Static Trace-Based Deadlock Analysis for Synchronous Mini-Go

Kai Stadtmüller[1], Martin Sulzmann[1], and Peter Thiemann[2]

[1] Faculty of Computer Science and Business Information Systems
Karlsruhe University of Applied Sciences
Moltkestrasse 30, 76133 Karlsruhe, Germany
kai.stadtmueller@live.de
martin.sulzmann@hs-karlsruhe.de
[2] Faculty of Engineering, University of Freiburg
Georges-Köhler-Allee 079, 79110 Freiburg, Germany
thiemann@acm.org

Abstract. We consider the problem of static deadlock detection for programs in the Go programming language which make use of synchronous channel communications. In our analysis, regular expressions extended with a fork operator capture the communication behavior of a program. Starting from a simple criterion that characterizes traces of deadlock-free programs, we develop automata-based methods to check for deadlock-freedom. The approach is implemented and evaluated with a series of examples.

1 Introduction

The Go programming language [6] attracts increasing attention because it offers an elegant approach to concurrent programming with message-passing in the style of Communicating Sequential Processes (CSP) [9]. Although message passing avoids many of the pitfalls of concurrent programming with shared state (atomicity violations, order violations, issues with locking, and so on), it still gives rise to problems like deadlock. Hence, the goal of our work is the static detection of deadlocks in Go programs which make use of (synchronous) message-passing using the unbuffered version of Go's channels.

1.1 Related work

Leaving aside data races, deadlocks constitute one of the core problems in concurrent programming. However, most work on static detection of deadlocks on the programming language level deals with shared-memory concurrency.

Boyapati and coworkers [1] define a type-based analysis that relies on a partial order on locks and guarantees that well-typed programs are free of data races and deadlocks. The approaches by Williams and coworkers [22] and Engler and Ashcraft [5] detect cycles in a precomputed static lock-order graph to highlight potential deadlocks. In distributed and database systems, most approaches are dynamic but also involve cycle detection in wait-for graphs (e.g., [10]). In these approaches, the main points of interest are the efficiency of the cycle detection algorithms and the methods employed for the construction and maintenance of the wait-for graph.

Mercouroff [16] employs abstract interpretation for an analysis of CSP programs using an abstract domain that approximates the number of messages sent between processes. Colby [4] presents an analysis that uses control paths to identify threads that may be created at the same point and constructs the communication topology of the program. A more precise control-flow analysis was proposed by Martel and Gengler [15]. Similar to our approach, in their work the accuracy of the analysis is enhanced by analyzing finite automata to eliminate some impossible communication traces.

For message-passing programs, there are elaborate algorithms that attempt accurate matching of communications in process calculi (e.g., the work of Ladkin and Simon [14]). However, they consider messages between fixed partners whereas we consider communication between multiple partners on shared channels.

Further analysis of message passing in the context of Concurrent ML (CML) [20] is based on effect systems that abstract programs into regular-expression-like behaviors with the goal of detecting finiteness of communication topologies [19]. The deadlock detection analysis of Christakis and Sagonas [3] also constructs a static graph and searches it for cycles. Specific to Go, the paper by Ng and Yoshida [18] translates Go programs into a core calculus with session types and then attempts to synthesize a global choreography that subsumes all session. A program is deemed deadlock-free if this synchronization succeeds and satisfies some side condition. Like our work, they consider a fixed number of processes and synchronous communication. Section 6 contains a more detailed comparison with this work.

Kobayashi [13] considers deadlock detection for the π-calculus [17]. His type inference algorithm infers usage constraints among receive and send operations. In essence, the constraints represent a dependency graph where the program is deadlock-free if there are no circular dependencies among send and receive operations. The constraints are solved by reduction to Petri net reachability [12]. A more detailed comparison with Kobayashi's work is given in Section 6.

1.2 Contributions

Common to all prior work is their reliance on automata-/graph-based methods. The novelty of our work lies in the use of a symbolic deadlock detection method based on forkable behavior.

Forkable behaviors in the form of regular expressions extended with fork and general recursion were introduced by Nielson and Nielson [19] to analyze the communication topology of CML (which is based on ideas from CSP, just like Go). In our own recent work [21], we establish some important semantic foundations for forkable behaviors such as a compositional trace-based semantics and a symbolic Finite State Automata (FSA) construction method via Brzozowski-style

```
func sel(x, y chan bool) {
        z := make(chan bool)
        go func() { z <- (<-x) }()
        go func() { z <- (<-y) }()
        <-z
}
func main() {
        x := make(chan bool)
        y := make(chan bool)
        go func() { x <- true }()
        go func() { y <- false }()
        sel(x,y)
        sel(x,y)
}
```

Listing 1.1. Message passing in Go

derivatives [2]. In this work, we apply these results to statically detect deadlocks in Go programs.

Specifically, we make the following contributions:

- We formalize Mini-Go, a fragment of the Go programming language which is restricted to synchronous message-passing (Section 3).
- We approximate the communication behavior of Mini-Go programs with forkable behaviors (Section 4).
- We define a criterion for deadlock-freedom in terms of the traces resulting from forkable behaviors. We give a decidable check for deadlock-freedom for a large class of forkable behaviors by applying the FSA construction method developed in prior work [21]. We also consider improvements to eliminate false positives (Section 5).
- We evaluate our approach with examples and conduct a comparison with closely related work (Section 6).

The appendix contains further details such as proofs etc.

2 Highlights

Before we delve into deadlocks and deadlock detection, we first illustrate the message passing concepts found in Go with the example program in Listing 1.1. The `main` function creates two synchronous channels `x` and `y` that transport Boolean values. Go supports (a limited form of) type inference and therefore no type annotations are required. We create two threads using the `go` ***exp*** statement. It takes an expression ***exp*** and executes it in a newly spawned go-routine (a thread). Each of these expressions calls an anonymous function that performs a send operation on one of the channels. In Go, we write `x <- true` to send value `true` via channel `x`. Then we call the function `sel` twice. This function creates another Boolean channel `z` locally and starts two threads that "copy" a value from one of the argument channels to `z`. In Go, we write `<-x` to receive a value via channel `x`. Thus, `z <- (<-x)` sends a value received via channel `x` to channel `z`.

So, the purpose of `sel` is to choose a value which can either be received via channel `x` or channel `y`. As each channel is supplied with a value, each of the two calls to `sel` might be able to retrieve a value. While there is a schedule such that the `main` program runs to completion, it is also possible that execution of the second `sel` call will get stuck. Consider the case that in the first call to `sel` both helper threads get to execute the receive operations on `x` and `y` and forward the values to channel `z`. In this case, only one of the values will be picked up by the `<-z` and returned, but the local thread with the other value will be blocked forever waiting for another read on `z`. In the second call to `sel`, none of the local threads can receive a value from `x` or `y`, hence there will be no send operation on `z`, so that the final receive `<-z` remains blocked.

Our approach to detect such devious situations is to express the communication behavior of a program in terms of forkable behaviors. For the `main` function in Listing 1.1, we obtain the following forkable behavior

$$Fork(x!) \cdot Fork(y!) \cdot Fork(x? \cdot z_1!) \cdot Fork(y? \cdot z_1!) \cdot z_1? \cdot Fork(x? \cdot z_2!) \cdot Fork(y? \cdot z_2!) \cdot z_2?$$

We abstract away the actual values sent and write $x!$ to denote sending a message to channel `x` and $x?$ to denote reception via channel `x`. $Fork()$ indicates a forkable (concurrent) behavior which corresponds to `go` statements in the program. The concatenation operator $\cdot$ connects two forkable behaviors in a sequence. The function calls to `sel` are inlined and the local channels renamed to z_1 and z_2, respectively.

The execution schedules of `main` can be described by a matching relation for forkable behaviors where we symbolically rewrite expressions. Formal details follow later. Here are some possible matching steps for our example.

$$\begin{array}{ll}
 & Fork(x!) \cdot Fork(y!) \cdot Fork(x? \cdot z_1!) \cdot Fork(y? \cdot z_1!) \cdot z_1? \cdot \\
 & Fork(x? \cdot z_2!) \cdot Fork(y? \cdot z_2!) \cdot z_2? \\
\Rightarrow & \{\!\{\underline{x!}, y!, \underline{x?} \cdot z_1!, y? \cdot z_1!, z_1? \cdot Fork(x? \cdot z_2!) \cdot Fork(y? \cdot z_2!) \cdot z_2?\}\!\} \\
\xRightarrow{x! \cdot x?} & \{\!\{y!, \underline{z_1!}, y? \cdot z_1!, \underline{z_1?} \cdot Fork(x? \cdot z_2!) \cdot Fork(y? \cdot z_2!) \cdot z_2?\}\!\} \\
\xRightarrow{z_1! \cdot z_1?} & \{\!\{y!, y? \cdot z_1!, Fork(x? \cdot z_2!) \cdot Fork(y? \cdot z_2!) \cdot z_2?\}\!\} \\
\Rightarrow & \{\!\{y!, y? \cdot z_1!, x? \cdot z_2!, y? \cdot z_2!, z_2?\}\!\} \\
\xRightarrow{y! \cdot y? \cdot z_2! \cdot z_2?} & \{\!\{y? \cdot z_1!, x? \cdot z_2!\}\!\}
\end{array}$$

We first break apart the expression into its concurrent parts indicated by the multiset notation $\{\!\{\cdot\}\!\}$. Then, we perform two rendezvous (synchronization) steps where the partners involved are underlined. In essence, the first call to `sel` picks up the value sent via channel `x`. The last step where we combine two synchronization steps (and also omit underline) shows that the second call to `sel` picks up the value sent via channel `y`. Note that the main thread terminates but as for each call to `sel` one of the helper threads is stuck our analysis reports a deadlock.

As mentioned above, another possible schedule is that the first call to `sel` picks up both values sent via channels `x` and `y`. In terms of the matching relation, we find the following

$$\begin{array}{l} \mathit{Fork}(x!) \cdot \mathit{Fork}(y!) \cdot \mathit{Fork}(x? \cdot z_1!) \cdot \mathit{Fork}(y? \cdot z_1!) \cdot z_1? \cdot \\ \mathit{Fork}(x? \cdot z_2!) \cdot \mathit{Fork}(y? \cdot z_2!) \cdot z_2? \\ \xRightarrow{x! \cdot x? \cdot y! \cdot y? \cdot z_1! \cdot z_1?} \{\!\{ z_1!, x? \cdot z_2!, y? \cdot z_2!, z_2? \}\!\} \end{array}$$

As we can see, the second helper thread of the first call to `sel` is stuck, both helper threads of the second call are stuck as well as the main thread. In fact, this is the deadlock reported by our analysis as we attempt to find minimal deadlock examples.

The issue in the above example can be fixed by making use of *selective* communication to non-deterministically choose among multiple communications.

```
func selFixed(x, y chan bool) {
  select {
    case z = <-x:
    case z = <-y:
  }
}
```

The select statement blocks until one of the cases applies. If there are multiple select cases whose communication is enabled, the Go run-time system 'randomly' selects one of those and proceeds with it. Based on a pseudo-random number the select cases are permuted and tried from top to bottom. Thus, the deadlocking behavior observed above disappears as each call to `selFixed` picks up either a value sent via channel `x` or channel `y` but it will never consume values from both channels.

3 Mini-Go

We formalize a simplified fragment of the Go programming language where we only consider a finite set of pre-declared, synchronous channels. For brevity, we also omit procedures and first-class channels and only consider Boolean values.

Definition 1 (Syntax).

$$\begin{array}{lll} x, y, \ldots & & \textit{Variables, Channel Names} \\ s & ::= v \mid \mathit{Chan} & \textit{Storables} \\ v & ::= \mathit{True} \mid \mathit{False} & \textit{Values} \\ vs & ::= [\,] \mid v : vs & \textit{Value Queues} \\ b & ::= v \mid x \mid b \&\& b \mid !b & \textit{Expressions} \\ e, f & ::= x \leftarrow y^{\mathsf{r}} \mid y^{\mathsf{s}} \leftarrow b & \textit{Receive/Send} \\ p, q & ::= \mathsf{skip} \mid \mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ q \mid \mathsf{while}\ b\ \mathsf{do}\ p \mid p; q & \textit{Commands} \\ & \mid\ \ \mathsf{select}\ [e_i \Rightarrow p_i]_{i \in I} & \textit{Communications} \\ & \mid\ \ \mathsf{go}\ p & \textit{Threads} \end{array}$$

Variables are either bound to Boolean values or to the symbol *Chan* which denotes a synchronous channel. Like in Go, we use the 'arrow' notation for the send and receive operations on channels. We label the channel name to distinguish receive from send operations. That is, from $x \leftarrow y^{\mathbf{r}}$ we conclude that y is the channel via which we receive a value bound to variable x. From $y^{\mathsf{s}} \leftarrow b$ we conclude that y is the channel to which some Boolean value is sent. Send and receive communications are shorthands for unary selections: $e = \mathsf{select}\ [e \Rightarrow \mathsf{skip}]$.

The semantics of a Mini-Go program is defined with a small-step semantics. The judgment $\langle S, \{\!\{p_1, \ldots, p_n\}\!\}\rangle \stackrel{T}{\Longrightarrow} \langle S', \{\!\{p'_1, \ldots, p'_m\}\!\}\rangle$ indicates that execution of program threads p_i may evolve into threads p'_j with trace T. The notation $\{\!\{p_1, ..., p_n\}\!\}$ represents a multi-set of concurrently executing programs p_1, ..., p_n. For simplicity, we assume that all threads share a global state S and that distinct threads have only variables bound to channels in common.

Program trace T records the communication behavior as a sequence of symbols where symbol $x!$ represents a send operation on channel x and symbol $x?$ represents a receive operation on channel x. As we assume synchronous communication, each communication step involves exactly two threads as formalized in the judgment $\langle S, \{\!\{p, q\}\!\}\rangle \stackrel{T}{\Longrightarrow} \langle S', \{\!\{p', q'\}\!\}\rangle$.

The semantics of Boolean expressions is defined with a big-step semantics judgment $S \vdash b \Downarrow v$, where S is the state in which expression b evaluates to value v. For commands, the judgment $S \vdash p \Rightarrow q$ formalizes one (small-) step that executes a single statement. Thus, we are able to switch among different program threads after each statement. Here are the details.

Definition 2 (State). *A state S is either empty, a mapping, or an override a state with a new mapping:* $S ::= () \mid (x \mapsto s) \mid S \lhd (x \mapsto s)$

We write $S(x)$ to denote state lookup. We assume that mappings in the right operand of the map override $\lhd$ take precedence. They overwrite any mappings in the left operand. That is, $(x \mapsto \mathit{True}) \lhd (x \mapsto \mathit{False}) = (x \mapsto \mathit{False})$. We assume that for each channel x the state contains a mapping $x \mapsto \mathit{Chan}$.

Definition 3 (Expression Semantics $S \vdash b \Downarrow v$).

$$S \vdash \mathit{True} \Downarrow \mathit{True} \qquad\qquad S \vdash \mathit{False} \Downarrow \mathit{False}$$

$$\frac{S(x) = v}{S \vdash x \Downarrow v} \qquad \frac{S \vdash b_1 \Downarrow \mathit{False}}{S \vdash b_1 \&\& b_2 \Downarrow \mathit{False}} \qquad \frac{S \vdash b_1 \Downarrow \mathit{True} \quad S \vdash b_2 \Downarrow v}{S \vdash b_1 \&\& b_2 \Downarrow v}$$

$$\frac{S \vdash b \Downarrow \mathit{False}}{S \vdash !b \Downarrow \mathit{True}} \qquad\qquad \frac{S \vdash b \Downarrow \mathit{True}}{S \vdash !b \Downarrow \mathit{False}}$$

Definition 4 (Commands $S \vdash p \Rightarrow q$).

$$\textit{(If-T)}\ \frac{S \vdash b \Downarrow \mathit{True}}{S \vdash \mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ q \Rightarrow p} \qquad \textit{(If-F)}\ \frac{S \vdash b \Downarrow \mathit{False}}{S \vdash \mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ q \Rightarrow q}$$

$$\textit{(While-F)}\ \frac{S \vdash b \Downarrow \mathit{False}}{S \vdash \mathsf{while}\ b\ \mathsf{do}\ p \Rightarrow \mathsf{skip}}$$

$$\textit{(While-T)}\ \frac{S \vdash b \Downarrow \mathit{True}}{S \vdash \mathsf{while}\ b\ \mathsf{do}\ p \Rightarrow p; \mathsf{while}\ b\ \mathsf{do}\ p} \qquad \textit{(Skip)}\ S \vdash \mathsf{skip}; p \Rightarrow p$$

$$\textit{(Reduce)}\ \frac{S \vdash p \Rightarrow p'}{S \vdash p; q \Rightarrow p'; q} \qquad \textit{(Assoc)}\ S \vdash (p_1; p_2); p_3 \Rightarrow p_1; (p_2; p_3)$$

Definition 5 (Communication Traces).

$$\begin{array}{rll} T ::= & \epsilon & \textit{empty trace} \\ \mid & x! & \textit{send event} \\ \mid & x? & \textit{receive event} \\ \mid & T \cdot T & \textit{sequence/concatenation} \end{array}$$

As we will see, the traces obtained by running a program are of a particular 'synchronous' shape.

Definition 6 (Synchronous Traces). *We say T is a* synchronous *trace if T is of the following more restricted form.*

$$T_s ::= \varepsilon \mid \alpha \cdot \bar{\alpha} \mid T_s \cdot T_s$$

where $\bar{\alpha}$ denotes the complement of α and is defined as follows: For any channel y, $\overline{y?} = y!$ and $\overline{y!} = y?$.

We assume common equality laws for traces such as associativity of $\cdot$ and ϵ acts as a neutral element. That is, $\epsilon \cdot T = T$. Further, we consider the two synchronous traces $\alpha_1 \cdot \overline{\alpha_1} \cdot ... \cdot \alpha_n \cdot \overline{\alpha_n}$ and $\overline{\alpha_1} \cdot \alpha_1 \cdot ... \cdot \overline{\alpha_n} \cdot \alpha_n$ to be equivalent.

Definition 7 (Synchronous Communications $\langle S, \{\!\{p, q\}\!\}\rangle \overset{T}{\Longrightarrow} \langle S', \{\!\{p', q'\}\!\}\rangle$).

$$\textit{(Sync)}\ \frac{\begin{array}{c} \textit{for}\ k \in I \quad l \in J\ \textit{where} \\ e_k = x \leftarrow y^{\mathbf{r}} \quad f_l = y^{\mathsf{s}} \leftarrow b \\ S_1(y) = \mathit{Chan} \quad S_1 \vdash b \Downarrow v \quad S_2 = S_1 \lhd (x \mapsto v) \end{array}}{\langle S_1, \{\!\{\mathsf{select}\ [e_i \Rightarrow p_i]_{i \in I}, \mathsf{select}\ [f_j \Rightarrow q_j]_{j \in J}\}\!\}\rangle \overset{y! \cdot y?}{\Longrightarrow} \langle S_2, \{\!\{p_k, q_l\}\!\}\rangle}$$

A synchronous communication step non-deterministically selects matching communication partners from two select statements. The sent value v is immediately bound to variable x as we consider unbuffered channels here. Programs

p_k and q_l represent continuations for the respective matching cases. The communication effect is recorded in the trace $y! \cdot y?$, which arbitrarily places the send before the receive communication. We just assume this order (as switching the order yields an equivalent, synchronous trace) and use it consistently in our formal development.

In the upcoming definition, we make use of the following helper operation: $p \mathbin{⨾} q = \begin{cases} p & q = \mathsf{skip} \\ p; q & \text{otherwise.} \end{cases}$ Thus, one rule can cover the two cases that a go statement is final in a sequence or followed by another statement. If the go statement is final, the pattern $\mathsf{go}\ p \mathbin{⨾} q$ implies that q equals skip. See upcoming rule (Fork). Similar cases arise in the synchronous communication step. See upcoming rule (Comm).

Definition 8 (Program Execution $\langle S, \{\!\{p_1, \dots, p_n\}\!\}\rangle \overset{T}{\Rightarrow} \langle S', \{\!\{p'_1, \dots, p'_m\}\!\}\rangle$).

$$\textit{(Comm)}\ \frac{\langle S, \{\!\{p_1, p_2\}\!\}\rangle \overset{T}{\Rightarrow} \langle S', \{\!\{p'_1, p'_2\}\!\}\rangle}{\langle S, \{\!\{p_1 \mathbin{⨾} p''_1, p_2 \mathbin{⨾} p''_2, p_3, \dots, p_n\}\!\}\rangle \overset{T}{\Rightarrow} \langle S', \{\!\{p'_1 \mathbin{⨾} p''_1, p'_2 \mathbin{⨾} p''_2, p_3, \dots, p_n\}\!\}\rangle}$$

$$\textit{(Step)}\ \frac{S \vdash p_1 \Rightarrow p'_1}{\langle S, \{\!\{p_1, \dots, p_n\}\!\}\rangle \overset{\varepsilon}{\Rightarrow} \langle S, \{\!\{p'_1, \dots, p_n\}\!\}\rangle}$$

$$\textit{(Fork)}\ \langle S, \{\!\{\mathsf{go}\ p_1 \mathbin{⨾} q_1, p_2, \dots, p_n\}\!\}\rangle \overset{\varepsilon}{\Rightarrow} \langle S, \{\!\{p_1, q_1, p_2, \dots, p_n\}\!\}\rangle$$

$$\textit{(Stop)}\ \langle S, \{\!\{\mathsf{skip}, p_2, \dots, p_n\}\!\}\rangle \overset{\varepsilon}{\Rightarrow} \langle S, \{\!\{p_2, \dots, p_n\}\!\}\rangle$$

$$\textit{(Closure)}\ \frac{\langle S, P\rangle \overset{T}{\Rightarrow} \langle S', P'\rangle \quad \langle S', P'\rangle \overset{T'}{\Rightarrow} \langle S'', P''\rangle}{\langle S, P\rangle \overset{T \cdot T'}{\Longrightarrow} \langle S'', P''\rangle}$$

Rule (Comm) performs a synchronous communication step whereas rule (Step) executes a single step in one of the threads. Rule (Fork) creates a new thread. Rule (Stop) removes threads that have terminated. Rule (Closure) executes multiple program steps. It uses P to stand for a multiset of commands.

We are interested in identifying *stuck programs* as characterized by the following definition.

Definition 9 (Stuck Programs). *Let $\mathcal{C} = \langle S, \{\!\{p_1, \dots, p_n\}\!\}\rangle$ where $n > 1$ be some configuration which results from executing some program p. We say that p is* stuck *w.r.t. $\mathcal{C}$ if each p_i starts with a select statement*[3] *and no reduction rules are applicable on $\mathcal{C}$. We say that p is stuck if there exists a configuration $\mathcal{C}$ such that p is stuck w.r.t. $\mathcal{C}$.*

A stuck program indicates that *all* threads are *asleep*. This is commonly referred to as a *deadlock*. In our upcoming formal results, we assume that for

[3] Recall that primitive send/receive communications are expressed in terms of select.

technical reasons there must be at least two such threads. Hence, a 'stuck' program consisting of a single thread, e.g. $x^{\mathsf{s}} \leftarrow \mathit{True}; y \leftarrow x^{\mathbf{r}}$, is not covered by the above definition. Our implementation deals with programs in which only a single or some of the threads are stuck.

Our approach to detect deadlocks is to (1) abstract the communication behavior of programs in terms of forkable behaviors, and then (2) perform some behavioral analysis to uncover deadlocks. The upcoming Section 4 considers the abstraction. The deadlock analysis is introduced in Section 5.

4 Approximation via Forkable Behaviors

Forkable behaviors extend regular expressions with a fork operator and thus allow for a straightforward and natural approximation of the communication behavior of Mini-Go programs.

Definition 10 (Forkable Behaviors [21]). *The syntax of forkable behaviors (or behaviors for short) is defined as follows:*

$$r, s, t ::= \phi \mid \varepsilon \mid \alpha \mid r + s \mid r \cdot s \mid r^* \mid \mathit{Fork}(r)$$

where α are symbols from a finite alphabet Σ.

We find the common regular expression operators for alternatives (+), concatenation ($\cdot$), repetition (*) and a new fork operator $\mathit{Fork}()$. We write ϕ to denote the empty language and ε to denote the empty word.

In our setting, symbols α are send/receive communications of the form $x!$ and $x?$, where x is a channel name (viz. Definition 5). As we assume that there are only finitely many channels, we can guarantee that the set of symbols Σ is finite.

A program p is mapped into a forkable behavior r by making use of judgments $p \rightsquigarrow r$. The mapping rules are defined by structural induction over the input p. Looping constructs are mapped to Kleene star. Conditional statements and select are mapped to alternatives and a sequence of programs is mapped to some concatenated behaviors.

Definition 11 (Approximation $p \rightsquigarrow r$).

$$\mathsf{skip} \rightsquigarrow \varepsilon \qquad \frac{p \rightsquigarrow r \quad q \rightsquigarrow s}{\mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ q \rightsquigarrow r + s} \qquad \frac{p \rightsquigarrow r}{\mathsf{while}\ b\ \mathsf{do}\ p \rightsquigarrow r^*} \qquad \frac{p \rightsquigarrow r \quad q \rightsquigarrow s}{p; q \rightsquigarrow r \cdot s}$$

$$x \leftarrow y^{\mathbf{r}} \rightsquigarrow y? \qquad y^{\mathsf{s}} \leftarrow b \rightsquigarrow y! \qquad \frac{e_i \rightsquigarrow r_i \quad p_i \rightsquigarrow s_i \quad \mathit{for}\ i \in I}{\mathsf{select}\ [e_i \Rightarrow p_i]_{i \in I} \rightsquigarrow \sum_{i \in I}\ r_i \cdot s_i}$$

$$\frac{p \rightsquigarrow r}{\mathsf{go}\ p \rightsquigarrow \mathit{Fork}(r)}$$

What remains is to verify that the communication behavior of p is safely approximated by r. That is, we need to show that all traces resulting from executing p are also covered by r.

A similar result appears already in the Nielsons' work [19]. However, there are significant technical differences as we establish connections between the traces resulting from program execution to the trace-based language semantics for forkable behaviors introduced in our prior work [21].

In that work [21], we give a semantic description of forkable behaviors in terms of a language denotation $L(r, K)$. Compared to the standard definition, we find an additional component K which represents a set of traces. Thus, we can elegantly describe the meaning of an expression $\mathit{Fork}(r)$ as the shuffling of the meaning of r with the 'continuation' K. To represent Kleene star in the presence of continuation K, we use a fixpoint operation μF that denotes the least fixpoint of F in the complete lattice formed by the powerset of Σ^*. Here, F must be a monotone function on this lattice, which we prove in prior work.

Definition 12 (Shuffling). *The (asynchronous) shuffle $v \| w \subseteq \Sigma^*$ is the set of all interleavings of words $v, w \in \Sigma^*$. It is defined inductively by*

$$\varepsilon \| w = \{w\} \qquad v \| \varepsilon = \{v\} \qquad xv \| yw = \{x\} \cdot (v \| yw) \cup \{y\} \cdot (xv \| w)$$

The shuffle operation is lifted to languages by $L \| M = \bigcup \{v \| w \mid v \in L, w \in M\}$.

Definition 13 (Forkable Expression Semantics). *For a trace language $K \subseteq \Sigma^*$, the semantics of a forkable expression is defined inductively by*

$$\begin{array}{rlrl} L(\phi, K) &= \emptyset & L(r + s, K) &= L(r, K) \cup L(s, K) \\ L(\varepsilon, K) &= K & L(r \cdot s, K) &= L(r, L(s, K)) \\ L(x, K) &= \{x \cdot w \mid w \in K\} & L(r^*, K) &= \mu\, \lambda X. L(r, X) \cup K \\ & & L(\mathit{Fork}(r), K) &= L(r) \| K \end{array}$$

As a base case, we assume $L(r) = L(r, \{\varepsilon\})$.

Next, we show that when executing some program p under some trace T, the resulting program state can be approximated by the left quotient of r w.r.t. T where r is the approximation of the initial program p. This result serves two purposes. (1) All communication behaviors found in a program can also be found in its approximation. (2) As left quotients can be computed via Brzozowski's derivatives [2], we can employ his FSA methods for static analysis. We will discuss the first point in the following. The second point is covered in the subsequent section.

If L_1 and L_2 are sets of traces, we write $L_1 \backslash L_2$ to denote the left quotient of L_2 with L_1 where $L_1 \backslash L_2 = \{w \mid \exists v \in L_1. v \cdot w \in L_2\}$. We write $x \backslash L_1$ as a shorthand for $\{x\} \backslash L_1$. For a word w we give the following inductive definition: $\varepsilon \backslash L = L$ and $x \cdot w \backslash L = w \backslash (x \backslash L)$.

To connect approximations of resulting programs to left quotients, we introduce some matching relations which operate on behaviors. To obtain a match we effectively rewrite a behavior into (parts of) some left quotient. Due to the fork operation, we may obtain a multiset of (concurrent) behaviors written $\{\!\!\{r_1, ..., r_n\}\!\!\}$. We sometimes use R as a short-hand for $\{\!\!\{r_1, ..., r_n\}\!\!\}$. As in the case of program execution (Definition 8), we introduce a helper operation $r \bullet s = \begin{cases} r & s = \varepsilon \\ r \cdot s & \text{otherwise} \end{cases}$ to cover cases where a fork expression is either the final expression, or possibly followed by another expression. We write $\cdot \Rightarrow \cdot$ as a short-hand for $\cdot \stackrel{\varepsilon}{\Rightarrow} \cdot$. We also treat r and $\{\!\!\{r\}\!\!\}$ as equal.

Definition 14 (Matching Relation).

$$\boxed{r \stackrel{T}{\Rightarrow} s}$$

$$\textit{(L)}\ r + s \Rightarrow r \qquad\qquad \textit{(R)}\ r + s \Rightarrow s$$

$$\textit{(K}_n\textit{)}\ r^* \Rightarrow r \cdot r^* \qquad \textit{(K}_0\textit{)}\ r^* \Rightarrow \varepsilon \qquad \textit{(X)}\ \alpha \cdot r \stackrel{\alpha}{\Rightarrow} r$$

$$\textit{(A1)}\ \varepsilon \cdot r \Rightarrow r \qquad \textit{(A2)}\ \frac{r \Rightarrow s}{r \cdot t \Rightarrow s \cdot t} \qquad \textit{(A3)}\ (r \cdot s) \cdot t \Rightarrow r \cdot (s \cdot t)$$

$$\boxed{\{\!\!\{r_1, \ldots, r_m\}\!\!\} \stackrel{T}{\Rightarrow} \{\!\!\{s_1, \ldots, s_n\}\!\!\}}$$

$$\textit{(F)}\ \{\!\!\{\mathit{Fork}(r) \bullet s, r_1, \ldots, r_n\}\!\!\} \stackrel{\varepsilon}{\Rightarrow} \{\!\!\{s, r, r_1, \ldots, r_n\}\!\!\} \qquad \textit{(C)}\ \frac{R \stackrel{T}{\Rightarrow} R' \quad R' \stackrel{T'}{\Rightarrow} R''}{R \stackrel{T \cdot T'}{\Longrightarrow} R''}$$

$$\textit{(S1)}\ \frac{r \stackrel{T}{\Rightarrow} s}{\{\!\!\{r, r_1, \ldots, r_n\}\!\!\} \stackrel{T}{\Rightarrow} \{\!\!\{s, r_1, \ldots, r_n\}\!\!\}} \qquad \textit{(S2)}\ \{\!\!\{\varepsilon, r_1, \ldots, r_n\}\!\!\} \stackrel{\varepsilon}{\Rightarrow} \{\!\!\{r_1, \ldots, r_n\}\!\!\}$$

We establish some basic results for the approximation and matching relation. The following two results show that matches are indeed left quotients.

Proposition 1. *Let r, s be forkable behaviors and T be a trace such that $r \stackrel{T}{\Rightarrow} s$. Then, we find that $L(s) \subseteq T \backslash L(r)$.*

Proposition 2. *Let r_1,...,r_m, s_1,...,s_n be forkable behaviors and T be a trace such that $\{\!\!\{r_1, \ldots, r_m\}\!\!\} \stackrel{T}{\Rightarrow} \{\!\!\{s_1, \ldots, s_n\}\!\!\}$. Then, we find that $L(s_1) \| ... \| L(s_n) \subseteq T \backslash (L(r_1) \| ... \| L(r_m))$.*

Finally, we establish that all traces resulting during program execution can also be obtained by the match relation. Furthermore, the resulting behaviors are approximations of the resulting programs.

Proposition 3. *If $S \vdash p \Rightarrow q$ and $p \leadsto r$ then $r \Rightarrow s$ for some s where $q \leadsto s$.*

Proposition 4. *If $\langle S, \{\!\!\{p_1, ..., p_m\}\!\!\}\rangle \stackrel{T}{\Rightarrow} \langle S', \{\!\!\{q_1, ..., q_n\}\!\!\}\rangle$ and $p_i \leadsto r_i$ for $i = 1, ..., m$ then $\{\!\!\{r_1, ..., r_m\}\!\!\} \stackrel{T}{\Rightarrow} \{\!\!\{s_1, ..., s_n\}\!\!\}$ where $q_j \leadsto s_j$ for $j = 1, .., n$.*

5 Static Analysis

Based on the results of the earlier section, all analysis steps can be carried out on the forkable behavior instead of the program text. In this section, we first develop a 'stuckness' criterion in terms of forkable behaviors to identify programs with a potential deadlock. Then, we consider how to statically check stuckness.

5.1 Forkable Behavior Stuckness Criterion

Definition 15 (Stuck Behavior). *We say that r is* stuck *if and only if there exists $r \overset{T}{\Longrightarrow} \varepsilon$ for some non-synchronous trace T.*

Recall Definition 6 for a description of synchronous traces.

The following result shows that if the stuck condition does *not* apply, we can guarantee the absence of a deadlock. That is, non-stuckness implies deadlock-freedom.

Proposition 5. *Let p be a stuck program and r be a behavior such that $p \rightsquigarrow r$. Then, r is stuck.*

The above result does not apply to stuck programs consisting of a single thread. For example, consider $p = x^{\mathbf{s}} \leftarrow \mathit{True}; y \leftarrow x^{\mathbf{r}}$ and $r = x! \cdot x?$ where $p \rightsquigarrow r$. Program p is obviously stuck, however, r is not stuck because any matching trace for r is synchronous. For example, $r \overset{x! \cdot x?}{\Longrightarrow} \varepsilon$. Hence, Definition 9 assumes that execution of program p leads to some state where all threads are asleep, so that we can construct a non-synchronous trace for the approximation of p.

Clearly, the synchronous trace $x! \cdot x?$ is not observable under any program run of p. Therefore, we will remove such non-observable, synchronous traces from consideration. Before we consider such refinements of our stuckness criterion, we develop static methods to check for stuckness.

5.2 Static Checking of Stuckness

To check for stuckness, we apply an automata-based method where we first translate the forkable behavior into an equivalent finite state machine (FSA) and then analyze the resulting FSA for stuckness. The FSA construction method for forkable behaviors follows the approach described in our prior work [21] where we build a FSA based on Brzozowski's derivative construction method [2].

We say that a forkable behavior r is *well-behaved* if there is no fork inside a Kleene star expression. The restriction to well-behaved behaviors guarantees finiteness (i.e., termination) of the automaton construction.

Proposition 6 (Well-Behaved Forkable FSA [21]). *Let r be a well-behaved behavior. Then, we can construct an $\mathcal{FSA}(r)$ where the alphabet coincides with the alphabet of r and states can be connected to behaviors such that (1) r is the initial state and (2) for each non-empty trace $T = \alpha_1 \cdot \ldots \cdot \alpha_n$ we find a path $r = r_0 \overset{\alpha_1}{\to} r_1 \ldots r_{n-1} \overset{\alpha_n}{\to} r_n$ in $\mathcal{FSA}(r)$ such that $T \backslash L(r) = L(r_n)$.*

The kind of FSA obtained by our method [21] guarantees that all matching derivations (Definition 14) which yield a non-trivial trace can also be observed in the FSA.

Proposition 7 (FSA covers Matching). *Let r be a well-behaved behavior such that $r \stackrel{T}{\Longrightarrow} \{\!\{s_1, ..., s_m\}\!\}$ for some non-empty trace $T = \alpha_1 \cdot ... \cdot \alpha_n$. Then, there exists a path $r = r_0 \stackrel{\alpha_1}{\rightarrow} r_1 ... r_{n-1} \stackrel{\alpha_n}{\rightarrow} r_n$ in $\mathcal{FSA}(r)$ such that $L(s_1) \| ... \| L(s_m) \subseteq L(r_n)$.*

Based on above, we conclude that stuckness of a behavior implies that the FSA is *stuck* as well. That is, we encounter a non-synchronous path.

Proposition 8. *Let r be a well-behaved behavior such that r is stuck. Then, there exists a path $r = r_0 \stackrel{\alpha_1}{\rightarrow} r_1 ... r_{n-1} \stackrel{\alpha_n}{\rightarrow} r_n$ in $\mathcal{FSA}(r)$ such that $L(r_i) \neq \{\}$ for $i = 1, ..., n$ and $\alpha_1 \cdot ... \cdot \alpha_n$ is a non-synchronous trace.*

Proposition 9. *Let r be a well-behaved behavior such that $\mathcal{FSA}(r)$ is stuck. Then, any non-synchronous path that exhibits stuckness can be reduced to a non-synchronous path where a state appears at most twice along that path*

Based on the above, it suffices to consider minimal paths. We obtain these paths as follow. We perform a breadth-first traversal of the $\mathcal{FSA}(r)$ starting with the initial state r to build up all paths which satisfy the following criterion: (1) We must reach a final state, and (2) a state may appear at most twice along a path. It is clear that the set of all such paths is finite and their length is finite. If among these paths we find a non-synchronous path, then the $\mathcal{FSA}(r)$ is stuck.

Proposition 10. *Let r be a well-behaved behavior. Then, it is decidable if the $\mathcal{FSA}(r)$ is stuck.*

Based on the above, we obtain a simple and straightforward to implement method for static checking of deadlocks in Mini-Go programs. Any non-synchronous path indicates a potential deadlock and due to the symbolic nature of our approach, erroneous paths can be traced back to the program text for debugging purposes.

5.3 Eliminating False Positives

Naive application of the criterion developed in the previous section yields many false positives. In our setting, a false positive is a non-synchronous path that is present in the automaton $\mathcal{FSA}(r)$, but which cannot be observed in any program run of p. This section introduces an optimization to eliminate many false positives. This optimization is integrated in our implementation.

For example, consider the forkable behavior $r = \mathit{Fork}(x! \cdot y!) \cdot x? \cdot y?$ resulting from the program $p = \textsf{go}\ (x^{\mathsf{s}} \leftarrow \mathit{True}; y^{\mathsf{s}} \leftarrow \mathit{False}); z \leftarrow x^{\mathbf{r}}; z \leftarrow y^{\mathbf{r}}$. Based on our FSA construction method, we discover the non-synchronous path $r \xrightarrow{x! \cdot y! \cdot x? \cdot y?} \varepsilon$ where ε denotes some accepting state. However, just by looking at this simple program it is easy to see that there is no deadlock. There are two threads and for each thread, each program statement synchronizes with the program statement of the other thread at the respective position. That is, $\langle \_, \{\!\{p\}\!\} \rangle \xrightarrow{x! \cdot x? \cdot y! \cdot y?} \langle \_, \{\!\{\}\!\} \rangle$.

So, a possible criterion to 'eliminate' a non-synchronous path from consideration seems to be to check if there exists an alternative synchronous permutation of this path. There are two cases where we need to be careful: (1) Conditional statements and (2) inter-thread synchronous paths.

Conditional statements Let us consider the first case. For example, consider the following variant of our example:

$$r = \mathit{Fork}(x! \cdot y!) \cdot (x? \cdot y? + y? \cdot x?)$$

$$\begin{aligned} p = {} & \mathsf{go}\ (x^{\mathsf{s}} \leftarrow \mathit{True}; y^{\mathsf{s}} \leftarrow \mathit{False}); \\ & \mathsf{if}\ \mathit{True}\ \mathsf{then}\ (z \leftarrow x^{\mathbf{r}}; z \leftarrow y^{\mathbf{r}})\ \mathsf{else}\ (z \leftarrow y^{\mathbf{r}}; z \leftarrow x^{\mathbf{r}}) \end{aligned}$$

By examining the program text, we see that there is no deadlock as the program will always choose the 'if' branch. As our (static) analysis conservatively assumes that both branches may be taken, we can only use a synchronous permutation to eliminate a non-synchronous path if we do not apply any conditional statements along this path. In terms of the matching relation from Definition 14, we can characterize the absence of conditional statements if none of the rules (L), (R), (K_n) and (K_0) has been applied.

Inter-thread synchronous paths The second case concerns synchronization within the same thread. Consider yet another variant of our example:

$$r = \mathit{Fork}(x! \cdot x?) \cdot y! \cdot y?$$

$$p = \mathsf{go}\ (x^{\mathsf{s}} \leftarrow \mathit{True}; z \leftarrow x^{\mathbf{r}}); y^{\mathsf{s}} \leftarrow \mathit{False}; z \leftarrow y^{\mathbf{r}}$$

The above program will deadlock. However, in terms of the abstraction, i.e. forkable behavior, we find that for the non-synchronous path there exists a synchronous permutation which does not make use of any of the conditional matching rules, e.g. $r \xrightarrow{x! \cdot x? \cdot y! \cdot y?} \{\!\{\}\!\}$. This is clearly not a valid alternative as for example $x!$ and $x?$ result from the same thread.

To identify the second case, we assume that receive/send symbols α in a trace carry a distinct thread identifier (ID). We can access the thread ID of each symbol α via some operator $\sharp(\cdot)$. Under our assumed restrictions (i.e., no forks inside of loops, which is no go inside a while loop) it is straightforward to obtain this information precisely.

We refine the approximation of a program's communication behavior in terms of a forkable behavior such that communications carry additionally the thread identification number. Recall that we exclude programs where there is a go statement within a while loop. Thus, the number of threads is statically known and thread IDs can be attached to communication symbols via a simple extension

$p \overset{i}{\leadsto} r$ of the relation $p \leadsto r$. The additional component i represents the identification number of the current thread. We start with $p \overset{0}{\leadsto} r$ where 0 represents the main thread. We write symbol $x!^i$ to denote a transmission over channel x which takes place in thread i. Similarly, symbol $x?^i$ denotes reception over channel x in thread i. For each symbol, we can access the thread identification number via operator $\sharp(\cdot)$ where $\sharp(x!^i) = i$ and $\sharp(x?^i) = i$.

The necessary adjustments to Definition 11 are as follows.

$$\mathsf{skip} \leadsto \varepsilon i \qquad \frac{p \leadsto ri \quad q \leadsto si}{\mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ q \leadsto r + si} \qquad \frac{p \leadsto ri}{\mathsf{while}\ b\ \mathsf{do}\ p \leadsto r^* i}$$

$$\frac{p \leadsto ri \quad q \leadsto si}{p; q \leadsto r \cdot si} \qquad x \leftarrow y^{\mathsf{r}} \leadsto y?^i i \qquad y^{\mathsf{s}} \leftarrow b \leadsto y!^i i$$

$$\frac{e_i \leadsto r_i i \quad p_i \leadsto s_i i \quad \text{for } i \in I}{\mathsf{select}\ [e_i \Rightarrow p_i]_{i \in I} \leadsto \sum_{i \in I}\ r_i \cdot s_i} i \qquad \frac{p \leadsto ri + 1}{\mathsf{go}\ p \leadsto \mathit{Fork}(r)} i$$

We summarize our observations.

Definition 16 (Concurrent Synchronous Permutation). *Let T_1 and T_2 be two traces. We say that T_1 is a* concurrent synchronous permutation *of T_2 iff (1) T_1 is a permutation of the symbols in T_2, (2) T_1 is a synchronous trace of the form $\alpha_1 \cdot \overline{\alpha_1} \cdot ... \cdot \alpha_n \cdot \overline{\alpha_n}$ where $\sharp(\alpha_i) \neq \sharp(\overline{\alpha_i})$ for $i = 1, ..., n$.*

Proposition 11 (Elimination via Concurrent Synchronous Permutation). *Let p be a program. Let r be a well-behaved behavior such that $p \leadsto r$. For any non-synchronous path T in $\mathcal{FSA}(r)$, there exists a synchronous path T_1, a non-synchronous path T_2 and a concurrent synchronous permutation T_3 of T_2 such that $r \overset{T_1}{\Longrightarrow} \{\!\{r_1, ..., r_m\}\!\}$, $\{\!\{r_1, ..., r_m\}\!\} \overset{T_2}{\Longrightarrow} \{\!\{\}\!\}$, and $\{\!\{r_1, ..., r_m\}\!\} \overset{T_3}{\Longrightarrow} \{\!\{\}\!\}$ where in the last match derivation none of the rules (L), (R), (K_n) and (K_0) have been applied. Then, program p is not stuck.*

The 'elimination' conditions in the above proposition can be directly checked in terms of the $\mathcal{FSA}(r)$. Transitions can be connected to matching rules. This follows from the derivative-based FSA construction. Hence, for each non-synchronous path in $\mathcal{FSA}(r)$ we can check for a synchronous alternative. We simply consider all (well-formed) concurrent synchronous permutations and verify that there is a path which does not involve conditional transitions.

A further source for eliminating false positives is to distinguish among nondeterminism resulting from selective communication and nondeterminism due to conditional statements. For example, the following programs yield the same (slightly simplified) abstraction

$$r = \mathit{Fork}(x!) \cdot (x? + y?)$$

$$p_1 = \mathsf{go}\ x^{\mathsf{s}} \leftarrow \mathit{True}; \mathsf{select}\ [z \leftarrow x^{\mathbf{r}} \Rightarrow \mathsf{skip}, z \leftarrow y^{\mathbf{r}} \Rightarrow \mathsf{skip}]$$

$$p_2 = \mathsf{go}\ x^{\mathsf{s}} \leftarrow \mathit{True}; \mathsf{if} \mathit{True}\ \mathsf{then}\ z \leftarrow x^{\mathbf{r}} \Rightarrow\ \mathsf{else}\ z \leftarrow y^{\mathbf{r}}$$

It is easy to see that there is a non-synchronous path, e.g. $r \xrightarrow{x! \cdot y?} \varepsilon$. Hence, we indicate that the program from which this forkable behavior resulted may get stuck. In case of p_1 this represents a false positive because the non-synchronous path will not be selected.

The solution is to distinguish between both types of nondeterminism by abstracting the behavior of select via some new operator $\oplus$ instead of $+$. We omit the straightforward extensions to Definition 11. In terms of the matching relation, $+$ and $\oplus$ behave the same. The difference is that for $\oplus$ certain non-synchronous behavior can be safely eliminated.

Briefly, suppose we encounter a non-synchronous path where the (non-synchronous) issue can be reduced to $\{\!\{\alpha_1 \oplus ... \oplus \alpha_n, \beta_1 \oplus ... \oplus \beta_m\}\!\} \xRightarrow{\alpha_i \cdot \beta_j} \{\!\{\}\!\}$ for some $i \in \{1, ..., n\}$ and $j \in \{1, ..., m\}$ where $\alpha_i \cdot \beta_j$ is non-synchronous. Suppose there exists $l \in \{1, ..., n\}$ and $k \in \{1, ..., m\}$ such that $\{\!\{\alpha_1 \oplus ... \oplus \alpha_n, \beta_1 \oplus ... \oplus \beta_m\}\!\} \xRightarrow{\alpha_l \cdot \beta_k} \{\!\{\}\!\}$ and $\alpha_l \cdot \beta_k$ is synchronous. Then, we can eliminate this non-synchronous path. The reason why this elimination step is safe is due to rule (Sync) in Definition 7. This rule guarantees that we will always synchronize if possible. As in case of the earlier 'elimination' approach, we can directly check the $\mathcal{FSA}(r)$ by appropriately marking transitions due to $\oplus$.

Further note that to be a safe elimination method, we only consider select statements where case bodies are trivial, i.e. equal skip. Hence, we find α_i and β_j in the above instead of arbitrary behaviors. Otherwise, this elimination step may not be safe. For example, consider

$$r = \mathit{Fork}(x! \cdot y!) \cdot (x? \cdot y? \oplus x? \cdot x?)$$

$$p = \mathsf{go}\ (x^{\mathsf{s}} \leftarrow \mathit{True}; y^{\mathsf{s}} \leftarrow \mathit{False}); \mathsf{select}\ [z \leftarrow x^{\mathbf{r}} \Rightarrow z \leftarrow y^{\mathbf{r}}, z \leftarrow x^{\mathbf{r}} \Rightarrow z \leftarrow x^{\mathbf{r}}]$$

Due to the non-trivial case body $z \leftarrow x^{\mathbf{r}}$ we encounter a non-synchronous path which cannot be eliminated.

6 Experimental Results

6.1 Implementation

We have built a prototype of a tool that implements our approach, referred to as gopherlyzer [8]. Our analysis operates on the Go source language where we make use of the oracle tool [7] to obtain (alias) information to identify matching channel names. We currently assume that all channels are statically known and all functions can be inlined. The implementation supports select with default cases, something which we left out in the formal description for brevity. Each default case is treated as an empty trace ε.

Go's API also contains a close operation for channels. Receiving from a closed channel returns a default value whereas sending produces an error. An integration of this feature in our current implementation is not too difficult but left out for the time being. The technical report provides further details.

Gopherlyzer generates the FSA 'on-the-fly' while processing the program. It stops immediately when encountering a deadlock. We also aggressively apply the 'elimination' methods described in Section 5.3 to reduce the size of the FSA. When encountering a deadlock, the tool reports a minimal trace to highlight the issue. We can also identify stuck threads by checking if a non-synchronous communication pattern arises for this thread. Thus, we can identify situations where the main thread terminates but some local thread is stuck. The reported trace could also be used to replay the synchronization steps that lead to the deadlock. We plan to integrate extended debugging support in future versions of our tool.

6.2 Examples

For experimentation, we consider the examples `deadlock`, `fanin`, and `primesieve` from Ng and Yoshida [18]. To make `primesieve` amenable to our tool, we moved the dynamic creation of channels outside of the (bounded) for-loop. Ng and Yoshida consider two further examples: `fanin-alt` and `htcat`. We omit `fanin-alt` because our current implementation does not support closing of channels. To deal with `htcat` we need to extend our frontend to support certain syntactic cases. In addition, we consider the examples `sel` and `selFixed` from Section 2 as well as `philo` which is a simplified implementation of the dining philosophers problem where we assume that all forks are placed in the middle of the table. As in the original version, each philosopher requires two forks for eating. All examples can be found in the gopherlyzer repository [8].

6.3 Experimental results

Comparison with dingo-hunter [18] For each tool we report analysis results and the overall time used to carry out the analysis. Table 1 summarizes our results which were carried out on some commodity hardware (Intel i7 3770 @ 3.6GHz, 16 GB RAM, Linux Mint 17.3).

Our timings for dingo-hunter are similar to the reported results [18], but it takes significantly longer to analyze our variant of `primesieve`, where we have unrolled the loop. There is also significant difference between `sel` and `selFixed` by an order of magnitude. A closer inspection shows that the communicating finite state machines (CFSMs) generated by dingo-hunter can grow dramatically in size with the number of threads and channels used.

Example	LoC	Channels	Goroutines	Select	Deadlock	dingo-hunter		gopherlyzer	
						result	time	result	time
deadlock	34	2	5	0	true	true	155	true	21
fanin	37	3	4	1	false	false	107	false	29
primesieve	57	4	5	0	true	true	8000	true	34
philo	34	1	4	0	true	true	480	true	31
sel	25	4	4	0	true	true	860	true	24
selFixed	25	2	2	2	false	false	85	false	30

Table 1. Experimental results. All times are reported in ms

The analysis time for our tool is always significantly faster (between 3x and 235x with a geometric mean of 17x). Judging from the dingo-hunter paper, the tool requires several transformation steps to carry out the analysis, which seems rather time consuming. In contrast, our analysis requires a single pass over the forkable behavior where we incrementally build up the FSA to search for non-synchronous paths.

Both tools report the same analysis results. We yet need to conduct a more detailed investigation but it seems that both approaches are roughly similar in terms of expressive power. However, there are some corner cases where our approach appears to be superior.

Consider the following (contrived) examples in Mini-Go notation: $(\mathsf{go}\ x^{\mathsf{s}} \leftarrow \mathit{True}); y \leftarrow x^{\mathbf{r}}$ and $y \leftarrow x^{\mathbf{r}}; (\mathsf{go}\ x^{\mathsf{s}} \leftarrow \mathit{True})$. Our tool reports that the first example is deadlock-free but the second example may have a deadlock. The second example is out of scope of the dingo-hunter because it requires all goroutines to be created before any communication takes place. Presently, dingo-hunter does not seem to check this restriction because it reports the second example as deadlock-free.

Our approximation with forkable behaviors imposes no such restrictions. The first example yields $\mathit{Fork}(x!) \cdot x?$ whereas the second example yields $x? \cdot \mathit{Fork}(x!)$. Thus, our tool is able to detect the deadlock in case of the second example.

Comparison with Kobayashi [13] We conduct a comparison with the TyPiCal tool [11] which implements Kobayashi's deadlock analysis [13]. As the source language is based on the π-calculus, we manually translated the Go examples to the syntax supported by TyPiCal's Web Demo Interface available from Kobayashi's homepage. The translated examples can be found in the gopherlyzer repository [8].

To the best of our knowledge, TyPiCal does *not* support a form of selective communication. Hence, we need to introduce some helper threads which results in an overapproximation of the original Go program's behavior and potentially introduces a deadlock. Recall the discussion in Section 2.

For programs not making use of selective communication (and closing of channels; another feature not supported by TyPiCal), we obtain the same analysis results. Analysis times seem comparable to our tool. The exception being

```
// Go program
x := make(chan int)
y := make(chan int)
go func() {
  x <- 42
  v1 := <-y          // P1
  x <- 43
  v2 := <-y }()
v3 := <-x
v4 := <-x            // P2
v5 := <-x
y <- 42
```

```
/*** TyPiCal input ***/
new x in
new y in
  x!42.y?v1.x!43.y?v2
| x?v3.x?v4.x?v5.y!42

/*** TyPiCal output ***/
new x in
new y in
  x!!42.y?v1.x!!43.y?v2
| x??v3.x?v4.x?v5.y!!42
```

Analysis report: $x!^1 \cdot x?^2 \cdot \underline{y?^1 \cdot x?^2} \ldots$

Fig. 1. Analysis Report: Gopherlyzer versus TyPiCal

the `primesieve` example which cannot be analyzed within the resource limits imposed by TyPiCal's Web Demo Interface which we used in the experiments. Like our tool, TyPiCal properly maintains the order among threads. Recall the example $y \leftarrow x^{\mathbf{r}}; (\mathtt{go}\ x^{\mathbf{s}} \leftarrow \mathit{True})$ from above.

Finally, gopherlyzer reports the analysis result in a different way than TyPiCal. The left side of Figure 1 contains a simple Go program and the right side its translation to TyPiCal's source language. TyPiCal reports that the program is unsafe and might deadlock. Annotations `?` and `!` denote potentially stuck receive and send operations whereas `??` and `!!` indicate that the operations might succeed. The trace-based analysis (on the left) yields a non-synchronous trace from which we can easily pinpoint the position(s) in the program which are likely to be responsible. In the example, the underlined events are connected to program locations `P1` and `P2`.

7 Conclusion

We have introduced a novel trace-based static deadlock detection method and built a prototype tool to analyze Go programs. Our experiments show that our approach yields good results and its efficiency compares favorably with existing tools of similar scope.

In future work, we intend to lift some of the restrictions of the current approach, for example, supporting programs with dynamically generated goroutines. Such an extension may result in a loss of decidability of our static analysis. Hence, we consider mixing our static analysis with some dynamic methods.

Acknowledgments

We thank the APLAS'16 reviewers for their constructive feedback.

References

1. C. Boyapati, R. Lee, and M. C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proc. of OOPSLA'02*, pages 211–230. ACM, 2002.
2. J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.
3. M. Christakis and K. F. Sagonas. Detection of asynchronous message passing errors using static analysis. In *Proc. of PADL'11*, volume 6539 of *LNCS*, pages 5–18. Springer, 2011.
4. C. Colby. Analyzing the communication topology of concurrent programs. In *Proc. of PEPM'95*, pages 202–213. ACM, 1995.
5. D. R. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proc. of SOSP'03*, pages 237–252. ACM, 2003.
6. The Go programming language. https://golang.org/.
7. oracle: a tool for answering questions about go source code. https://godoc.org/golang.org/x/tools/cmd/oracle.
8. Gopherlyzer: Trace-based deadlock detection for mini-go. https://github.com/KaiSta/gopherlyzer.
9. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, Aug. 1978.
10. S. T. Huang. A distributed deadlock detection algorithm for CSP-like communication. *ACM Trans. Program. Lang. Syst.*, 12(1):102–122, Jan. 1990.
11. N. Kobayashi. TyPiCal: Type-based static analyzer for the Pi-Calculus. `http://www-kb.is.s.u-tokyo.ac.jp/~koba/typical/`.
12. N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Inf.*, 42(4-5):291–347, 2005.
13. N. Kobayashi. A new type system for deadlock-free processes. In *Proc. of CONCUR'06*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006.
14. P. B. Ladkin and B. B. Simons. *Static Deadlock Analysis for CSP-Type Communications*, pages 89–102. Springer US, Boston, MA, 1995.
15. M. Martel and M. Gengler. Communication topology analysis for concurrent programs. In *Proc. of SPIN'00*, volume 1885 of *LNCS*, pages 265–286. Springer, 2000.
16. N. Mercouroff. An algorithm for analyzing communicating processes. In *Proc. of MFPS'91*, volume 598 of *LNCS*, pages 312–325. Springer, 1991.
17. R. Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, New York, NY, USA, 1999.
18. N. Ng and N. Yoshida. Static deadlock detection for concurrent go by global session graph synthesis. In *Proc. of CC'16*, pages 174–184. ACM, 2016.
19. H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *Proc. of POPL'94*, pages 84–97. ACM Press, Jan. 1994.
20. J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, New York, NY, USA, 1999.
21. M. Sulzmann and P. Thiemann. Forkable regular expressions. In *Proc. of LATA'16*, volume 9618 of *LNCS*, pages 194–206. Springer, 2016.
22. A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *Proc. of ECOOP'05*, volume 3586 of *LNCS*, pages 602–629. Springer, 2005.

Proofs and further details concerning select with default and closing of channels

A Proofs

A.1 Proof of Proposition 2

Proposition 12. *Let $r_1,...,r_m$, $s_1,...,s_n$ be forkable behaviors and T be a trace such that $\{\!\{r_1,\ldots,r_m\}\!\} \stackrel{T}{\Rightarrow} \{\!\{s_1,\ldots,s_n\}\!\}$. Then, we find that $L(s_1)\|...\|L(s_n) \subseteq T\backslash(L(r_1)\|...\|L(r_m))$.*

Proof. By induction on the derivation. We consider some of the cases.

Case (S1):

$$\frac{r \stackrel{T}{\Rightarrow} s}{\{\!\{r,r_1,\ldots,r_n\}\!\} \stackrel{T}{\Rightarrow} \{\!\{s,r_1,\ldots,r_n\}\!\}}$$

By Proposition 1, $L(s) \subseteq T\backslash L(r)$. We exploit the following facts: $\alpha\backslash L(\alpha\cdot r) = L(r)$ and $\alpha\backslash(L_1\|L_2) = ((\alpha\backslash L_1)\|L_2) \cup (L_1\|(\alpha\backslash L_2))$. The desired result follows immediately.

Case (F): By assumption $\{\!\{\mathit{Fork}(r),r_1,\ldots,r_n\}\!\} \stackrel{\varepsilon}{\Rightarrow} \{\!\{r,r_1,\ldots,r_n\}\!\}$. We have that $L(\mathit{Fork}(r)) = L(r)\|\{\varepsilon\} = L(r)$. Thus, the desired result follows immediately. ◻

A.2 Proof of Proposition 3

Proposition 13. *If $S \vdash p \Rightarrow q$ and $p \leadsto r$ then $r \Rightarrow s$ for some s where $q \leadsto s$.*

Proof. By induction.

Case (If-T):

$$\frac{S \vdash b \Downarrow \mathit{True}}{S \vdash \mathsf{if}\ b\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 \Rightarrow p_1}$$

By assumption $\mathsf{if}\ b\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 \leadsto r_1 + r_2$ for some r_1 and r_2 where $p_1 \leadsto r_1$ and $p_2 \leadsto r_2$. Via rule (L) we find that $r_1 + r_2 \Rightarrow r_1$ and we are done.

Case (If-F): Similar to the above.

Case (While-F):

$$\frac{S \vdash b \Downarrow \mathit{False}}{S \vdash \mathsf{while}\ b\ \mathsf{do}\ p \Rightarrow \mathsf{skip}}$$

By assumption $\mathsf{while}\ b\ \mathsf{do}\ p \leadsto r^*$ for some r. Via rule (K_0) we find that $r^* \Rightarrow \varepsilon$. By definition $\mathsf{skip} \leadsto \varepsilon$. Thus, we are done.

Case (While-T):

$$\frac{S \vdash b \Downarrow \mathit{True}}{S \vdash \mathsf{while}\ b\ \mathsf{do}\ p \Rightarrow p; \mathsf{while}\ b\ \mathsf{do}\ p}$$

By assumption $\mathsf{while}\ b\ \mathsf{do}\ p \rightsquigarrow r^*$ for some r where $p \rightsquigarrow r$. Via rule (K_n) we find that $r^* \Rightarrow r \cdot r^*$. Based on our assumption we find that $p; \mathsf{while}\ b\ \mathsf{do}\ p \rightsquigarrow r \cdot r^*$ and thus we are done.

Case (Skip): $S \vdash \mathsf{skip}; p \Rightarrow p$. By assumption $\mathsf{skip}; p \rightsquigarrow \varepsilon \cdot r$. Via rule (A1) we conclude that $\varepsilon \cdot r \Rightarrow r$ and are done.

Case (Reduce):

$$\frac{S \vdash p \Rightarrow p'}{S \vdash p; p'' \Rightarrow p'; p''}$$

By assumption $p; p'' \rightsquigarrow r \cdot r''$ where $p \rightsquigarrow r$ and $p'' \rightsquigarrow r''$ for some r and r''. By induction $r \Rightarrow r'$ for some r' where $p' \rightsquigarrow r'$. Via rule (A2) we obtain $r \cdot r'' \Rightarrow r' \cdot r''$ and we are done again.

Case (Assoc): $S \vdash (p_1; p_2); p_3 \Rightarrow p_1; (p_2; p_3)$. Follows via rule (A3). □

A.3 Proof of Proposition 4

Proposition 14. *If $\langle S, \{\!\{p_1, ..., p_m\}\!\}\rangle \stackrel{T}{\Rightarrow} \langle S', \{\!\{q_1, ..., q_n\}\!\}\rangle$ and $p_i \rightsquigarrow r_i$ for $i = 1, ..., m$ then $\{\!\{r_1, ..., r_m\}\!\} \stackrel{T}{\Rightarrow} \{\!\{s_1, ..., s_n\}\!\}$ where $q_j \rightsquigarrow s_j$ for $j = 1, .., n$.*

Proof. By induction.

Case (Fork):

$$\langle S, \{\!\{\mathsf{go}\ p_1 \mathbin{⨟} q_1, p_2, ..., p_n\}\!\}\rangle \stackrel{\varepsilon}{\Rightarrow} \langle S, \{\!\{p_1, q_1, p_2, ..., p_n\}\!\}\rangle$$

We assume $q_1 \neq \mathsf{skip}$. By assumption $\mathsf{go}\ p_1 \mathbin{⨟} q_1 \rightsquigarrow r$ for some r. Hence, $r = \mathit{Fork}(r_1) \cdot s_1$ where $p_1 \rightsquigarrow r_1$ and $q_1 \rightsquigarrow s_1$. We further assume $p_i \rightsquigarrow r_i$ for $i = 2...n$. Then, we find via rule (F2) $\{\!\{\mathit{Fork}(r_1) \cdot s_1, r_2, ..., r_n\}\!\} \Rightarrow \{\!\{r_1, s_1, r_2, ..., r_n\}\!\}$ and we are done. For $q_1 = \mathsf{skip}$ the reasoning is similar.

Case (Step):

$$\frac{S \vdash p_1 \Rightarrow p_1'}{\langle S, \{\!\{p_1, ..., p_n\}\!\}\rangle \stackrel{\varepsilon}{\Rightarrow} \langle S, \{\!\{p_1', ..., p_n\}\!\}\rangle}$$

By Proposition 3 we find that $r_1 \Rightarrow s_1$ where $p_1' \rightsquigarrow s_1$. Via rule (S1) we can conclude that $\{\!\{r_1, ..., r_n\}\!\} \stackrel{\varepsilon}{\Rightarrow} \{\!\{s_1, r_2, ..., r_n\}\!\}$ and we are done.

Case (Stop): Via rule (S2).

Case (Closure): By induction and application of rule (C).

□

A.4 Proof of Proposition 5

We require some auxiliary statements which both can be verified by some straightforward induction.

The language denotation obtained is never empty.

Proposition 15. *Let p be a program and r be a forkable behavior such that $p \rightsquigarrow r$. Then, we find that $L(r) \neq \{\}$.*

A non-empty language can always be matched against some trace.

Proposition 16. *Let r be a forkable behavior such that $L(r) \neq \{\}$. Then, we find that $r \stackrel{T}{\Longrightarrow} \varepsilon$ for some trace T.*

Proposition 17. *Let p be a stuck program and r be a forkable behavior such that $p \rightsquigarrow r$. Then, r is stuck.*

Proof. By assumption we find $\langle \_, \{\!\{p\}\!\} \rangle \stackrel{T_s}{\Longrightarrow} \langle \_, \{\!\{p_1, ..., p_n\}\!\} \rangle$ where $n > 1$ and each p_i starts with a communication primitive or a select statement. For brevity, we ignore the state component which is abbreviated by $\_$. By construction, T_s is a synchronous trace.

By Proposition 4 we find $r \stackrel{T_s}{\Longrightarrow} \{\!\{r_1, ..., r_n\}\!\}$ where $p_i \rightsquigarrow r_i$ for $i = 1...n$.

By assumption none of the p_i can be reduced further. Recall that $n > 1$. Hence, we must be able to further reduce at least two of the r_i's such that we obtain a non-synchronous trace. For example, $r \xrightarrow{T_s \cdot \alpha \cdot \beta} \{\!\{r'_1, r'_2, r_3, ..., r_n\}\!\}$ where $\overline{\alpha} \neq \beta$. Based on Propositions 15 and 16 we can argue that $\{\!\{r'_1, r'_2, r_3, ..., r_n\}\!\}$ can be further reduced. Hence, $r \xrightarrow{T_s \cdot \alpha \cdot \beta \cdot T} \varepsilon$ for some T. The overall trace $T_s \cdot \alpha \cdot \beta \cdot T$ is non-synchronous. Thus, we can conclude that r is stuck. □

A.5 Proof of Proposition 7

Proposition 18 (FSA covers Matching). *Let r be a well-behaved behavior such that $r \stackrel{T}{\Longrightarrow} \{\!\{s_1, ..., s_m\}\!\}$ for some non-empty trace $T = \alpha_1 \cdot ... \cdot \alpha_n$. Then, there exists a path $r = r_0 \stackrel{\alpha_1}{\to} r_1 ... r_{n-1} \stackrel{\alpha_n}{\to} r_n$ in $\mathcal{FSA}(r)$ such that $L(s_1) \| ... \| L(s_m) \subseteq L(r_n)$.*

Proof. By Proposition 2 we have that $L(s_1) \| ... \| L(s_m) \subseteq T \backslash L(r)$. By property (2) for $\mathcal{FSA}(r)$ we find there exists a $r = r_0 \stackrel{\alpha_1}{\to} r_1 ... r_{n-1} \stackrel{\alpha_n}{\to} r_n$ in $\mathcal{FSA}(r)$ such that $T \backslash L(r) = L(r_n)$. From above, we derive that $L(s_1) \| ... \| L(s_m) \subseteq T \backslash L(r) = T \backslash L(r) = L(r_n)$ and we are done.

A.6 Proof of Proposition 9

Proposition 19. *Let r be a well-behaved behavior such that $\mathcal{FSA}(r)$ is stuck. Then, any non-synchronous path that exhibits stuckness can be reduced to a non-synchronous path where a state appears at most twice along that path*

Proof. By assumption the $\mathcal{FSA}(r)$ is stuck. We need to verify that for each non-synchronous path there exists a non-synchronous, minimal path. By minimal we mean that a state appears at most twice along that path.

W.l.o.g., we assume the following

$$r \stackrel{w_1}{\to} s \stackrel{w_2}{\to} s \stackrel{w_3}{\to} s \stackrel{w_4}{\to} t$$

where state s is repeated more than twice. There are possible further repetitions within the subpath $s \stackrel{w_4}{\to} t$ but not within the subpath $r \stackrel{w_1}{\to} s$. By assumption

$w_1 \cdot w_2 \cdot w_3 \cdot w_4$ is non-synchronous. To show that we can derive a minimal non-synchronous path, we distinguish among the following cases.

Suppose w_1 and w_4 are synchronous. Hence, either w_2 or w_3 must be non-synchronous. Suppose w_2 is non-synchronous. Then we can 'simplify' the above to the non-synchronous example $r \overset{w_1}{\to} s \overset{w_2}{\to} s \overset{w_4}{\to} t$. A similar reasoning applies if w_3 is non-synchronous.

Suppose w_1 is synchronous and w_4 is non-synchronous. Immediately, we obtain a 'simpler' non-synchronous example $r \overset{w_1}{\to} s \overset{w_4}{\to} t$.

Suppose w_1 is non-synchronous. We consider among the following subcases. Suppose that $w_1 = \alpha$. Suppose that $w_1 \cdot w_2 \cdot w_4$ and $w_1 \cdot w_3 \cdot w_4$ are synchronous (otherwise we are immediately done). Suppose that $w_1 \cdot w_4$ is synchronous. We will show that this leads to a contradiction. From our assumption, we derive that $w_4 = \overline{\alpha} \cdot w_4'$. As we assume that $w_1 \cdot w_2 \cdot w_4$ and $w_1 \cdot w_3 \cdot w_4$ are synchronous, we can conclude that $w_2 = \overline{\alpha} \cdot ... \cdot \alpha$ and $w_3 = \overline{\alpha} \cdot ... \cdot \alpha$. However, this implies that $w_1 \cdot w_2 \cdot w_3 \cdot w_4$ is synchronous which is a contradiction. Hence, either $w_1 \cdot w_2 \cdot w_4$ or $w_1 \cdot w_3 \cdot w_4$ is non-synchronous and therefore the example can be further simplified.

Suppose that w_1 contains more than two symbols, e.g. $w_1 = w_1' \cdot \alpha$. If w_1' is a non-synchronous, we can immediately conclude that $r \overset{w_1}{\to} s \overset{w_4}{\to} t$ is a (more minimal) non-synchronous path. Let us assume that w_1' is synchronous. Then, we can proceed like above (case $w_1 = \alpha$) to show that a more minimal, non-synchronous path exists.

These are all cases. Note that $w_1 = \varepsilon$ is covered by the above (cases where w_1 is assumed to be synchronous). $\square$

B Select with default

We show how to support `select` with a default case written $\mathsf{select}\ [e_i \Rightarrow q_i \mid q]_{i \in I}$. The default case will only be executed if none of the other cases apply.

$$\text{(DefaultStep)}\ \frac{\not\exists i \in I, j \in \{2, ..., n\}\ \langle S, \{\!\!\{ q_i, p_i \}\!\!\} \rangle \Rrightarrow \langle \_, \{\!\!\{ q_i', p_i \}\!\!\} \rangle}{\langle S, \{\!\!\{ \mathsf{select}\ [e_i \Rightarrow q_i \mid q]_{i \in I} \,\mathbin{;}\, p_1'', p_2, \ldots, p_n \}\!\!\} \rangle \Rrightarrow \langle S, \{\!\!\{ q \,\mathbin{;}\, p_1'', p_2, \ldots, p_n \}\!\!\} \rangle}$$

In case of the approximation, we represent the default case via ε.

$$\frac{e_i \leadsto r_i \quad q_i \leadsto s_i \quad \text{for } i \in I \quad q \leadsto s}{\mathsf{select}\ [e_i \Rightarrow q_i \mid q]_{i \in I} \leadsto (\sum_{i \in I}\ r_i \cdot s_i) + \varepsilon \cdot s}$$

To eliminate false positives in the presence of select with default we assume that a non-synchronous path can be eliminated/resolved by making use of $\{\!\!\{ \alpha_1 \oplus ... \oplus \alpha_n \oplus \varepsilon, r_2, ..., r_n \}\!\!\} \Rightarrow \{\!\!\{ r_2, ..., r_n \}\!\!\}$.

C Closing Channels

In Go, a channel can be closed which means that no more values which will be sent to it. Any receive operation invoked after a channel is closed will succeed and yield the default value. However, any send operation leads to a 'panic' which we consider as unsafe.

In terms of our analysis framework, we can integrate this additional language feature by simply removing any receive event from the trace which occurs after a channel has been closed. As we generate traces from the FSA and FSA states can be connected to program points, it is straightforward to identify the position in the trace after which all receive events (for that channel) shall be removed.

For example, consider

```
x := make(chan int)
go func() {
  x <- 1 }()
<-x
close(x)
<-x
```

Our analysis reports the non-synchronous trace

$$x!^2 \cdot x?^1 \cdot x?^1$$

where $x?^1$ represents the strictly non-synchronous portion of the trace. By taking into account the feature of closing of channels, this portion can be eliminated. Hence, our analysis reports that the program is safe.

Consider the following variant where the close operation is part of the 'then' branch of a conditional statement. The actual condition is omitted for brevity.

```
x := make(chan int)
go func() {
  x <- 1 }()
<-x
if ... {
  close(x)
}
<-x
```

The tricky bit here is that the channel will only be closed if the if-condition applies. We therefore use a slightly refined language of forkable behaviors to carry out the approximation of the program's communication behavior.

$$\mathit{Fork}(x!^2) \cdot x?^1 \cdot (\mathit{close}(x) + \varepsilon) \cdot x?^1$$

where the new event $\mathit{close}(x)$ represents closing of a channel.

In the resulting FSA, we find the trace

$$x!^2 \cdot x?^1 \cdot \mathit{close}(x) \cdot x?^1$$

As any receive following a close operation will be non-blocking, the program is safe for this specific program run.

There is however another alternative path reported by our analysis which is unsafe

$$x!^2 \cdot x?^1 \cdot \varepsilon \cdot x?^1$$

We include the redundant ε to highlight that the (implicit) 'else' branch was chosen.