

Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis

Nicholas Ng

Imperial College London, UK
nickng@imperial.ac.uk

Nobuko Yoshida

Imperial College London, UK
n.yoshida@imperial.ac.uk

Abstract

Go is a programming language developed at Google, with channel-based concurrent features based on CSP. Go can detect global communication deadlocks at runtime when all threads of execution are blocked, but deadlocks in other paths of execution could be undetected. We present a new static analyser for concurrent Go code to find potential communication errors such as communication mismatch and deadlocks at compile time. Our tool extracts the communication operations as session types, which are then converted into Communicating Finite State Machines (CFSMs). Finally, we apply a recent theoretical result on choreography synthesis to generate a global graph representing the overall communication pattern of a concurrent program. If the synthesis is successful, then the program is free from communication errors.

We have implemented the technique in a tool, and applied it to analyse common Go concurrency patterns and an open source application with over 700 lines of code.

Categories and Subject Descriptors D.1.3 [Programming Techniques]: Concurrent Programming; D.2.4 [Software Engineering]: Software/Program Verification

Keywords Concurrent Go, Session types, Synthesis, Deadlock freedom, Type safety, Communication safety, Static Analysis.

1. Introduction

Concurrent programming is made difficult by the subtleties required for correct access to shared memory. The research community has dedicated years of efforts to understand concurrent behaviours with channel-based process calculi such as Hoare's CSP [11, 12], Milner's CCS [26] and π -calculus [27]. However, most mainstream programming languages do not adopt these channel-based models but instead support concurrency through thread programming or external libraries such as OpenMP [5], or abstractions with complex semantics [2].

The Go programming language [32] from Google takes a different approach and supports concurrency as a fundamental part of the language, with a channel-based concurrency model inspired by CSP. As a result of the design, Go offers a high-level, compositional way of constructing concurrent applications. In Go, values are shared by communicating over channels between lightweight

threads called goroutines. With this principle, the Go concurrency makes it easy to construct streaming data pipelines that make efficient use of multiple CPUs. The recently introduced `core.async` in the Clojure language also adopts a similar model for concurrent programming.

In this work we analyse concurrency in Go programs leveraging *session types* [13, 14, 35], which is a formal typing discipline for communication that has its roots in the π -calculus. Session types consist of type structures for communications, such as a series of sending, receiving, choices and recursions, promoting structured communications programming. Further, the theory of multiparty session types [14] guarantees of deadlock-freedom and communication safety between multiple participants, offering a choreographic view of communications in concurrent programs. We demonstrate how recent advanced techniques from the multiparty session type theory (*synthesis* of global (session) graphs in [23]) can be applied to detect and preempt concurrency problems present in Go applications.

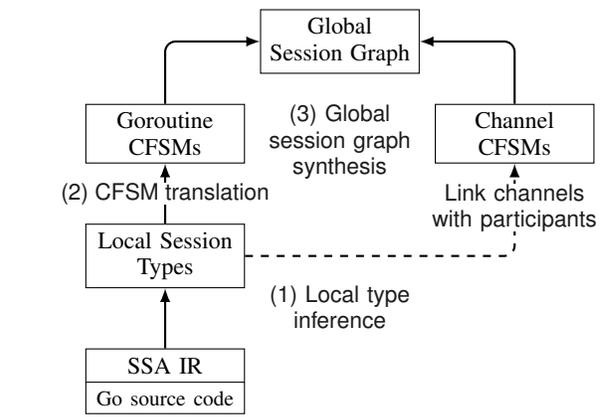


Figure 1. Overall workflow of the tool chain.

Contributions This paper proposes the first static deadlock detection tool, *dingo-hunter* based on session types, for the Go language. The main techniques we use are *inferring* communicating finite state machines (CFSMs) [3] by extracting communications in a given Go program; and *synthesising* global session graphs which represent the overall structure of communications from CFSMs.

Figure 1 outlines the workflow of our tool chain.

§ 2 gives an overview of concurrency in the Go language and the limitations of the existing Go deadlock detector.

§ 3 shows an automatic inference of local session types from the Go language. This is divided into two steps. First we translate

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481.

CC'16, March 17-18, 2016, Barcelona, Spain
Copyright © 2016 ACM 978-1-4503-4241-4/16/03...\$15.00
DOI: <http://dx.doi.org/10.1145/2892208.2892232>

Reprinted from CC'16, [Unknown Proceedings], March 17-18, 2016, Barcelona, Spain, pp. 1–11.

a Go program to the Single Static Assignment (SSA) intermediate representation [10]. Then we translate it into a set of local session types.

§ 4 first shows how to convert local session types into two kinds of CFSMs, Channel CFSMs (which model channels between participants) and Goroutine CFSMs (which model communication behaviour of goroutines). Then it explains how to synthesise global session graphs from these CFSMs.

§ 5 examines case studies of common Go concurrency patterns to show how they can be analysed by our tool; and

§ 6 evaluates the tool on existing programs to demonstrate the robustness of our approach.

The source code for the analysis tool and supplementary materials are available from [1]. The synthesis tool of [23] used in the work is available from [18].

2. The Go Programming Language

Go [32] is a statically typed, compiled systems programming language in the tradition of C and C++. One of the more well-known features of Go is the built-in support for channel-based concurrency, where channels are first-class objects and integrated in the core language design.

In the following example, Go program calculates the sum of two numbers.

```
1 package main
2 import "fmt"
3
4 func add(x, y int) int { return x + y }
5
6 func main() {
7     sum := add(1, 2)
8     fmt.Printf("Result: %d\n", sum)
9 }
```

Listing 1. Example Go program.

The entry point of the executable code is the `main` function in line 6. In the body of the `main` function, the `add` function is called, which was defined in line 4. Notice that in line 7, the type of the `sum` variable is automatically inferred and the line is equivalent to `var sum int = add(1, 2)`.

2.1 Concurrency in Go

Go supports concurrency through a feature called *goroutines*. Goroutines are lightweight functions which *execute concurrently with other goroutines in the same address space*¹ and communicate through typed *channels*.

A function is executed as a goroutine if the function call is prepended by the keyword `go`, and it does not block the caller. Channels in Go are typed FIFO queues which can be shared between goroutines. They are synchronous by default, meaning that they block on both send and receive commands. Consider the following code which is modified from Listing 1 to be executed concurrently using goroutines and channels.

```
4 func add(x, y int, resultCh chan int) {
5     resultCh <- x + y
6 }
7
8 func main() {
9     result := make(chan int)
10    go add(1, 2, result) // Call goroutine, non blocking
11    fmt.Printf("Result: %d\n", <-result) // Receive
12 }
```

Listing 2. Example Go program with goroutines and channels.

¹From the Go language specification

Listing 2 shows a modified Listing 1 to use goroutines. Since the `add` function is executed as a goroutine and the return value will be inaccessible, we instead pass an integer channel (i.e. variable of type `chan int`) as a parameter when calling `add`. The result of the calculation can be then *sent* to the channel using the channel send operator `<-` on line 5.

The integer channel is created in the `main` function (i.e. `go add` callee) using the built-in `make` allocation function on line 9. After `main` starts the goroutine on line 10, `main` blocks and waits to receive using the receive operator `<-` on the channel `result`. The difference between the send and receive operators is the position of the channel variable (the arrow points into the channel for send and out of the channel for receive).

Goroutines and channels are the primary means of concurrency in Go (with the exception of the low-level synchronisation, `sync`, package in the standard library).

2.2 Deadlock Detection in Go

Communication deadlocks are potential pitfalls of concurrency in Go. The Go runtime comes with a runtime deadlock detector. The tool detects a deadlock if all of the running (non-idle and unlocked) goroutines are blocked waiting.

The deadlock detector is sound (i.e. does not give false positives), but the tool has certain limitations:

- Only global deadlocks are detected, which means that if not all goroutines are involved in the deadlock, the tool will not detect it is a deadlock. For example in Listing 3, `Work` is a long running goroutine that performs calculations unrelated to communication, keeping the number of active goroutines non-zero. This hides the fact that one of the `Recv` goroutines are blocked waiting forever since there is no sender.
- Since the runtime detector only detects deadlocks which manifest themselves at runtime, deadlocks in long running processes may go undetected until a specific execution path is triggered.

Listing 3 shows a Go program with a communication deadlock which the Go deadlock detector cannot detect. There are four goroutines in the program: `Send` sends a message, `Recv` receives a message and signals completion, and an extra `Recv` which introduces a communication mismatch on channel `ch`. `Work` is a non-ending goroutine which performs computation unrelated to other goroutines.

```
1 func Send(ch chan<- int) { ch <- 42 }
2 func Recv(ch <-chan int, done chan<- int) {
3     val := <-ch
4     done <- val
5 }
6
7 func Work() {
8     for {
9         fmt.Println("Working..")
10        time.Sleep(1 * time.Second)
11    }
12 }
13
14 func main() { // main
15     ch, done := make(chan int), make(chan int)
16
17     go Send(ch) // Send_280
18     go Recv(ch, done) // Recv_293
19     go Recv(ch, done) // Recv_312
20     go Work() // Work_331
21
22     <-done // First receive
23     <-done // Second receive
24 }
```

Listing 3. A Go program with an undetected deadlock.

The runtime detector works by counting the number of running goroutines during execution, and detects a deadlock if the number is 0. In the example, `Work` is always running, and despite that one of the `Recv` blocks forever, it does not deadlock in the global level, hence the running goroutines count is at least one and no deadlocks are detected. In our approach, we perform static analysis on the source code. Our deadlock detection is syntactic and does not depend on runtime properties of the program. We can identify a communication mismatch as we analyse the code, and find one send for `ch` and two receives.

3. Inferring Local Session Types from Go code

This section describes how our static tool analyses Go source code to derive a set of local session types by abstracting concurrent interaction patterns amongst the goroutines.

3.1 Local Session Types

We begin by extracting the communication operations in the source code as a local (session) type – a control-flow graph with session primitives. A program is a closed communicating system which we call a session. Each local type represents a single *participant* of the session, hence corresponds to a single instance of a goroutine. In a Go program, `main` is an implicit goroutine, and every instance of a goroutine spawned by `go functionName()` is a separate participant even for the same function.

The local session type, or simply local type, is a control-flow graph where each node is one of following: (1) *Channel* `ch T` (create a new channel with name `ch` of type `T`), (2) *Send* `ch` (send to a channel `ch`), (3) *Recv* `ch` (receive from a channel `ch`), (4) *Close* `ch` (close or terminate a channel `ch`), and (5) *Label* (a named jump label). (2)–(4) are communication operations, and (2) and (3) correspond to send and receive type from [14], written $ch! \langle T \rangle ; T'$ and $ch? \langle T \rangle ; T'$ respectively where T' is the continuation (child node), and (4) correspond to the closing channel operation in [9].

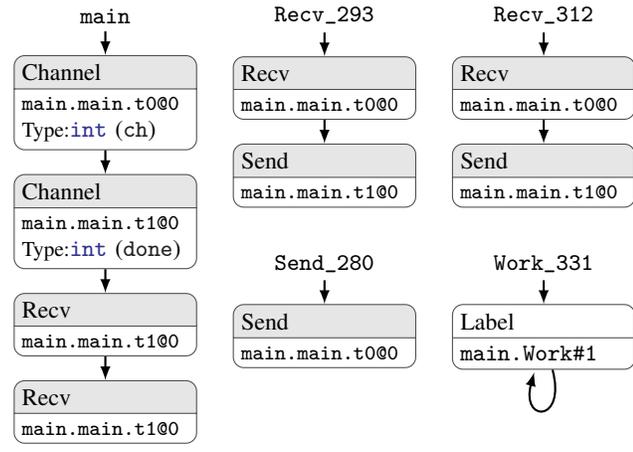


Figure 2. Local types of Listing 3.

Figure 2 lists the set of local types extracted from the source code of Listing 3. The figure lists five subgraphs, where each of them represents one local type for one goroutine.

The names `Send_280`, `Recv_293`, `Recv_312` and `main` are identifiers for the instances of goroutines, and are the initial nodes of the local graphs. The channels `ch` and `done`, created in the `main` function, are represented in the `main` graph as unique names `main.main.t0@0` and `main.main.t1@0` respectively (line 14), followed by two receives (lines 21 and 22). The unique names are used for referencing the channel in `Send`, `Recv` and `Close` nodes.

The next three local types correspond to `go Send`, the first `go Recv` and the second `go Recv` in lines 16–18. `Send_280` calls the function on line 1 so that it has `Send` node to `main.main.t0@0` (`ch`). Similarly both `Recv_293` and `Recv_312` are followed by two nodes, `Recv main.main.t1` and `Send main.main.t0`, which correspond to the function in line 2–5.

In the local type for `Work` (i.e. `Work_331`), there are no communication operations and hence describes only the control-flow of the `Work` function, which includes a loop. The resulting local type contains communication primitives that interact between participants through channels. A channel is a dynamic medium between a sender and a receiver, connecting the participants.

3.2 Extraction with SSA IR

We implement our analysis using the Single Static Assignment (SSA) intermediate representation of the Go program, constructed by the `go/ssa` package in the standard library. The SSA representation simplifies the syntax of a Go program into a limited set of instructions, and flattens the control flow of the program as jumps between blocks of instructions for analysis.

We explain below how a local type can be inferred from Go code. The analysis starts from the program’s main entry point, the `main` function, and interprets the SSA instructions following the program control-flow. Instructions related to communications are converted to nodes of the local type graph, and instructions related to control-flow are converted to edges of the local type graph. At the end of the analysis, a set of local type graphs is generated to represent the set of goroutines in the program. We first look at communication related instructions.

Communication Instructions (Table 1) These are the most important instructions in our analysis, because they define the communication behaviour of the goroutines. When encountering the following instructions, new local type nodes are created.

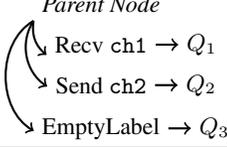
`MakeChan{Size}` is an instruction to initialise a new shared channel for communications. A typed channel (i.e. `chan T`) is a reference type and can only be initialised by a `make(chan T)` in the source code. Since channels are first class objects in Go, the initialisation is captured by this special instruction. Once created, channels do not change (except for `close`). In this work we only consider unbuffered (i.e. `Size = 0`), hence synchronous, channels. In the local type, we assign each channel created with `MakeChan` with a unique name in the form of `package.function.register@version`. The part before `@` is a notation for fully qualified variable name, and the part after is a counter for distinguishing between channels created by multiple calls to the function. This unique name is used as a reference for other communication instructions since channels are shared between goroutines.

`Send{Chan=ch, X}` is an instruction representing sending of a value (`X`) to a channel (`Chan`) variable. It corresponds to a send node in the local type (`Send ch`).

`UnOp{Op=ARROW, X=ch}` is an instruction that corresponds to receive. The receive operator (`<-ch`) is implemented in Go as a unary expression on the channel (`X`), and as a result, the received values are not required to be stored. It corresponds to a receive node in the local type (`Receive ch`).

`Select{States, Blocking}` represents a choice based on the communication on channels. Each of the cases in the `select` is guarded by a communication operation (either send or receive), and the case is chosen if the operation does not block. `States` is a list containing details of the guard: the channel, the type of operation (send or receive) and the value to send (if it is a send operation). The other argument, `Blocking`, is a boolean which indicates if a `default` case exists. If a default case exists, then when all of the communication cases are blocking, the default case is chosen (i.e. `Blocking = false`). In our conversion to local types, the local types of each case in the

Table 1. Communication related instructions for local type nodes.

SSA instruction		Local type nodes	Example Go code
<code>MakeChan{Size}</code>	Create and initialise a channel with <code>Size</code> buffer	Channel <code>ch</code> <i>Type</i>	<code>ch := make(chan <i>Type</i>)</code>
<code>Send{Chan, X}</code>	Send a value (<code>X</code>) to a channel <code>Chan</code>	Send <code>ch</code>	<code>ch <- val</code>
<code>UnOp{Op=ARROW, X}</code>	Unary <code><-</code> , i.e. Receive from a channel <code>X</code>	Recv <code>ch</code>	<code>var := <-ch</code>
<code>Select{States}</code>	Non-deterministic choice on channel		<pre>select { case <-ch1: Q1 case ch2 <- val: Q2 default: Q3 }</pre>
<code>Builtin{Name=close}</code>	Builtin function to close a channel	Close <code>ch</code>	<code>close(ch)</code>

`select` is appended as a child of the parent local graph node of `Select` and the `default` case is translated to an empty label to denote no communication operations.

`Builtin{Name=close, Arg=[ch]}` represents the use of a built-in functions defined by the language. Other notable builtin functions are `len()` (length or size of array/slice/struct). The `close` builtin function takes a channel as an argument and closes the channel. Once a channel is closed, it cannot be used by any goroutines anymore. The local type of `close` is a close node: `Close ch`.

Control-flow Instructions (Table 2) These instructions define the control flow of a Go program. As a Go program is translated into SSA IR, the function bodies are segmented into *blocks* of related instructions, for example, Listing 5 shows a for-loop split into four blocks, one for loop entry and initialisation (Block 0), one for loop body (Block 1), one for exit block with continuation after for-loop (Block 2) and one for calculating and checking loop conditions (Block 3). The right hand column are the types of the registers (e.g. `t0` is an `int`). These blocks are connected by conditional and unconditional jumps at the end of each block. The jumps between blocks follows the flow of control of the program.

```
1 sum := 0
2 for i := 0; i < 10; i++ {
3     sum += i
4 }
5 fmt.Println(sum)
```

Listing 4. For-loop in Go.

```
0:                                     entry P:0 S:1
  jump 3
1:                                     for.body P:1 S:1
  t0 = t7 + t8                          int
  t1 = t8 + 1:int                        int
  jump 3
2:                                     for.done P:1 S:0
  t2 = new [1]interface{} (varargs)      *[1] interface{}
  t3 = &t2[0:int]                         *interface{}
  t4 = make interface{} <- int (t7)      interface{}
  *t3 = t4
  t5 = slice t2[:]                        [] interface{}
  t6 = fmt.Println(t5...)                 (n int, err error)
  return
3:                                     for.loop P:2 S:2
  t7 = phi [0: 0:int, 1: t0] #sum         int
  t8 = phi [0: 0:int, 1: t1] #i           int
  t9 = t8 < 10:int                       bool
  if t9 goto 1 else 2
```

Listing 5. For-loop from Listing 4 in SSA IR form.

`Call{Func, Method, Args}` is an instruction to call functions. The function (`Func`) may be either an ordinary function, a closure or a `Builtin` function such as `close`. The control flow of the program follows the function and returns to the caller when the end of

the function body is reached. In the local types graph, an edge is added from the caller to the subgraph representing the local types of the callee body. During function calls, parameters of the callee are translated from arguments from the caller (`Args`), so variables used in the body of the function are aliases of the variables in the caller. This is particularly important to retrieve unique references to channels created by `MakeChan` instructions in callers.

`Go{Func, Method, Args}` is an instruction to spawn a goroutine. The instruction is similar to `Call`, but the function called will be started as a separate goroutine from the caller, thus there is no edge connecting the caller and the callee. A new local type graph is added to the session, and its initial node points to the subgraph representing the local types of the callee body.

`Jump` is an unconditional jump from the current block to a successor block. The `Jump` instruction only appears at the end of a block, and is a sequential transition from the current block to the next block in the same scope (e.g. within a function body). In the local types graph, this is a simply an edge from the local types of the current block to the local types of the next block.

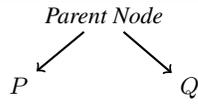
`Return` is an instruction to return from the callee to the caller function. When the control-flow reaches the end of a function, the control continues at the caller after the `Call` instruction that entered this function. In the local type graph, an edge is added to join the last node of the function to the continuation at the caller. For the analysis, this is the dual of the `Call` instructions, and the return values are copied back to the caller.

`If{Cond}` is a conditional jump instruction. `Cond` is the boolean jump condition which may be the result of an evaluation in the preceding instructions, with exactly two outcomes. If `Cond` is true, the control-flow continues at the direct successor block (i.e. same behaviour as `Jump`), otherwise it continues in the successor+1 block. Like `Jump`, the `If` instruction only appears in the end of a block. The `If` instruction is translated into the local type graph as two new edges, which point to the local types for then-block and the else-block respectively as child subgraphs.

`Defer{Func, Method, Args}` and `RunDefers` are a pair of instructions that defer a call to a function to the end of the current function. When a `Defer` instruction is encountered, the callee `Func` is pushed on a stack of deferred function calls. When a `RunDefers` instruction is encountered, the deferred function calls are then evaluated like normal `Calls`. It is guaranteed that `RunDefers` only appears once in each control-flow path per function. A new edge is added to the end of the local type graph of the caller function, and points to the local type subgraph of the deferred function.

Memory Access Instructions It is sufficient to use the two categories of instructions above, namely communication and control flow instructions, to infer local types from simple Go programs. The memory access instructions do not affect the control flow of

Table 2. Control-flow related instructions for local type edges; P , Q and F denote local type subgraphs for block P , Q and function F , respectively.

SSA instruction		Transitions	Go code
<code>Call{Func, Method, Args}</code>	Function (including closure) call	P_0 \downarrow F	<pre>func F(x chan T) { ... } func P() { ch := make(chan T) // Block P0 F(ch) }</pre>
<code>Go{Func, Method, Args}</code>	Start a function as a goroutine	P_0 new Q \downarrow \downarrow P_1 ...	<pre>func P() { // Block P0 go Q() // Block P1 }</pre>
<code>Jump</code>	Unconditional jump	P_0 \downarrow P_1	<pre>func P() { // Block P0 // Block P1 }</pre>
<code>Return</code>	Return from a function to caller	F \downarrow P_1 (caller)	<pre>func F() { return } func P() { F() // Block P1 }</pre>
<code>if{Cond}</code>	Diversion in the control flow		<pre>if e { P // Block 1 } else { Q // Block 2 }</pre>
<code>Defer{Func, Method, Args}</code> <code>RunDefers</code>	Defer execution to end of function Run deferred functions	P_0 \downarrow P_1 → Q	<pre>func P() { // Block P0 defer Q() // Block P1 }</pre>

the program, nor do they perform any communication with other goroutines. Their purpose is to make the channel variables available to the communication instructions later in the flow of the program when analysing the instructions. Hence no nodes or edges are added to the local type graph, and the memory access information is safely discarded after the analysis of the program is complete and local types are generated.

Go code that uses channel variables created by `make(chan T)` can be analysed if they are used directly with a `Send` or `Recv` instruction. However, programs often use data structures to organise data and pass around related groups of data. The example of a context structure in Listing 6 shows passing of a context variable `ctx` which contains a channel `ctx.done` to a function. In the actual execution and the generated SSA instructions, the `ctx` variable is first allocated with an `Alloc` instruction on line 4. Then the channel is created by a `MakeChan`, and stored in field of the allocated struct with a `Store` instruction. The `address` of the `ctx` struct is then passed to the function `f` on line 5. The function body of `f` uses the address of the `ctx` variable to locate a reference of the struct variable in `main`, and use the field access instruction `FieldAddr` to retrieve the `done` field of the struct, and use the channel for a `Send` on line 2.

Since the channels are not passed as variables directly, there is an indirection of memory access (via a structure and then its field). Our tool stores structure instances created in the source code and keeps track of where channels are stored in the allocated structures, then returns the correct channels when they are accessed.

```
1 type T struct { done chan struct{}; value int }
2 func f(ctx T) { ctx.done <- struct{}{} }
3 func main() {
4     ctx := T{ done: make(chan struct{}), value: 42 }
5     go f(ctx)
6     <-ctx.done
7 }
```

Listing 6. Context variables in Go.

Here we do not cover all memory access instructions and focus only on `struct`, as it is the most common data structure used for storing channels. The other data structures (arrays, slices) are similar but with a different set of instructions for allocation, access, and storage.

`Alloc{Heap}` is an instruction for allocating memory. We record the variable of the allocated memory if its type is a structure. The `Heap` parameter is a boolean indicating whether the memory is in heap or in the local activation (stack) frame.

`Field{X, Field}` and `FieldAddr{X, Field}` are instructions to access a field of a structure. The two instructions are for accessing the structure `X` as a value and as a pointer respectively. The field of a structure is indexed by a numeric field number `Field`. We lookup the allocation of structure `X`, and return the stored value in field `Field`. If the field is undefined, we keep a track of its parent structure and update the structure when a `Store` instruction is used on the field.

`Store{Addr, Val}` stores value (`Val`) in address (`Addr`). Changes to the `Val` are updated in `Addr` and will be reflected in the next

access to the Addr. If Addr is a field accessed by FieldAddr, the struct that holds the field will also be updated.

4. Global Graph Synthesis

This section explains how to convert a set of local types inferred from the Go source code into a global graph, which describes an overall concurrent interaction pattern of a Go program. We call this procedure *global graph synthesis*, and use the GMC-Synthesis tool introduced in a recent work [23].

The GMC-Synthesis tool takes Communicating Finite State Machines (CFSMs) as input and merges the CFSMs into a global graph of transitions. The first step of using the tool for global graph synthesis is therefore converting the local session graph obtained from Go source code into CFSMs.

4.1 From Inferred Local Types to CFSMs

The CFSMs are finite state machines where the transitions between states are labelled with either send or receive. The session types representation we use in the work can be characterised as a simple model of CFSM [7].

Each local session graph is translated to a single CFSM. The nodes in the local session graph are events in the CFSM model, and are represented in the CFSM model as transitions to new states. A label on transitions is decided by a type of a local type node: Send *ch* nodes become transitions with a send label using the notation *ch ! T* where *T* is the type of the channel *ch*; Recv *ch* nodes become transitions with a receive label with the notation *ch ? T*; Close *ch* nodes become transitions with a send label, but the message is STOP to indicate the termination of the channel, i.e. *ch ! STOP*.

In CFSMs, channels are fixed between two machines. A machine can use multiple channels, but the endpoints of the channels are always the same. However the Go channels are shared names which can be used by multiple goroutines. A Go channel is a shared location for two or more goroutines to synchronise on send and receive operations, thus the sender and receiver can be different at different points of an execution. An example is the done channel in the main goroutine from Listing 3 which is received twice, and is expected to receive from each of the two Recv goroutines once. This is reflected in the inferred local type in Figure 2, where the channel *main.main.t1@0* is used twice in the local type of main. The translation from local types into CFSM preserves this behaviour, where the CFSM communicates with channels which are variables and not endpoint machines. To illustrate the problem and our solution, Figure 3 shows the corresponding generated CFSMs. We also added the two channels *t0* and *t1* and annotated them with how messages are transferred from the senders to the receivers via the Go channels.

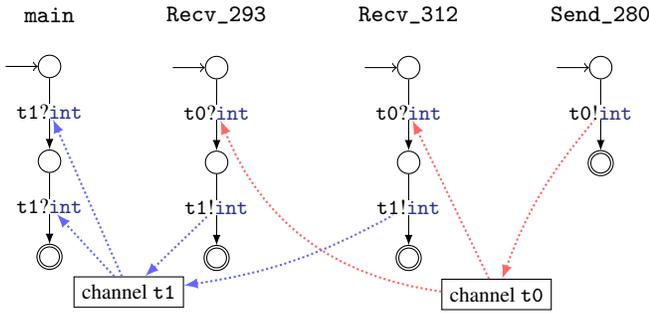


Figure 3. Goroutine CFSMs generated from Listing 2. It shows how channels connect to the CFSMs as proxies; dotted arrows represent channels used by Goroutine CFSMs.

From Figure 3, we can observe that the CFSMs do not communicate directly and Go channels resemble switches or multiplexors between the communicating machines, where the dotted arrows depict the direction of the message flow to and from the channels. There are no one-to-one connections between the CFSMs that use the same Go channels, but the link between the CFSMs and a Go channel is always one-to-one. So instead of treating channels as a static link between goroutines, we model the Go channels as CFSMs as well to represent all possible transitions that a Go channel are allowed such that they can connect multiple goroutines dynamically. To avoid confusion, we call these machines **Channel CFSMs** and distinguish Channel CFSMs from those which model the communication behaviour of goroutines. We call these CFSMs **Goroutine CFSMs**.

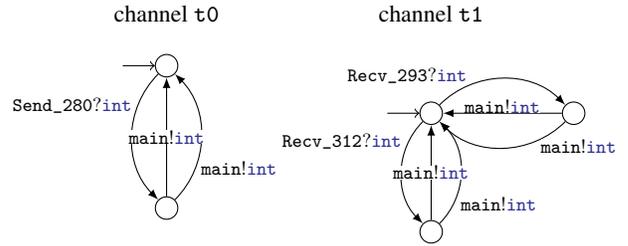


Figure 4. Channel CFSMs generated from Figure 2.

To generate Channel CFSMs from the local types, we first identify the channels. In Figure 2, the channels are *t0* and *t1*. We then construct a CFSM for each channel where the first transition from the initial state is a receive action at this channel and the second one is a send action which is matched to the initial action, returning to the initial state. Channel CFSMs of Figure 2 are given in Figure 4. For *t1*, there are two receive actions from Recv_293 and Recv_321, which are matched to two main actions. This corresponds to dotted blue lines in Figure 3, representing a synchronisation proxy (channel) between machines (participants).

4.2 Global Graph Synthesis

Given both Goroutine CFSMs which represent control flows of the inferred local types and Channel CFSMs synthetically generated from the local types, we can then synthesise a global graph from the CFSMs. The synthesis procedure involves generating all possible combinations of synchronous labelled transitions of the composed CFSMs. As explained in the previous subsection, CFSMs are FSMs with transitions labelled send or receive. Send and receive transitions in the same channel can be matched and synchronised as a single node in the global graph. Figure 5 shows the global graph of synthesised from the Goroutine CFSMs in Figure 3 and Channel CFSMs in Figure 4. In the global graph, rectangle nodes stand for interactions between two participants (CFSMs), where the number is the identifier of the machine. Diamond nodes stand for choices, and a branch is chosen from one of the outgoing edges.

Table 3. A possible execution of Listing 3; Leftmost column is the id of the CFSM.

#	Goroutine	(unmatched)
0	Chan t0	
1	Chan t1	
2	main	Recv t1; Recv t1
3	Send_280	Send t0
4	Recv_293	Recv t0; Send t1
5	Recv_312	Recv t0; Send t1

Table 3 shows a possible execution of the program in Listing 3. The execution can be mapped on to a trace in the global graph from

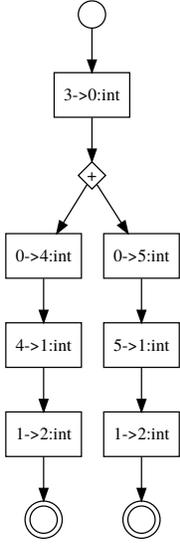


Figure 5. Global graph of Listing 3 with a deadlock.

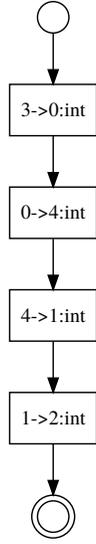


Figure 6. Global graph of the corrected example without deadlock (satisfies GMC).

initial state to one of the final states. From the column alignment, it is clear that one of the `Recv` goroutine is stuck and has a deadlock.

Generalised Multiparty Compatibility As shown in Figure 5, a successful construction of a global graph does not guarantee communication safety and deadlock freedom. We use a sound and complete condition for constructing global graph introduced in [23] called *Generalised Multiparty Compatibility* (GMC). Communicating systems that satisfy the GMC conditions are communication safe and deadlock free. GMC is based on two conditions: (i) representability, where each trace and choices in the CFSMs are represented in the global graph; and (ii) branching property, which states that whenever there is a choice in the global graph, a unique machine takes the decision and the decision is propagated to other machines. Condition (i) ensures that no information is lost in the construction of the global type, and condition (ii) ensures that all branches are well-formed.

In [23], the representability condition is applied on all CFSMs used to construct the global graph. In our approach we introduced the generic Channel CFSMs. Channel CFSMs are usually not representable because that would imply that all Channel CFSMs have synchronised with all pairs of Goroutine CFSMs once or more. As the Channel CFSMs are only generated to limit valid synchronisations and does not give any influence on the communication behaviour of Goroutine CFSMs, we only need to check the representability condition on Goroutine CFSMs.

Consider Figure 5 and the execution snapshot of Table 3. The GMC condition is not satisfied because, when the first non-deterministic choice is made, i.e. the choice between $0(t_0) \rightarrow 4 \text{ Recv } t_0$ at `Recv_293` and $0(t_0) \rightarrow 5 \text{ (Recv_312)}$, we can tell from the global graph that the node in the unselected branch will not be revisited again. Hence participant 4 (resp. 5) is waiting forever if the branch of $0 \rightarrow 5$ (resp. $0 \rightarrow 4$) is selected. So the branching condition is violated and as a result, Listing 3 is not GMC.

Suppose we rectify the problem in Listing 3, by removing the extra `go Recv(ch, done)` on line 19 and the extra `<-done` on line 23. The global graph synthesised from the corrected example is shown in Figure 6. The GMC condition in this case is valid, and therefore the program is safe and deadlock free. Note that

while checking GMC is sufficient to check the deadlock-freedom, global graphs generated from the CFSMs (such as those shown in Figure 5) offer useful information for debugging programs, see [23].

4.3 Mixed Choices

The original theory [23] does not allow a machine to have a mixed choice which has both sending and receiving from the same state. The `select` statement in Go allows to write mixed choices, hence we are required to relax this condition. The original branching condition [23] requires the sender to be unique, i.e. the unique participant should initiate sending actions. Since we are taking a synchronous model (instead of an asynchronous model in [23]), we can relax this condition by replacing the unique sender by the unique machine condition (as explained in the previous paragraph).

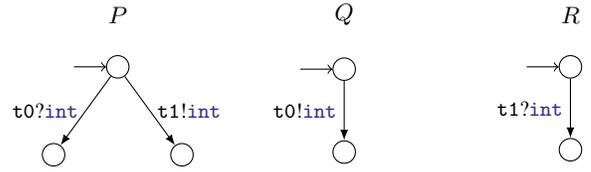


Figure 7. Mixed Choice with `select` statement.

Consider the simple mixed choice in Figure 7. In this example, *P* initiates both initial sending and receiving actions so that it satisfies the unique machine condition. This extension was implemented in the GMC tool to verify deadlock in Go programs with `select` statements.

4.4 Limitations

Our approach uses sets of CFSMs to represent systems of concurrent processes, and supports a flexible range of program control-flow patterns with little restrictions in their structures. Examples include for-loops, which are translated as two branches of transitions from a state: one for *exit loop* and another for *continue loop* that loops back to the starting state.

This, however, limits our support for dynamic concurrency such as creating channels in a loop or conditional creation of goroutines for error handling. They correspond to runtime spawning of new channel CFSMs and new goroutine CFSMs respectively, hence cannot be represented at static time. To overcome the limitations, CFSMs are constructed using the inference technique outlined in Section 3 for each goroutine. If a condition in the control-flow of the program decides that if a goroutine will be spawned, then a subset of all generated CFSMs – which translated from goroutines that are spawned under the same condition – are selected for global graph synthesis to check that the system is safe under the condition.

5. Case Studies

We demonstrate our approach by the following case studies, adapted from common Go concurrency patterns. Table 4 and Section 6 summarise their evaluations.

Pipeline: Prime Sieve The pipeline is a straightforward pattern for chaining together phases of concurrent computation by channels. Listing 7 is an implementation of the Prime sieve of Eratosthenes in Go, which chains together goroutines with channels as prime filters. The example prints the first 10 prime numbers.

In the code, there are three goroutines (`main`, `filter` and `generate`) and two channels. The `generate` goroutine continuously sends numbers to the `ch` channel.

```

1 // Input generator (send and increment)
2 func generate(ch chan<- int) {
3     for i := 2; ; i++ { ch <- i }
4 }
5 // A filter for given prime
6 func filter(in <-chan int, out chan<- int, prime int) {
7     for {
8         i := <-in
9         if i%prime != 0 { out <- i }
10    }
11 }
12 func main() {
13     ch := make(chan int)
14     go generate(ch) // Number generator
15     for i := 0; i < 10; i++ {
16         prime := <-ch
17         fmt.Println(prime)
18         ch1 := make(chan int)
19         go filter(ch, ch1, prime) // Extend pipeline
20         ch = ch1
21     }
22 }

```

Listing 7. Implementation of concurrent prime sieve.

In main, a prime number is read from the `ch` channel and is used for the prime filter. The `filter` goroutine repeatedly takes an input channel (`ch`), and an output channel (`ch1`) and filters out input values that are divisible by prime. Finally, in the next iteration the outputs channel of the `filter` is used as the new input channel for the next filter on line 20. The CFSMs for Listing 7 is given in Figure 8.

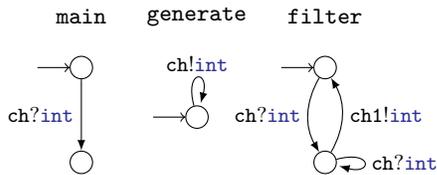


Figure 8. Goroutine CFSMs for primesieve.

Fan-in: Multiplexing with `select` The fan-in pattern uses the `select` primitive to multiplex multiple communication by receiving from a set of input channels, then deliver the results in a single output channel. A simple implementation of the fan-in pattern is captured in Listing 8, where the `fanin` function merges two input streams into one output stream.

```

1 func work(ch chan int) { for { ch <- 42 } }
2 func fanin(input1, input2 <-chan int) <-chan string {
3     ch := make(chan string)
4     go func() {
5         for {
6             select {
7                 case s := <-input1: ch <- s
8                 case s := <-input2: ch <- s
9             }
10        }
11    }()
12    return ch
13 }
14 func main() {
15     input1, input2 := make(chan int), make(chan int)
16     go work(input1)
17     go work(input2)
18     c := fanin(input1, input2)
19     for {
20         fmt.Println(<-c)
21     }
22 }

```

Listing 8. Implementation of the fan-in pattern.

Figure 9 shows the CFSMs for the fan-in implementation. The two worker uses `input1` and `input2` channels respectively to send their results. The `fanin` function reads from the two channels on line 7 and 8 to combine them into a single channel `ch` in an anonymous goroutine. The main function then loops over the combined channel `c` returned by `fanin` to print out the received value on line 20.

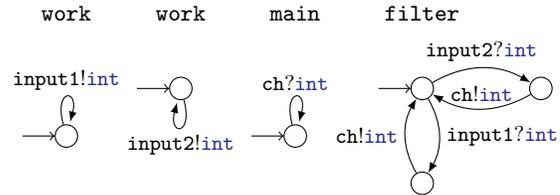


Figure 9. Goroutine CFSMs for the fan-in pattern.

The `fanin` pattern can be implemented alternatively, where the `select` statement has a `default` branch when no more values will be sent. The output channel `ch` is explicitly closed to notify the downstream. The syntax `for val := range ch` is a shorthand for reading a value `val` from a channel `ch`, test if the channel is closed, and repeat until the channel is closed. This alternative implementation highlights the use of `close` to close a channel and is shown in Listing 9

```

1 func fanin(input1, input2 <-chan int) <-chan string {
2     ch := make(chan string)
3     go func() {
4         for {
5             select {
6                 case s := <-input1: ch <- s
7                 case s := <-input2: ch <- s
8                 default: close(ch); return // No more values
9             }
10        }
11    }()
12    return ch
13 }
14 func main() {
15     input1, input2 := make(chan int), make(chan int)
16     go work(input1)
17     go work(input2)
18     for c := range fanin(input1, input2) { // Exit when
19         // channel closed
20         fmt.Println(<-c)
21     }
22 }

```

Listing 9. Implementation of the alternative fan-in pattern.

Figure 10 shows the CFSMs of Listing 9. The two work goroutines are the same as those in Figure 9. The difference between the two implementations is the `close(ch)` which sends a STOP message to the channel. The STOP message is propagated to the main goroutine which decides whether to loop receiving more values or terminate.

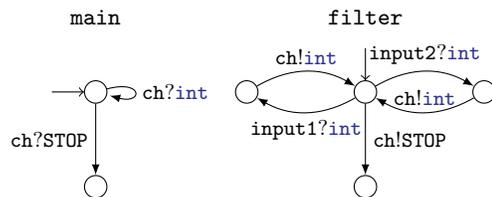


Figure 10. Goroutine CFSMs for the alternative fan-in pattern.

Table 4. Experimental evaluation.

	LoC.	Local Type nodes	Channels	Goroutines	Safe	Inference Time (s)	Synthesis Time (s)
deadlock (Listing 3)	26	16	2	5	No	0.74	0.36
fanin	36	23	3	4	Yes	0.42	1.47
fanin-alt	37	26	3	4	Yes	0.42	1.97
primesieve	38	20	2	3	No*	0.40	0.27
htcat	721	9632	8	11	Yes	1.38	2.72

6. Experimental Evaluation

We evaluate our static analysis tool with the case studies, and a real software written in Go that uses concurrency.

deadlock The deadlock example in Listing 3 demonstrates the strength of our static analysis approach as the runtime deadlock detector does not consider the program to deadlock. Our synthesis shows that the `main` goroutine in the program is not representable, as it is impossible for both of the `Recv` goroutines to send the final done message to `main`. Figure 3 shows the CFSMs generated from the source code. Note that machine 5 is missing because the `Work` goroutine does not contain any communication. As explained in the previous section, the deadlock can be removed and the corrected global graph obtained in Figure 5, and satisfies GMC.

fanin The `fanin` implementation is safe. In the original implementation where the inputs are continuous streams of numbers, and the `fanin` function combines both streams into a single stream. The output of the `fanin` is printed continuously.

fanin-alt The alternative `fanin` implementation where the output channel is explicitly closed is also safe, as the termination of the channel is explicit (as a `STOP` message in the CFSM model). The output of the `fanin` was updated to use the `for-range` syntax to take into consideration of the closing of the channel.

primesieve Concurrent prime sieve is unsafe because there is an infinite number of send generated by `generate` and `filter`, yet the number of receive from `main` is limited. In practice this does not affect the program execution because the final filter thread simply blocks if `main` is not receiving and `filter` is sending the latest prime. The goroutine CFSMs are shown in Figure 8. The global graph synthesis confirms that the `filter` CFSM is not representable in the global graph. This is a false positive result as the program is safe in practice as long as the main loop is bounded, but cannot be verified statically unless the loops are unrolled.

htcat `htcat`² is a concurrent HTTP file download tool from Heroku. It issues multiple HTTP GET in parallel to download a file and coordinate the merging of fragments using channels. In the main body of the tool, based on the response header from the HTTP server, the file to download is split into fragments and GET requests issued in goroutines. When the download of a fragment is complete, the corresponding goroutines issue a registration message with the downloaded content to the main goroutine. The main goroutine uses a `for-select` loop to wait for fragments to arrive, possibly out-of-order. Another `for-select` case, is to receive a cancel message from the user, which terminates the download process immediately. A final case is to receive a number to determine the number of fragments left. The other channels in the tool are for handling errors. In our analysis, we obtained 19 CFSMs (11 goroutines, 8 channels) from the Go program, and was reduced from 9632 local session graph nodes extracted. Most of the extracted session graph nodes are control-flow nodes such as labels or jumps. We further isolate the error handling goroutines manually from the

main program (which does not spawn new goroutines) to verify deadlock-freedom, due to the limitations explained in Section 4.4. An example from `htcat` is shown below, depicting one of the main functions which creates goroutines for fetching fragments.

```

...
if err != nil {
    go cat.d.cancel(err)
    return
}
...
go func() { // start GET worker

```

The function is short-circuited if an error is detected during the initialisation and a new goroutine `cat.d.cancel` is spawned to terminate the rest of the program. Since the `cancel` goroutine will not be created in a normal execution, the CFSM for `cancel` is not included in the global graph synthesis but will be considered separately.

7. Related Work

Deadlock Detection and Checking based on Graphs Process calculi such as CSP are models for a number of previous works on deadlock detection in parallel systems. Huang [17] proposed a distributed deadlock detection algorithm based on wait-for graphs for systems with CSP-like communication. Later Zhao et al. [37] introduced a runtime monitoring approach for deadlock detection for Occam based on wait-for graphs. Occam is a concurrent programming language built on CSP. Like Go, it uses channels as a medium for communication between processes. The work in [4] also applied wait-for/state graphs for deadlock detection in Java and X10, formalising a phaser-based calculus. These approaches prevent detection of potential deadlocks which do not manifest themselves at runtime at specific runs. Martin [25] proposed a static analysis tool³ to detect deadlocks in CSP, but is not applicable to Go directly since there is no precise correspondence between CSP and Go primitives.

Message Flow Graphs (MFGs) by Ladkin et al. [19, 20, 22, 21] construct a global FSM from local communications but assume senders and receivers are statically determined. In this work, the input model uses a dynamic channel abstraction and requires resolving the channels into actual senders and receivers before such construction can be used. Moreover, our work can handle mixed choices in the use of `select` primitive which is more general.

Naik et al. [28] delivered an unsound but effective deadlock detection tool which targets multi-threaded Java by analysing necessary conditions for deadlocks. Other deadlock detection approaches such as concolic execution [8] are between static analysis and runtime detection.

To our best knowledge, the present work is the first work to statically verify deadlocks in the Go language based on the formal type theory of session types.

Verification of Concurrent Programs based on Session Types Many programming languages based on session types have been

²<https://github.com/htcat/htcat> (414 watchers on GitHub)

³The tool is available at <http://wotug.org/parallel/theory/formal/csp/Deadlock/>

developed in the past decade. See [36] for a recent comprehensive survey. Here we discuss closely related work. Hu et al. [16] developed Session Java (SJ) with binary session types for socket programming, which was further extended with multiparty session types by Sivaramakrishnan et al. [34]. Both works statically verify the extended Java source code to ensure the lack of communication errors. Session C [31] uses a static approach, and instead of extending the language, a custom communication API is provided for message-passing in C. Similarly a static type checking for MPI programs based on global dependent session types is studied in [24].

Recent works based on multiparty session types use dynamic, runtime monitoring for verifications of distributed programs. For example, runtime monitoring of Python extended to interrupts and actors are studied in [6] and [29], respectively. The above works use a protocol description language, Scribble [33], developed with Red Hat. Another approach taken in [30, 15] is code generation from Scribble to ensure deadlock-freedom by construction. The work [30] generates parallelised MPI code combining MPI backbones generated from parameterised Scribble and sequential Kernels; and the work [15] generates Java APIs from Scribble [33] and ensures safety of Java programs with light-weight linearity check of channel usages. All of the above works use the top-down approach based on the end-point projection. To our best knowledge, there is no work which applies the synthesis of multiparty session types to deadlock detection for real programming languages.

In this work, we analysed the Go programming language natively without extra annotations or runtime libraries, thanks to the language’s first-class support for concurrent programming.

8. Conclusion and Future Work

We have presented the first static analyser for Go that uses a global session graph synthesis tool [23] to detect communication deadlocks. We have applied our tool to Go code and open source projects with up to 700 lines of code. We demonstrated a combination of session type inference and synthesis that is directly applicable to verify deadlocks in a practical concurrent programming language.

Immediate future work includes extending our approach to support buffered channels which enable asynchronous communications with bounded queues. In addition to channels, Go provides the `sync.WaitGroup` package in its standard library for barrier-style synchronisation, implemented in atomic instructions and semaphores. These features can be encoded in CFSMs and checked with the global graph synthesis approach. Finally, some of the dynamic concurrency limitations can be overcome by performing multiple global graph synthesis per program on different subset of CFSMs, and the generation of subsets can be automated. We leave these extensions to future work.

Acknowledgments

The authors would like to thank Julien Lange for his comments and revisions of the synthesis tool, Raymond Hu and the anonymous reviewers for their comments and valuable suggestions. This work is supported in part by EPSRC projects EP/K034413/1, EP/K011715/1, and EP/L00058X/1; and by EU FP7 Project under grant agreement 612985 (UpScale).

References

[1] On-line appendix. <http://www.doc.ic.ac.uk/~cn06/go>.
 [2] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell. The Problem of Programming Language Concurrency Semantics. In *ESOP*, LNCS, pages 283–307. Springer, 2015.

[3] D. Brand and P. Zafiropulo. On communicating finite-state machines. *JACM*, 30(2):323–342, 1983.
 [4] T. Cogumbreiro, R. Hu, F. Martins, and N. Yoshida. Dynamic deadlock verification for general barrier synchronisation. In *PPoPP 2015*, pages 150–160. ACM, 2015.
 [5] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computer Science Engineering, IEEE*, 5(1):46–55, 1998.
 [6] R. Demangeon, K. Honda, R. Hu, R. Neykova, and N. Yoshida. Practical interruptible conversations: Distributed dynamic verification with multiparty session types and python. *FMSD*, pages 1–29, 2015.
 [7] P.-M. Denielou and N. Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP*, volume 7966 of LNCS, pages 174–186, 2013.
 [8] M. Eslamimehr and J. Palsberg. Sherlock: scalable deadlock detection for concurrent programs. In *FSE 2014*, pages 353–365. ACM, 2014.
 [9] S. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *JFP*, 2009.
 [10] Google. Go Single Static Assignment (SSA) IR packages. <https://golang.org/x/tools/ssa>.
 [11] C. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.
 [12] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
 [13] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP’98*, volume 1381 of LNCS, pages 22–138. Springer, 1998.
 [14] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL ’08*, volume 5201 of LNCS, pages 273–284. ACM Press, 2008.
 [15] R. Hu and N. Yoshida. Hybrid Session Verification through Endpoint API Generation. In *FASE 2016*, LNCS. Springer, 2016. To appear.
 [16] R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In *ECOOP 2008*, volume 5142 of LNCS, pages 516–541. Springer, 2008.
 [17] S.-T. Huang. A Distributed Deadlock Detection Algorithm for CSP-like Communication. *ACM TOPLAS*, 12(1):102–122, 1990.
 [18] Julien Lange. gmc-synthesis tool homepage. <https://bitbucket.org/julien-lange/gmc-synthesis>.
 [19] P. Ladkin and B. Simons. Compile-time analysis of communicating processes. In *ICS’92*, pages 248–259. ACM, 1992.
 [20] P. Ladkin and B. Simons. Static Deadlock Analysis for CSP-type Communication. In D. S. Fussell and M. Malek, editors, *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems*. Kluwer Academic Publishers, 1995.
 [21] P. B. Ladkin and S. Leue. Interpreting Message Flow Graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.
 [22] P. B. Ladkin and B. B. Simons. *Static Analysis of Interprocess Communication*. LNCS. Springer-Verlag, 1995.
 [23] J. Lange, E. Tuosto, and N. Yoshida. From communicating machines to graphical choreographies. In *POPL 2015*, pages 221–232. ACM, 2015.
 [24] H. A. López, E. R. B. Marques, F. Martins, N. Ng, C. Santos, V. T. Vasconcelos, and N. Yoshida. Protocol-based verification of message-passing parallel programs. In *OOPSLA’15*, pages 280–298. ACM, 2015.
 [25] J. Martin. *The Design and Construction of Deadlock-Free Concurrent Systems*. PhD thesis, University of Buckingham, 1996.
 [26] R. Milner. *A Calculus of Communicating Systems*, volume 92. Springer-Verlag, 1980.
 [27] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Process, Part I. *Information and Computation*, pages 1–77, 1992.

- [28] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective Static Deadlock Detection. In *ICSE'09*, pages 386–396. IEEE Computer Society, 2009.
- [29] R. Neykova and N. Yoshida. Multiparty session actors. In *COORDINATION 2014*, volume 8459 of *LNCS*, pages 131–146. Springer, 2014.
- [30] N. Ng, J. G. Coutinho, and N. Yoshida. Protocols by default: Safe mpi code generation based on session types. In *CC 2015*, volume 9031 of *LNCS*, pages 212–232. Springer, 2015.
- [31] N. Ng, N. Yoshida, and K. Honda. Multiparty session c: Safe parallel programming with message optimisation. In *TOOLS 2012*, volume 7304 of *LNCS*, pages 202–218. Springer, 2012.
- [32] R. Pike. Go at Google. In *SPLASH*, pages 5–6, New York, NY, USA, 2012. ACM.
- [33] Scribble project home page. <http://www.scribble.org>.
- [34] K. Sivaramakrishnan, K. Nagaraj, L. Ziarek, and P. Eugster. Efficient session type guided distributed interaction. In *COORDINATION 2010*, volume 6116 of *LNCS*, pages 152–167. Springer Berlin Heidelberg, 2010.
- [35] K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413, 1994.
- [36] Survey on languages based on behavioural types. <http://www.behavioural-types.eu/publications/WG3-State-of-the-Art.pdf>.
- [37] J. Zhao, H. Abe, Y. Nomura, J. Cheng, and K. Ushijima. Run-Time Detection of Communication Deadlocks in occam 2 Programs. *Correct Models of Parallel Computing*, pages 97–107, 1997.

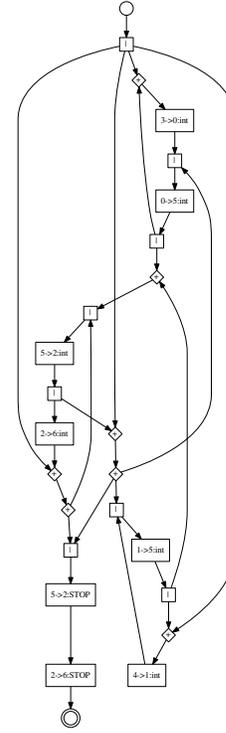


Figure 12. Global graph of fanin-alt from Listing 9.

Appendix

A. Global graphs from case study examples

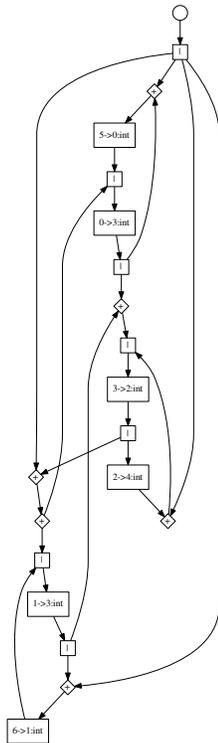


Figure 11. Global graph of fanin from Listing 8.

