Citation for published version


DOI

Link to record in KAR

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

Document Version

Author's Accepted Manuscript

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
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 calculus 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:

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

ICSE ’18, May 27-June 3, 2018, Gothenburg, Sweden  
© 2018 Copyright held by the owner(s).  
This is the author’s version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in ICSE ’18: ICSE ’18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden. https://doi.org/10.1145/3180155.3180157.

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 calculus 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 undetected errors. Our integration with a general purpose model checker also enables us to modularly verify arbitrary safety and
func prod(ch chan int) {
    for i := 0; i < 5; i++ { ch <- i } // Send i to ch
    close(ch) // No further values accepted at ch
}

func cons(ch1, ch2 chan int) {
    x := <-ch2 // or input from ch2
    // No further values accepted at ch1
}

func main() {
    ch1, ch2 := make(chan int), make(chan int)
    go prod(ch1)
    go prod(ch2)
    cons(ch1, ch2)
}

Figure 1: Partially deadlocked Producer-Consumer in Go

Figure 2: The Godel Checker [31] workflow.

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 1-4) 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 5-11) 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 6) containing a selective communication construct (line 7). 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 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 12-16) sets up the producers and consumer by creating the two synchronous channels ch1 and ch2 (line 13), 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.

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 1-4) 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 5-11) 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 6) containing a selective communication construct (line 7). 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 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 12-16) sets up the producers and consumer by creating the two synchronous channels ch1 and ch2 (line 13), 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 liveliness. 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 inferred 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.

\[ \alpha \vdash \bar{u} | u | \tau \]

\[ T \vdash \{ t_i(\bar{u}_i) = T_i \}_i \in S \]

\[ T, S \vdash \alpha; T | T; S | \&\{ \alpha; T_i \}_i \in \{ T | S \} | 0 \]

Communication is specified with the \( \& \) prefix, where \( \alpha \) can be \( \bar{u} \), denoting a send on channel \( u \), \( u \), denoting the dual receive action on channel \( u \) and \( \tau \), 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 \( \& \{ \} \) signals that channel \( u \) is to be closed. We represent conditional branching with the \( T @ S \) 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 \( \&\{ \alpha; T_i \}_i \in \{ T | S \} \). The construct \( T @ S \) for the availability of one of the \( \alpha_i \) communication actions. When some action \( \alpha_j \) 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 \( \tau \) actions are always available to fire, we can use a \( \tau \) action in a select construct to model timeouts or default behaviours when no other actions are available. The parallel composition construct \( T @ S \) denotes the parallel execution of \( T \) and \( S \); the construct \( 0 \) denotes no behaviour.

Channels in Go are synchronous buffers by default, but may also be created with a \( \text{bind} \), 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. (new \( \alpha \); \( T \) denotes the creation of a channel \( \alpha \) (with a bound \( n \)) which can be used locally in \( T \). If the bound \( n \) is set to 0, then \( \alpha \) is a synchronous channel. We often write (new \( \alpha \); \( T \) for (new \( \alpha \); \( T \) and assume that the scope of \( \alpha \) extends as far to the right as possible.

Construct \( t(\bar{u}) \) denotes a function call with parameter \( \bar{u} \). We often identify a list \( \bar{u} \) with its underlying set and write \( x \in \bar{u} \) if \( x \) is a element of \( \bar{u} \). We use the following two runtime constructs to define the semantics of types in §4: \( [a]_n^k \) 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 \( a^* \) represents a closed channel \( a \).

The type of a program, sometimes written \( \{ t_i(\bar{u}_i) = T_i \}_i \in S \), 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 \( t_i(\bar{u}_i) = T_i \) we assume that the free names of \( T_i \) are included in \( \bar{u}_i \).

Example 2.1 (Type for Consumer-Producer and their Properties).

The behavioural type for the program of Figure 1 is given below.

\[
\{ \text{prod}(ch) = \text{ch} : \text{prod}(ch) @ \text{close} ch \\
\text{cons}(ch_1, ch_2) = \&\{ \text{cons}(ch_1, ch_2), ch_2 : \text{cons}(ch_1, ch_2) \}
\}

\text{main}() = (\text{new} ch_1, ch_2) ; (\text{prod}(ch_1) | \text{prod}(ch_2) | \text{cons}(ch_1, ch_1))
\]

Definition \( \text{prod}(ch) \) specifies the type for the \text{Producer} function, while \( \text{cons}(ch_1, ch_2) \) stands for the type of the Consumer function, and \( \text{main}() \) 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 \( \text{prod}(ch) \) specifies the behaviour of performing an internal choice (denoted by the \( @ \) 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 \( \text{main}() \) in Example 2.1 satisfies GDF since the communication actions in subcomponents \( \text{prod}(ch_1) \) and \( \text{cons}(ch_1, ch_1) \) can always make progress, despite the fact that actions in \( \text{prod}(ch_2) \) 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 \( \text{cons}(ch_1, ch_1) \) with \( \text{cons}(ch_1, ch_2) \) makes the type \( \text{main}() \) 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 \( \text{select} \) construct), only branches that can be selected are required to eventually succeed. For instance, the following single-producer variant of \( \text{main}() \) also satisfies liveness even though the \( ch_2 \) branch in the select construct can never be taken: \( \text{new} ch_1, ch_2) ; (\text{prod}(ch_1) | \text{cons}(ch_1, ch_2)) \).

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

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
func main() {
    ch := make(chan int) // Create channel
    go sendFn(ch)       // Run as goroutine
    x := recvVal(ch)    // Ordinary func call
    for i := 0; i < x; i++ { print(i) }
    close(ch)           // Close channel ch
    func sendFn(c chan int) { c <- n }
    func recvVal(c chan int) int {
        return <-c
    }
    func main() { i := 0; i < x; i++ 
        if n%2 == 0 { c <- n }
    }
    t0 = make(chan int)           // Receive from channel c
    if t0 < -c return
    goto if 1 return
    if pred block is Blk_i
}

Listing 1: A simple concurrent program in Go.

The type for the function send (lines 1-4) is send(c) = T; send(c) @ send(c). Similarly, the type for recv (line 5) is recv(c) = c; recv(c).

The type for the program above is deemed live since the 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.

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 programmatically. 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 <- ch respectively (lines 7 and 9), spawning of goroutines (go sendFn() in line 3), and closing channels, i.e. close operation (line 6), 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 // case index` 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 `phi [ B1k1,...,vI ] i ∈ 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 `v3` (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 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 `fun(x)`, we generate a type signature of the form `fun_n(y, i, b)` where:

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

**Listing 2: A select statement in Go.**

![Figure 4: Simplified SSA IR built from Listing 2.](image)

\( y \) is a subsequence of \( x \) where each \( y \) in \( y \) is a channel parameter \( i \) 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 \( \Delta \) and write \( \Delta(fun, n) \) for the signature of block \( n \) in function `fun`.

### 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`; \( \rho \), the type we have constructed thus far; and \( \Gamma \), a context which maps each channel variable name to its `representative`.

The context \( \Gamma \) 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 \( \Gamma[t \mapsto t'] \) for the context \( \Gamma \) where the mapping from \( t \) is updated to \( t' \). We assume that \( \Gamma[t \mapsto t'] \) is `undefined` if \( t \in \text{dom}(\Gamma) \) and \( \Gamma(t) \neq \bot \) in order to `disallow` channel overwriting.

**Algorithm for genFunction.** Our algorithm relies on auxiliary (partial) functions for the translations from statements to types:
function genFunction(fun, n, k, ρ, Γ)
switch s ← statement at line k do
  case t = make chan T 5 do
    genFunction(fun, n, k+1, ρ; newT, Γ[t → t])
  case t = local chan T do
    genFunction(fun, n, k+1, ρ, Γ[t → ⊥])
  case t = v or t = ⊥ or t = s = t do
    genFunction(fun, n, k+1, ρ; mkPrefix(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; mkJump1(fun, i)
  case if _ goto i else j do
    return ρ; (mkJump1(fun, i) @ mkJump1(fun, j))
  case select b [ y1, ..., yj ] do
    ρc ← mkJump1(fun, n+1)
    for i in [ 1, ..., j ] do
      ρi ← mkPrefixx(yi)
      ρj ← mkJump1(fun, n+2+i)
      if b = nonblocking then
        ρd ← mkJump1(fun, n+1+2f)
        return &{ (ρi; ρj; ρc) ∈ [1, ...] } ∪ { (ρd; ρc) }
      else return { (ρi; ρj; ρc) ∈ [1, ...] } 
  case F(x) or t = F(x) do
    if t is a channel then abort
    else genFunction(fun, n, k+1, ρ; mkCall1(F, x), Γ)
  case go F(x) do
    return ρ; (mkCall1(F, x) | ρ′)
  case t = 0 or t = 1 do
    if t is a channel then
      genFunction(fun, n, k+1, ρ; Γ[t0 = Γ(t1)])
    else genFunction(fun, n, k+1, ρ, Γ)
  case phi [ Bi:vi | e1:en ] do
    if ∃i ∈ InEdges: vi 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

mkPrefix1(s) sends/receives actions and selects guards
mkJump1(fun, j) jumps statements
mkCall1(fun, 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 Γ[t → 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 Γ[t → ⊥]. 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 mkPrefix1(s), defined below:

\[
\text{mkPrefix1}(s) = \begin{cases} 
\text{false} & \text{if } t = \varnothing \text{ and } \Gamma(t) = u \\
\text{true} & \text{if } s \in \{\varnothing, \text{send}, \text{recv} \} \text{ and } \Gamma(t) = u \\
\text{false} & \text{if } t = \varnothing \text{ and } t \text{ is a timeout channel}
\end{cases}
\]

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 function call through the auxiliary function defined below, which uses the globally available signature environment (Λ).

\[
\text{mkJump1}(fun, j) = \begin{cases} 
\text{fun}(y, \Gamma(i), \tilde{v}) & \text{if } \Lambda(\text{fun}, j) = \text{fun}(y, \tilde{i}, \tilde{v}) \\
\bot & \text{and } \forall x : \Gamma(x) \neq \bot \\
\text{false} & \text{otherwise}
\end{cases}
\]

\[
\text{mkJump1}(fun, j) \models \text{a jump to another block within the same enclosing function hence there is no need to rename the parameters nor the "global" variables from the function signature (since they are fixed within the scope of the function). Instead, the internally declared variables are replaced by their representatives using } \Gamma.
\]

The function mkJump1(fun, j) is undefined if any of the } i arguments maps to an uninitialised channel in order to guarantee that these cannot be overwritten in the definition of fun. The } y and } v arguments are assumed to be initialised by the parent function.

Conditional constructs are also translated straightforwardly using @ 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 continuation of the select statement is stored in } p0, while the guard and body of each case is stored into } p1 and } p0, 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 mkCall1(fun, x), which defined as follows:

\[
\text{mkCall1}(fun, x) = \begin{cases} 
\text{fun}(\Gamma(x), \tilde{v}) & \text{if } \Lambda(\text{fun}, 0) = \text{fun}(y, \tilde{v}) \\
\bot & \text{and } \forall x : \Gamma(x) \neq \bot \\
\text{false} & \text{otherwise}
\end{cases}
\]

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.
function genBody(fun, n)
    funₙ(y, i, o) ← Δ(fun, n)
    Γ ← [x ← x]ₙ∈Φ,i,o
    return genFunction(fun, n, 0, Γ)

goroutines. If s spawns a new goroutine, we proceed similarly to
the function call case but place the call in parallel with its continu-
ation or concurrency.

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

```
main₀() = (new t₀); (sendF₀(t₀); recvVal₀(t₀); main₃(t₀))
main₁(t₀) = main₀(t₀)
main₂(t₀) = c₁(t₀); 0
main₃(t₀) = main₁(t₀) & main₂(t₀)
sendF₀(c) = t; 0
recvVal₀(c) = c; 0
```

Note that the spawning of the goroutine sendF₀(ch) becomes a
parallel composition of the main thread with sendF₀₀(t₀).

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

```
myselect₀(c) = &{c; myselect₂(c); myselect₁(c),
τ:myselect₄(c); myselect₁(c),
τ:myselect₃(c); myselect₁(c)}
myselect₁(c) = 0  for i ∈ [1, 2, 4, 5]
myselect₂(c) = myselect₄(c) & myselect₅(c)
```

The type’s entry point is myselect₀(c) and myselect₃(t₀) is un-
used. 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 myselect₄(t₀).

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 expre-
sed in the modal µ-calculus [28].

The notion of finite control has several definitions in the litera-
ture [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
t(x) = t(x) | T or t(x) = (new a); t(a) are not finite control as the
former generates infinitely many instances of type t(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 con-
structs that are specific to the Go programming language. The
labels, ranged over by α and β, have the form:

```
α, β ::= a | a | τ | rₐ | cloa | cloa | a* | a*
```

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 equiva-
ences defined in Figure 6. Thus, a program’s type is always of the
form:

```
{t₁(y₁) ∈ T₁; i ∈ T₀(∀a₀…∀aₖ; t₀)}``
0; T ≡ T | S ≡ S | T | (T′ | S) ≡ (T | T′) | S
T | 0 ≡ T ≡′ 0 ≡′ T ≡′ T′
(newα a); (new′ m b); T ≡ (new′ m b); (newα a); T
(newα a); 0 ≡ 0 (newα a); a∗ ≡ 0 (newα a); [a]Γ ≡ 0
T | (newα a)S ≡ (newα a)(T | S) (a ∉ fn(T))

Figure 6: Structural congruence for types.

where the several Ti contain no channel creations. We also make use of the following transition which initialises all the channels accordingly and write A for the set of all initialised channels:

\{(t;ij) = Ti}_{i∈I} \land \text{(new}α i \text{ an)} \ldots \text{(new}k \text{ an}) ; t0() \}
\xrightarrow{\tau} \{(t;ij) = Ti}_{i∈I} \land \text{(t0() | a0} | \ldots | [a]i_{k} \}

We give the semantic rules for behavioural types in Figure 5, adapted from [30], where T \xrightarrow{\tau} T′ denotes that T reduces to T′ by producing a, 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 type nform in normal form, we construct a finite labelled transition system which represents all possible executions of t0(), i.e., the entry point type under all the name restrictions. The LTS of t0() is a tuple T = (S, t0(), →, A) such that S is a set of states implicitly labelled by behavioural type terms (we often identify labels and states), t0() ∈ S is the initial state, A ⊆ {τ} ∪ {τa | a ∈ A} is the set of labels, and → ⊆ S × A × S is the transition relation T \xrightarrow{\tau} T′ where the label α can be either a silent move, i.e., τ, or a synchronisation over a channel, e.g., τa.

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 ↓\l o or T ↓\l o which holds if T is ready to fire action a or one of the actions in \tilde{a}, with a, α ∈ {a, \overline{a}, \taua, c\alpha a, a\alpha a, a\alpha a}. Table 2 explains the meaning of each label and Figure 7 gives the defining rules of the predicates T ↓\l o and T ↓\l o. Essentially, T ↓\l o if T is immediately ready to fire action a (with a ≠ τ) and T ↓\l o if T contains an external choice which does not feature a branch guarded by τ (i.e., τ ∉ \tilde{a}). We have, e.g., ¬(τ; T ↓\l o) for any a and ¬(a; T ↓\l o) for any \tilde{a}, which is an important subtlety for defining accurate safety and liveness properties.

Table 2: (Predicate) labels

| a; T ↓\l o | close a; T ↓\l o | ∀ i ∈ \{1, ..., n\} : a_i ↓\l o |
| a; T ↓\l o | a* ↓\l o | \& (a_i; T ↓\l o) ↓\l o |
| T ↓\l o | T ↓\l o | T ↓\l o | T ↓\l o |
| T ↓\l o | T ↓\l o | T ↓\l o | T ↓\l o |
| T ↓\l o | T ↓\l o | a ∉ fn(α) | T ↓\l o |
| T ↓\l o | T ↓\l o | (newα a); T ↓\l o | T ↓\l o |

Figure 7: Barb predicates for types.

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 \phi is interpreted on a pointed LTS, i.e., an LTS with a starting state T, we write T \models -\phi if \phi satisfies \phi in the LTS T. Namely, formula T holds for every T (while \models never holds). The construct [a]φ is a modal operator that is satisfied if, for each a-derivative T′ of T (i.e. T′ is reachable from T by performing action a), the formula \phi holds in T′. The dual modality is (a)φ which holds if there is an a-derivative T′ of T such that \phi holds in T′. Construct vx. φ (resp. µx. φ) is the standard greatest (resp. smallest) fixpoint operator (binding x in φ). The atomic proposition \l o (resp. \l o) holds iff T ↓\l o (resp. T ↑\l o). Given a set of actions A ⊆ A, we write [A]φ for \Lambda_{a∈A}[a]φ and (A)φ for \Lambda_{a∈A}(a)φ.

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 \phi, formula Ψ(\phi) holds if \phi holds for all reachable states, while formula Φ(\phi) holds if \phi holds in some reachable state. Formula Ψ(\phi) holds if no terminal state is reachable in T, i.e., the behavioural types only exhibit infinite executions; dually formula Ψ(\phi) holds if there are no cycles in T; this property is useful as it implies that liveness of types corresponds with liveness of programs (cf. §2.2). Formula Ψ(\phi) models the global deadlock-freedom property discussed in §2.1, i.e., the formula holds if for each state T in T if T is ready to execute a send or receive action, then T has a successor. Formula Ψ(Ψ and Ψ) models the liveness property (cf. §2.1). It holds if
for all state $T$ in $T$ (i) if $T$ is ready to send/receive on $a$, there is always eventually a synchronisation on $a$ (cf. $\psi_k$), and (ii) if $T$ has a select construct, which does not contain a $\tau$-branch, there is always eventually a synchronisation over one of the channels guarding the construct (cf. $\psi_k$). Formula $\Psi(\psi_s)$ models the channel safety property, i.e., no send nor close action is executed on a channel [Always].

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,

```go
func f(n int) { for i := 0; i < n; i++ { for j := 0; j < 10; j++ { ... } }}
```

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 uses 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 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 ($\psi_g$), liveness ($\psi_l$), channel safety ($\psi_c$) and eventual reception ($\psi_e$). A ✓ mark means that the considered tool reports that the property holds. For example, in the case of GoInfer Checker, a ✓ mark under column $\psi_g$ 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 GoInfer Checker, where Column 10 lists the inference time, Columns 11 and 12 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 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 hwcat 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 deadlock-free 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 GoInfer 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 GoInfer 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), Gode1 Checker is a more general tool since it can verify arbitrary properties expressible in the $\mu$-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 inite 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 targeting 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 liveliness 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 liveliness 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 liveliness 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 $\mu$-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 $\pi$-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 Gode1 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