursion; we assume that loop indices are not modified in loop bodies and that no **`goto`**-like constructs are used in a loop.

Since the analysis only takes into account loop parameters, a loop that indefinitely blocks (e.g. due to communication) may be identified as terminating. However, if our analysis identifies the inferred types as live *and* the termination check validates the program, both termination and program liveness are guaranteed.

6 EVALUATION

Table 3 lists several benchmarks of our tool against other static deadlock detection tools for Go (a detailed comparison of these tools is given in § 7). The benchmarks were run with go1.8.3 on an 8-core Intel i7-3770 machine with 16GB RAM on a 64-bit Linux. The model checker we used was mCRL2 v201707.1.

The results for `Godel Checker` are shown in columns 4–12. Column 4 shows the number of states in the input LTS as a measurement of the relative complexity of each program (proportional to the number of concurrency-related operations rather than the number of lines of code). Columns 5–8 shows the core formulae of Figure 8 in § 4, i.e. no global deadlock (ψ_g), liveness (ψ_l), channel safety (ψ_s) and eventual reception (ψ_e). A ✓ mark means that the considered tool reports that the property holds. For example, in the case of `Godel Checker`, a ✓ mark under column ψ_g means that the formula $\Psi(\psi_g)$ evaluates to `true`. In the case of `GoInfer/Gong` a ✓ mark under column Live means that the tool reports the program to be live, as defined in [30]. A × mark indicates that the considered tool reports that the property in question does not hold. Columns 9–12 list the running time of `Godel Checker`, where Column 10 lists the inference time, Columns 10 and 11 are the model checking times for liveness, and both liveness and channel safety, respectively. The total run time can be obtained by adding Column 9 to Column 10 or 11. Unless otherwise stated, all times are in milliseconds. Column 12 (Term) shows the result of the termination check, which proves the termination of loops in the given program, or times out after 15s. A program that times out is conservatively assumed not to terminate.

Columns 13–14 pertain to the `dingo-hunter` tool from [36]. The time includes both communicating finite state machine extraction and their analysis, but does not include building the global graph and only checks for liveness. Columns 15–16 pertain to the `gopherlyzer` tool [40], which only checks for global deadlock-freedom (most programs had to be manually adjusted in order to be accepted by this tool – see § 7 for the severe practical limitations of the tool). Columns 17–19 refer to the `GoInfer/Gong` tool from [30]. The times include both type inference and analysis stages, which only accounts for liveness and channel safety checks. Most programs in Table 3 are taken either from other papers on the static verification of Go programs [30, 36, 40] or from publicly available source code. Programs 7, and 15–22 are benchmarks introduced by this work. Programs that are unsupported by a tool are marked with *n/a*. Table 3 includes all the benchmarks from previous works, except for 3 non-finite control examples from [30] which our tool cannot analyse (i.e., like `sieve`) and `htcat` from [36] which is manually transformed to be supported by `dingo-hunter`.

Programs 1–7 are typical concurrent programs from the literature. The `sieve` program is not finite control (it spawns an infinite number of threads), thus it can only be analysed by `GoInfer/Gong`. Program 6 is a (three) dining philosophers program where the first fork can be released, while Program 7 is the traditional deadlocking version (Program 19 is as Program 6 but with 5 philosophers). `dingo-hunter` does not support Programs 6, 7, and 19 due to dynamically spawned goroutines, while `gopherlyzer` does not support them due to a nested select statement. `GoInfer/Gong` analyses them correctly, but is much slower than `Godel Checker`.

Programs 8–12 consist of idiomatic Go patterns which are all handled correctly and quickly by our tool. Program 13 is a publicly available program which is not live. Program 14 is an implementation of the alternating bit protocol. Program 15 is the Producer-Consumer example from § 1, which is not live. All tools were able to verify this simple program. Program 16 demonstrates the mismatch between type and program liveness, where the type is live but due to an erroneous loop the program does not terminate and causes a partial deadlock. The termination check identifies this as possibly non-terminating, while `GoInfer/Gong` incorrectly identifies it as live. Program 17 closes a channel twice which flags a violation of channel safety in `Godel Checker` and `GoInfer/Gong`. Interestingly, `dingo-hunter` detects a deadlock (a false alarm) due to its representation of channel closure as a message exchange, but not due to the double close. `gopherlyzer` also detects a deadlock incorrectly due to the same reason. Program 18 is a program that violates the *eventual reception* property by sending an asynchronous message that is never received – none of the earlier tools can detect this.

Programs 19–22 demonstrate the scalability of our approach. Program 22 is a concurrent data stream multiplexer, for handling multiple independent data streams in a single TCP connection. It consists of 16k lines of code of which only a small portion relate to concurrency (which is the case with common concurrent Go applications). The program is not natively finite control since it spawns request handlers as goroutines in a loop. Noting that request handlers do not interact with each other, we modified the program to handle requests sequentially and enable our analysis.

We note that while the execution time for small programs is slightly higher than the other tools (but still under 2 seconds), `Godel Checker` is a more general tool since it can verify *arbitrary* properties expressible in the μ-calculus and our precise inference allows us to reduce both the false alarms and, crucially, *undetected liveness errors*. The verification times also suffer from the initialisation of `mCRL2` (the tool uses 3 binaries). In small programs the running times are generally dominated by this fact, which is the reason why the times are quite similar. This is amortised in programs with large state spaces (cf. second part of Table 3), where the efficiency of `mCRL2` produces gains of several orders of magnitude over `gopherlyzer` and `GoInfer/Gong` while performing more detailed analyses. We note that a significant portion of the inference time is due to the translation into SSA by the `ssa` package.

Limitations. As explained in § 3, our inference does not support channel variable overwriting, i.e., we only support *immutable* channel variables. In addition, it does not support channels in dynamic data structures – such as arrays, slices (variable sized arrays) and dictionaries – or recursively defined data structures (e.g. linked lists). However, our tool ensures that such data structures do not contain channels and can be safely ignored, signalling an error otherwise. We also require channel buffer sizes to be statically known. While the inference is agnostic to the finite control limitation of the model checking tool, if a type is inferred successfully, it must be finite control in order for our type verifier to produce an output. We note that these limitations are also present in the other tools mentioned in this section (`GoInfer/Gong` supports infinite state systems by performing a bounded verification).

7 RELATED WORK AND CONCLUSION

Applying program analyses to real-world software. The error-prone nature of concurrent software has led to a plethora of works on automated verification of concurrency via program analysis. However, these works mostly target lock-based concurrency (such as those based on Java Pathfinder [3, 21, 38, 44] or abstract interpretation [29]) and so are of a fundamentally different nature than our work targetting message-based concurrency in Go.

Verification of Go programs. Despite the young age of Go, it has received some attention from the research community. The work of [36] is to the best of our knowledge the first to tackle static verification of Go programs. Their work uses multiparty session types [23] and their connection to communicating automata [12, 32] to check for liveness in Go by extracting communicating finite state machines from source code. However, their work cannot support dynamic spawning of goroutines (requiring all goroutines to be executing before any communication takes place) nor asynchrony. This severely limits the applicability of their work. Their analysis also does not cover many features of Go which results in crashes in the analysis, such as `phi` instructions and uninitialised channels.

Using a form of regular expressions with a fork construct, the work of [40] captures thread spawning in *synchronous* Go programs. Their analysis is extremely limited: it does not support asynchrony, closing channels nor selective communication with non-trivial case bodies. Moreover, their work uses the `guru` tool to manually obtain aliasing information in order to identify channels, and assumes that all functions can be inlined. As a result, their tool fails to analyse programs that cannot be trivially inlined — e.g. programs with aliased channel or repeated usages of the same function — ruling out most useful programs.

Our previous work [30] infers behavioural types from Go code which are checked for liveness and safety properties using a technique akin to bounded symbolic execution. The `GoInfer/Gong` tool explicitly executes the type LTS which has scalability issues with large state spaces (see § 6) and is specialised for liveness and channel safety, whereas our tool can check for a much wider range of properties of interest (in general we can verify any property that can be represented as a μ-calculus formula). The type inference of [30] (which was not formalised until this work) did not have full support for closures nor general sequencing, needed to accurately represent most imperative programming patterns.

Behavioural types. There is a vast body of work on behavioural types for concurrency (see [1, 24] for general surveys). The main contrast between our work and most of those in [1, 24] is that we use behavioural types as a component in a larger analysis that can automatically check for a range of safety and liveness properties, instead of focusing solely on forms of deadlock-freedom. The work [7] proposes a framework combining a behavioural type analysis with model checking. Their work uses the π-calculus as a source language and extracts CCS-like behavioural types based on [25], which can then be checked for properties written as an LTL formula. The main limitation of their work is the requirement of explicit type annotations in processes . Moreover, it is not clear how to represent our notions of global and partial deadlock-freedom (as well as channel safety) as a general LTL formula. LTL formulas can use "always" and "eventually" modalities to describe reachable

states, but cannot mention specific communication actions which requires non-obvious encodings. Note that most previous works [7, 24] are developed in the context of process calculi and are not applicable to a general purpose language.

Concluding remarks. We have presented a static verification framework for channel-based concurrency in Go which we have implemented in the `Godel Checker` tool. As shown in § 6, our inference procedure allows us to accurately cover a broader class of Go programs without the need for annotations or significant user input. By integrating our approach with a general purpose model checker, we are able to modularly verify arbitrary safety and liveness properties. Compared to other existing tools, our approach provides significantly better performance for larger programs, verifying more properties and with better outcomes in terms of both false alarms and, crucially, *undetected* liveness errors.

Given the general nature of our inference procedure, our framework is not necessarily limited to `mCRL2`, nor model checking techniques in general. For future work we plan to use other process calculi verification techniques such as [37], as well as other model checkers for concurrency such as [18]. Also, the general idea for our inference can in principle be applied to other concurrency-centric languages that rely on some form of SSA-like intermediate representation. We plan to apply our techniques to the Erlang language via the Core Erlang [6] intermediate representation. We also plan to address the OpenCL 2.0 heterogeneous programming framework [20] which provides *pipe* objects (akin to Go channels) that are used for inter-kernel communications and are prone to deadlocks.

ACKNOWLEDGMENTS

The work is partially supported by the EPSRC (grants EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1, and EP/N028201/1)

REFERENCES

[1] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. 2017. *Behavioral Types in Programming Languages.* Foundations and Trends in Programming Languages, Vol. 3(2-3). Now Publishers Inc. 95–230 pages.

[2] Andrew W. Appel. 1998. SSA is Functional Programming. *SIGPLAN Notices* 33, 4 (1998), 17–20. https://doi.org/10.1145/278283.278285

[3] Cyrille Artho, Masami Hagiya, Richard Potter, Yoshinori Tanabe, Franz Weitl, and Mitsuharu Yamamoto. 2013. Software model checking for distributed systems with selector-based, non-blocking communication. In *ASE*. 169–179.

[4] The Go Authors. 2013. package ssa. http://golang.org/x/tools/go/ssa. (2013). http://golang.org/x/tools/go/ssa.

[5] Brad Fitzpatrick. 2015. go 1.5.1 linux/amd64 deadlock detection failed. (9 2015). https://github.com/golang/go/issues/12734.

[6] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, and Robert Virding. 2000. Core Erlang 1.0 language specification. *Information Technology Department, Uppsala University, Tech. Rep* (2000).

[7] Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. 2002. Types as models: model checking message-passing programs. In *POPL*. 45–57.

[8] Witold Charatonik, Andrew D. Gordon, and Jean-Marc Talbot. 2002. Finite-Control Mobile Ambients. In *ESOP*. 295–313. https://doi.org/10.1007/3-540-45927-8_21

[9] CoreOS 2017. CoreOS. https://coreos.com/. (June 2017). https://coreos.com/.

[10] Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink, and Tim A. C. Willemse. 2013. *An Overview of the mCRL2 Toolset and Its Recent Advances.* Springer Berlin Heidelberg, Berlin, Heidelberg, 199–213. https://doi.org/10.1007/978-3-642-36742-7_15

[11] Mads Dam. 1996. Model Checking Mobile Processes. *Information and Computation* 129, 1 (1996), 35–51. https://doi.org/10.1006/inco.1996.0072

[12] Pierre-Malo Deniélou and Nobuko Yoshida. 2012. Multiparty Session Types Meet Communicating Automata. In *ESOP (LNCS)*, Vol. 7211. Springer, 194–213.

[13] E. W. Dijkstra. 1965. Cooperating sequential process. *Programming Languages* (1965), 43–112.

[14] Docker 2017. Docker. https://www.docker.com/. (June 2017).

[15] 2014. Open Sourcing our Go Libraries. https://blogs.dropbox.com/tech/2014/07/open-sourcing-our-go-libraries/. (July 2014).

[16] Stephan Falke, Deepak Kapur, and Carsten Sinz. 2011. Termination Analysis of C Programs Using Compiler Intermediate Languages. In *RTA*. 41–50.

[17] Stephan Falke, Deepak Kapur, and Carsten Sinz. 2012. *Termination Analysis of Imperative Programs Using Bitvector Arithmetic.* Springer Berlin Heidelberg, Berlin, Heidelberg, 261–277.

[18] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. 2014. FDR3 — A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, Vol. 8413. 187–201.

[19] Jan Friso Groote and Mohammad Reza Mousavi. 2014. *Modeling and Analysis of Communicating Systems.* MIT Press. https://mitpress.mit.edu/books/modeling-and-analysis-communicating-systems

[20] Khronos OpenCL Working Group. 2015. The OpenCL Specification Version 2.0. https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf. (2015).

[21] Klaus Havelund and Thomas Pressburger. 2000. Model Checking JAVA Programs using JAVA PathFinder. *STTT* 2, 4 (2000), 366–381.

[22] Tony Hoare. 1985. *Communicating Sequential Processes.* Prentice Hall.

[23] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *POPL'08*. ACM, 273–284. A full version in *JACM*: 63(1-9):1–67, 2016.

[24] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1, Article 3 (April 2016), 36 pages.

[25] Atsushi Igarashi and Naoki Kobayashi. 2004. A generic type system for the Pi-calculus. *Theor. Comput. Sci.* 311, 1-3 (2004), 121–163.

[26] Ilya Biin. 2017. AstraNet. (August 2017). https://github.com/zenhotels/astranet.

[27] K8S 2017. Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io/. (June 2017). https://kubernetes.io/.

[28] Dexter Kozen. 1983. Results on the Propositional mu-Calculus. *Theor. Comput. Sci.* 27 (1983), 333–354. https://doi.org/10.1016/0304-3975(82)90125-6

[29] Daniel Kroening, Daniel Poetzl, Peter Schrammel, and Björn Wachter. 2016. Sound static deadlock analysis for C/Pthreads. In *ASE*. 379–390.

[30] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2017. Fencing off Go: Liveness and Safety for Channel-based Programming. In *POPL 2017*. ACM, 748–761.

[31] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2017. Godel Checker. (2017). http://mrg.doc.ic.ac.uk/tools/godel-checker/.

[32] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. 2015. From Communicating Machines to Graphical Choreographies. In *POPL*. ACM, 221–232.

[33] Jeff Magee and Jeff Kramer. 1999. *Concurrency: State Models & Java Programs.* John Wiley & Sons, Inc., New York, NY, USA.

[34] Robin Milner. 1980. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science, Vol. 92. Springer.

[35] Robin Milner. 1989. *Communication and Concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

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

[37] Luca Padovani. 2014. Deadlock and Lock Freedom in the Linear π-Calculus. In *CSL-LICS'14*, Thomas A. Henzinger and Dale Miller (Eds.). ACM Press, 72:1–72:10. https://doi.org/10.1145/2603088.2603116

[38] Corina S. Pasareanu, Willem Visser, David H. Bushnell, Jaco Geldenhuys, Peter C. Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Autom. Softw. Eng.* 20, 3 (2013), 391–425.

[39] Sameer Ajmani. 2014. Go Concurrency Patterns: Pipelines and cancellation. (2014). https://blog.golang.org/pipelines.

[40] Kai Stadtmüller, Martin Sulzmann, and Peter Thiemann. 2016. Static Trace-Based Deadlock Analysis for Synchronous Mini-Go. In *APLAS*. 116–136.

[41] Nick Stenning. 2017. Building a new router for GOV.UK. https://gdstechnology.blog.gov.uk/2013/12/05/building-a-new-router-for-gov-uk/. (June 2017). https://gdstechnology.blog.gov.uk/2013/12/05/building-a-new-router-for-gov-uk/.

[42] Stillwater Supercomputing. 2017. Collection of Golang concurrency patterns. (June 2017). https://github.com/stillwater-sc/concurrency.

[43] The Go Authors. 2017. Effective Go. (June 2017). https://golang.org/doc/effective_go.html.

[44] Willem Visser, Klaus Havelund, Guillaume P. Brat, and Seungjoon Park. 2000. Model Checking Programs. In *The Fifteenth IEEE International Conference on Automated Software Engineering, ASE 2000*. 3–12.

[45] Kai Wei. 2016. How we built Uber engineering's highest query per second service using Go. https://eng.uber.com/go-geofence/. (2016).