Automated Verification of Parametric Channel-Based Process Communication

GEORGIAN-VLAD SAIOC*, Aarhus University, Denmark
JULIEN LANGE, Royal Holloway, University of London, United Kingdom
ANDERS MØLLER, Aarhus University, Denmark

A challenge of writing concurrent message passing programs is ensuring the absence of partial deadlocks, which can cause severe memory leaks in long running systems. Several static analysis techniques have been proposed for automatically detecting partial deadlocks in Go programs. For a large enterprise code base, we found these tools too imprecise to reason about process communication that is parametric, i.e., where the number of channel communication operations or the channel capacities are determined at runtime.

We present a novel approach to automatically verify the absence of partial deadlocks in Go program fragments with such parametric process communication. The key idea is to translate Go fragments to a core language that is sufficiently expressive to represent real-world parametric communication patterns and can be encoded into Dafny programs annotated with postconditions enforcing partial deadlock freedom. In situations where a fragment is partial deadlock free only when the concurrency parameters satisfy certain conditions, a suitable precondition can often be inferred.

Experimental results on a real-world code base containing 583 program fragments that are beyond the reach of existing techniques have shown that the approach can verify the absence of partial deadlocks in 145 cases. For an additional 228 cases, a nontrivial precondition is inferred that the surrounding code must satisfy to ensure partial deadlock freedom.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Theory of computation → Automated reasoning**.

Additional Key Words and Phrases: Go, message passing concurrency, partial deadlocks, static analysis, automated verification, invariant discovery

ACM Reference Format:
Georgian-Vlad Saioc, Julien Lange, and Anders Møller. 2024. Automated Verification of Parametric Channel-Based Process Communication. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 344 (October 2024), 27 pages. https://doi.org/10.1145/3689784

1 Introduction

Message passing concurrency in the style of Hoare's CSP [13] has experienced a resurgence through the Go programming language [37], especially for systems development. Go prominently treats concurrency as a first-class citizen, with its forefront feature being the forking of threads, colloquially known as *goroutines*, simply by prefixing function calls with the **go** keyword. Goroutines may communicate via shared memory or channel-based message passing, where the latter paradigm is favored by the language designers [36]. As with locks in shared memory concurrency, channel

*Also with Uber Technologies, Inc.

Authors' Contact Information: Georgian-Vlad Saioc, Aarhus University, Aarhus, Denmark, gvsaioc@cs.au.dk; Julien Lange, Royal Holloway, University of London, Egham, United Kingdom, Julien.Lange@rhul.ac.uk; Anders Møller, Aarhus University, Aarhus, Denmark, amoeller@cs.au.dk.

ACM 2475-1421/2024/10-ART344
https://doi.org/10.1145/3689784

communication operations may block (e.g., attempting to read from an empty channel). Because of the complex program structure that often exists in multi-threaded code, programming errors sometimes lead to a process being permanently stuck on a channel operation, which is referred to as a *partial deadlock*, also known as a *goroutine leak* or *blocking error*. Partial deadlocks not only cause possibly important actions to be missed; they also often result in severe memory leaks because the involved goroutines are never garbage collected by the Go runtime system [30, 31].

In many cases, communication is localized to a small set of related functions (a *program fragment*), and depends on a set of values (*concurrency parameters*) that dictate the number of communication operations or limit the sizes of channel message queues. The code surrounding the fragment may enforce certain properties on the concurrency parameters which rule out partial deadlocks. The state-of-the-art tools for detecting partial deadlocks in Go are fundamentally unable to reason about parametric fragments i.e., they cannot distinguish erroneous fragments from correct ones, and the extensions needed to overcome these shortcomings are not obvious. GCatch [22] ignores channels with capacities that are not fixed statically and only analyzes loops up to two iterations. Goat [38] has limited support for loops and conservatively over-approximates channel capacities without relating them to other program variables. Gomela [8] tests a finite number of execution paths by instantiating concurrency parameter. We cover related work in more detail in Section 8.

We propose a sound approach to automatically verify the absence of partial deadlocks in Go program fragments that involve parametric process communication. We have designed a core language, VirGo, that is sufficiently expressive to model many parametric fragments, and built the tool Ginger that encodes VirGo programs into Dafny [19] and suggests preconditions over concurrency parameters guaranteeing partial deadlock freedom. Encoded fragments are subsequently verified in Dafny relative to the suggested preconditions. The verification procedure is often able to confirm that the suggested precondition is the weakest precondition that ensures partial deadlock freedom in the original program fragment. The precondition is often trivial (i.e., '`true`'), meaning that the fragment is unconditionally free of partial deadlocks. Whenever the precondition is nontrivial, it can be used for fragment documentation or as requirements that the code surrounding the fragment must satisfy to avoid errors.

The approach is intended to complement existing techniques (which cover many advanced Go features) when fragments rely on parametric communication. Although we focus on Go in this paper, the core of the proposed approach is applicable to other programming languages that support message-passing programming. It is specifically relevant to languages that offer buffered channels with bounded capacity (e.g., Rust and Kotlin).

To summarize, we make the following contributions:

(1) We describe the VirGo language for modeling a family of Go program fragments that implement parametric communication, which other analysis tools cannot reason about.
(2) We show how to automatically verify partial deadlock freedom in VirGo programs by translating them into Dafny. In addition, we present a mechanism for suggesting preconditions, which can be incorporated into the verification procedure to reason about program fragments that are only conditionally correct. We aim for a sound approach: when a VirGo program is deemed free of partial deadlock (possibly conditionally, as expressed by the inferred precondition), then so is the original Go code.
(3) The approach has been implemented in a tool named Ginger, which automatically verifies Go code and suggests preconditions. We evaluate Ginger on a large industrial code base at Uber Technologies, Inc. containing 583 program fragments with parametric channel-based process communication, all of which are beyond the reach of existing Go analysis tools. The

```
1  func GetResult(x int) {
2    c := make(chan T)
3    // Send x messages
4    for i := 0; i < x; i++ {
5      go func() {
6        // Supply result to c
7        c <- work()
8      }
9    }
10   // Retrieve one result
11   ... <-c
12 }
```

```
func GetResult(x int) {
  c := make(chan T, x)

  for i := 0; i < x; i++ {
    go func() {
      c <- work()

    }
  }

  ... <-c
}
```

```
func GetResult(x int) {
  if x <= 0 {
    return // or panic/log
  }
  c := make(chan T, x)
  for i := 0; i < x; i++ {
    go func() {
      c <- work()
    }
  }
  ... <-c
}
```

Fig. 1. Go fragments with parametric communication. Left: Original buggy fragment. Middle: Fragment after fix; channel capacity guarantees all senders unblock. Right: Partial deadlock-free fragment (for all inputs).

experimental results show that the technique can verify the absence of partial deadlocks in 373 fragments, 228 thereof with a nontrivial precondition.

2 Motivating Example

Go is an imperative, concurrent programming language with support for channel-based communication. A channel is a bounded message queue with a capacity specified when the channel is created. A goroutine blocks if attempting to send to a full channel or receive from an empty channel until another goroutine has performed the dual operation. Channels support either synchronous (default) or asynchronous communication (when the capacity is strictly positive). The standard library `sync` provides other concurrency primitives, the most prominent being `Mutex` (standard locks) and `WaitGroup` (barriers managed by a counter). Go also has **`select`** statements, allowing a goroutine to wait on multiple channel operations and non-deterministically choose one once it becomes enabled, and channel **`close`**, which unblocks all receive operations.

Alongside concurrency primitives, Go supports standard imperative features, including conditional statements and loops. It is common to combine these features and channel-based concurrency to implement powerful patterns like the fork-join model [7]. The Go runtime signals global deadlocks by halting the entire system, but partial deadlocks remain invisible. Furthermore, global deadlocks are rare in practice, as real-world systems often involve several independent sets of related goroutines (service instances implemented in our benchmark predominantly feature over 2,000 concurrent goroutines), where only some may deadlock. Partial deadlocks, on the other hand, are common in practice, and generally more difficult to diagnose, with an impact that is not immediately observable. In fact, they are often undiagnosed for long periods of time, surreptitiously impacting overall memory efficiency and wasting server resources, and, in the worst cases, leading to out-of-memory exceptions and system failures [30, 31].

In typical real-world Go programs, it is possible to identify *fragments*, each consisting of a set of related functions that create and use a number of goroutines and channels to collectively achieve a specific purpose. Several existing Go analysis tools, including GCatch [22], Goat [38] and Gomela [8] decompose large programs by identifying such fragments and analyzing them one by one. This work builds on this idea to propose a novel approach to reason specifically about fragments where *concurrency parameters* [8] are statically unknown.

Figure 1 (left) shows a simplified real-life fragment encountered at Uber that implements parametric communication in the form of a single Go function, `GetResult`, that is intended to start a number of worker processes and await the first result. In this fragment, the integer `x` is a concurrency parameter that dictates the number of executed send operations. Following the instantiation of

$$
\begin{aligned}
a &::= \texttt{skip} \mid c! \mid c? \mid a;\ a \mid w.\texttt{Add}(e), \text{ where } c \in \mathbb{C},\ w \in \mathbb{W} \\
V &::= a \mid c = \texttt{chan}\ [e] \mid V;\ V \mid \texttt{go}\ \{\ V\ \} \mid \texttt{return} \mid w = \texttt{WaitGroup} \mid w.\texttt{Wait}() \\
&\quad \mid \texttt{for}\ e\ ..\ e\ \{\ a\ \} \mid \texttt{if}\ e\ \{\ V\ \}\ \texttt{else}\ \{\ V\ \} \\
e &::= x \mid v \mid !e \mid e \oplus e \quad \text{where } \oplus \in \{+,\ -,\ *,\ <,\ ==,\ \wedge\} \text{ and } x \in \mathbb{X} \\
v &::= \texttt{true} \mid \texttt{false} \mid n, \text{ where } n \in \mathbb{Z}
\end{aligned}
$$

Fig. 2. Syntax of VirGo. For the sake of presentation, the formal treatment of the shadowed constructs is omitted from Section 4 but discussed in Section 5.

channel c (line 2), x goroutines are spawned in a loop (line 5), each performing one send operation (line 7). Conversely, the parent executes one receive operation on c (line 11). Since the channel is unbuffered (the channel is created with the default capacity zero), only one send operation will unblock by synchronizing with the parent (line 11). As a result, partial deadlocks may form at any sender beyond the first (if $x > 1$), or at the receiver if no sender is created (if $x \leq 0$). The scope of c is restricted to GetResult, so in the case of a partial deadlock, blocked processes would hold a reference to c indefinitely, preventing garbage collection of both the channel and blocked goroutines. Our approach can automatically identify that precondition $x = 1$ is the weakest precondition that satisfies partial deadlock freedom. Such a strict constraint over x immediately reveals a likely unintended behavior. Identifying this early would have avoided the deployment of buggy code that caused memory leaks in affected services for a year until eventually discovered via other means [30].

Upon its discovery, the bug was patched by a programmer as shown in Figure 1 (middle), by supplying c with capacity x, such that all senders may unblock. Our technique automatically determines that the weakest precondition is $x > 0$, which significantly relaxes the constraints. A partial deadlock remains whenever $x = 0$, and a channel capacity error is triggered whenever $x < 0$. A design decision can be made as to whether callers of GetResult must satisfy this constraint at runtime, or if GetResult must internally ensure that no valuations of x violate correctness, as in Figure 1 (right). Regardless, developer awareness of this implicit constraint is beneficial.

As demonstrated by this example, the correct implementation of parametric communication relies on identifying the proper constraints over concurrency parameters to be satisfied to avoid partial deadlocks. In our approach, these constraints are presented as a precondition over concurrency parameters. When the precondition is 'true', partial deadlock freedom holds unconditionally; in other situations the programmer can use the precondition for documentation or runtime checks.

Section 3 introduces the VirGo core language for expressing parametric communication. For clarity, we present the technical contributions in two parts. In the first part, Section 4 presents the key ideas of the technique by explaining how to construct preconditions and encode VirGo programs in Dafny while focusing on buffered channel operations and for-loops. In the second part, Section 5 gives the technical details for the rest of the features supported by VirGo. We describe how Go program fragments can be translated to VirGo in Section 6, and evaluate the approach in Section 7.

3 A Core Language for Parametric Communication

We introduce VirGo (**V**erifiable **I**ntermediate **R**epresentation for **Go**), a core language modeled on a subset of Go to express fragments implementing parametric communication. Its syntax is shown in Figure 2. A VirGo program is defined over a set of channels ($\mathbb{C}$), waitgroups ($\mathbb{W}$), and concurrency

parameters ($\mathbb{X}$). A program consists of statements (V), which may be a sequence of atomics (a), channel declarations (c = **chan** [e], where e denotes the capacity), process termination (**return**), forking a new process (**go** { V }), conditional statements (**if**), bounded loops (**for**), or waiting on a waitgroup (w.Wait()). Sequences of atomics are broken down to noops (**skip**), send (c!) or receive (c?) channel operations (as discussed below, channel payloads are ignored), or offsetting the counter of a waitgroup (w.Add(e)) with the value of e. The number of iterations of a loop **for** e_1 .. e_2 {...} is bounded such that the number of iterations is $e_2 - e_1$ or 0, whichever is larger. Loop bodies can only contain sequences of atomics. Expressions (e) may be concurrency parameters ($x \in \mathbb{X}$), Boolean or integer constants (v), negation (!e), and common binary arithmetic and Boolean operations and comparisons.

By design, the expressiveness of VirGo is limited so that verification remains tractable, while supporting many real-world programming patterns in Go. Modeling a Go fragment in VirGo involves capturing communication operations and control-flow. Note that all variables in a VirGo program as they appear in expressions, e.g., capacities or loop bounds, are *concurrency parameters*. VirGo is designed to over-approximate the behavior of Go programs and thus support a sound end-to-end verification. Notably, features of a Go programs that cannot be modelled directly by VirGo may be replaced by fresh concurrency parameters (e.g., data received from a channel that is later used in the condition of an if-then-else). For brevity, we omit a formal definition of the VirGo semantics as it closely mirrors the concurrent semantics of Go (see, e.g., [17] for a similar formalization).

We summarize the key design choices below:

(1) *Immutable concurrency parameters*: the values of concurrency parameters may not change at runtime. This is enforced syntactically as only variables in $\mathbb{X}$ are allowed in expressions and VirGo does not support variable assignment.
(2) *Fixed loop bounds*: only finite iteration is supported and loop bounds cannot change during the execution of the program (a consequence of item 1).
(3) *Restricted looping behavior*: the bodies of loops may only contain communication actions. Branching and process spawning are syntactically disallowed. This notably rules out VirGo programs that spawn an arbitrary number of processes (we explain how to partially lift this restriction in Section 6).
(4) *Abstract channel payloads*: the payloads of messages sent over channels are not modeled, in line with many static checkers for Go [8, 11, 17, 18, 22, 27].
(5) *Focus on channels and waitgroups as primitives*: VirGo only supports channels and waitgroups, as they are most affected by parameterized communication. We describe how our implementation deals with mutexes in Section 6.

Note that items 2 and 3 above imply that under a fair process scheduler, each process of a VirGo program eventually successfully terminates or becomes permanently blocked on a communication action. The approach does not have general support for **close** operations and **select** statements. While the former may be supported at the cost of some engineering work, the latter poses a particular problem when they occur in loops as we cannot reasonably predict what branch of the select will fire at each iteration. In fact, non-deterministic **select** statements in loops generally lead to potential deadlocks when used with parameterized channel capacity, and non-parameterized versions are supported by complementary approaches [8, 38]. We describe in Section 6 how more features of the language (e.g., function calls and limited forms of **select**) are supported.

Technical overview. The crux of the approach is to use VirGo as an intermediate representation between a Go program fragment and a Hoare triple describing the conditions under which the fragment is free of partial deadlock. Given a VirGo program V, we generate a triple $\{P\}enc(V)\{Q\}$, where P is a precondition synthesized according to different strategies, $enc(V)$ is a Dafny program

```
{ x = 1 }

c = chan [0];
go {
  for 0 .. x { c! }
};
c?
```

```
{ 1 ≤ x }

c = chan [x];
go {
  for 0 .. x { c! }
};
c?
```

```
{ true }
if x <= 0 { return } else { skip };
c = chan [x];
go {
  for 0 .. x { c! }
};
c?
```

Fig. 3. VirGo translations of the program fragments from Figure 1, annotated with synthesized preconditions (first line).

(incl. invariants) that models the execution of the constituent sequential processes of V interleaved according to an abstract scheduler, and Q is a postcondition describing the termination of all processes. When Dafny determines that $\{P\}enc(V)\{Q\}$ is valid, this implies that V, and its Go counterpart, are free of partial deadlock whenever the precondition P is satisfied.

A Go fragment is identified wrt. the scope of its concurrency primitives (i.e., channels and waitgroups). The translation from Go to VirGo is largely straightforward but includes program transformations that help widen the applicability of the approach. Consider Figure 3 which gives the translation of each Go fragment from Figure 1. Channel creation and send/receive operations are translated as expected (note that we abstract away from the values carried by `c`). In order to obtain a VirGo program, we transform the `for-go` pattern of the Go program into a `go-for` pattern. This is a sound transformation since we do not model message payloads and the goroutines execute only one send action. We give more details on the translation from Go to VirGo in Section 6.

The procedure from V to $enc(V)$ is natural but quite involved as it transforms an imperative program into an automata-like representation. More details are given in Section 4.2. For Dafny to automatically verify $\{P\}enc(V)\{Q\}$, we must annotate $enc(V)$ with loop invariants. We describe the invariant generation procedure in Section 4.3, which also details how we generate a postcondition Q that models the termination of all processes (and therefore the absence of partial deadlocks).

The ideal precondition P is the weakest constraint over the concurrency parameters that guarantees termination of all processes, i.e., it is both *sufficient* (ensuring termination) and *necessary* (a non-gracefully-terminating execution is guaranteed to exist when the precondition is not satisfied). While a generalized process for finding such a precondition is still undiscovered, a good starting point is to optimistically check that fragments are correct for all inputs, i.e., that $\{\texttt{true}\}enc(V)\{Q\}$ holds. However, this may fail. For example, the first program in Figure 3 has partial deadlocks when $x > 1$, so we devise an additional strategy for suggesting preconditions. This strategy is based on the idea that all processes in a VirGo program terminate if the following two criteria are satisfied:

- **C1** The number of receive operations must not exceed the number of send operations (all receives unblock).
- **C2** The number of send operations must not exceed the sum of the capacity of the channel and the number of receive operations (all sends unblock).

In our experience, these criteria are often sufficient to generate a precondition that allows verification of partial deadlock freedom. For the examples in Figure 3, they are applied as follows:

(a) In the left-hand-side program **C1** and **C2** require that we have $1 \leq x$ and $x \leq 1$ respectively.
(b) For the middle program, we have $1 \leq x$ by **C1**, and $x \leq x + 1$ by **C2**. Thus $1 \leq x$ overall.
(c) The reasoning is similar for the right-hand-side program, but no channel operation is reachable if $x \leq 0$, hence we have $x \leq 0 \vee 1 \leq x$, a tautology.

Additionally, we propose a variant that also checks whether the suggested precondition P is the weakest. We present these strategies to discover preconditions in more detail in Section 4.4.

$$
\begin{aligned}
\mathbb{T} \ni T &::= \texttt{nat} \mid \texttt{int} \mid \texttt{bool} \mid \texttt{nat} \to \texttt{nat} \\
M &::= \texttt{method}\ f(\overline{y\colon T})\ \texttt{returns}\ (\overline{y\colon T})\ \overline{\texttt{requires}\ E}\ \overline{\texttt{ensures}\ E}\ \{\ S\ \} \\
\mathbb{S} \ni S &::= \{\} \mid S;\ S \mid \texttt{var}\ y\ :=\ E \mid y\ :=\ E \mid \texttt{return} \mid \texttt{if}\ E\ \{\ S\ \}\ [\texttt{else}\ \{\ S\ \}] \\
&\quad \mid \texttt{match}\ E\ \{\ \overline{\texttt{case}\ n \Rightarrow S}\ \} \mid \texttt{while}\ E\ \overline{\texttt{invariant}\ E}\ \{\ S\ \} \\
\mathbb{E} \ni E &::= e \mid E \implies E \mid E \iff E \mid \texttt{forall}\ y\ ::\ E \mid \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \mid x(E)
\end{aligned}
$$

Fig. 4. CoreDafny syntax.

4 Encoding and Verifying VirGo Programs

To automate the verification while allowing the intermediate representation to be human readable, our Hoare triples take the form of Dafny programs. The encoding consists of three main steps. **Step 1:** we decompose a VirGo program into several sequential processes, each encoded as a counter-automaton-like model [25]. All possible concurrent interleavings of these processes are modeled using an abstract scheduler and a `while`-loop. **Step 2:** we synthesize a postcondition describing the termination of all processes, and an invariant that represents the `while`-loop. **Step 3:** we construct a precondition, following three possible strategies, and ask Dafny to check whether the Hoare-triple holds.

The encoding to Dafny aims at modeling precisely the original VirGo program. Hence when Dafny validates a generated Hoare triple, the partial-deadlock freedom result holds for the source VirGo program; this in turns enables the sound verification of the original Go program.

To simplify the presentation of the approach, we require in this section that all channels are asynchronous (buffer capacity >0), and the shadowed syntactical categories of Figure 2 (i.e., conditional operations and waitgroups) are omitted for now. The formalization of the translation for the whole VirGo is given in Section 5.

4.1 Preliminaries: CoreDafny

Dafny is an imperative programming language that supports the formal specification of programs using pre/postcondition, assertions, and loop invariants. Dafny's ecosystem automatically (when loops are annotated with suitable invariants) checks that a program satisfies its postcondition whenever the precondition holds.

We distill the salient features of Dafny into CoreDafny (Figure 4), a subset of features employed in the encoding. A CoreDafny program consists of a single method declaration M. The method signature specifies its typed parameters and named typed return variables, where types include naturals, integers, Booleans, and functions over naturals. The signature includes *preconditions* (`requires` E) and *postconditions* (`ensures` E). Preconditions are propositions constraining the values of input parameters. Postconditions are propositions over the return variables that must be satisfied once the execution of the method terminates.

CoreDafny statements include the empty statement ($\{\}$), variable declarations (`var` y := E), assignments (y := E), conditional statements with optional `else` branches, and pattern matching over integers. CoreDafny also supports `while`-loops with arbitrary guards, which may be annotated with *invariants*. CoreDafny expressions include the usual logical operator, unary function calls, universal quantifiers (`forall`) and conditionals. Here we assume that VirGo expressions are a subset of CoreDafny expressions so that they do not require translation.

Figure 5 shows the structure of the CoreDafny program the encoding generates. We give more details in the remaining part of this section. The program is given in the form a template that

```
1  method Fragment(Sch: nat → nat, ∀x_i ∈ 𝕏: x_i : T_i) returns (∀π ∈ Π. P(π): int)
2    // Specification
3    requires forall n :: Sch(n) < |Π|
4    requires P
5    ensures Q
6  { // Initialisation
7    var step := 0;
8    P(π_0) := 0;
9    ∀π ∈ Π \ {π_0} : P(π) := -1;
10   ∀c ∈ dom(κ) : var c := 0;
11   ∀(_,_,_,i,e,_,_) ∈ ℒ : var i := e;
12
13   // Execution via interleaving
14   while enabled(V)
15   invariant inv(V) {
16     match Sch(step) {
17       ∀π ∈ Π : case π ⇒ match P(π) {
18         ∀n ∈ dom(Ξ[π]) : case n ⇒ Ξ[π][n]
19       }
20     }
21     step := step + 1;
22   }
23 }
```

Fig. 5. Structure of the CoreDafny encoding for a VirGo program represented as a mapping Ξ via the translation explained in Section 4.2.

combines CoreDafny syntax (in black or purple type-writer font) with meta-syntax (highlighted in yellow). A VirGo program is translated to a CoreDafny method whose input consists of an abstract scheduler Sch and the concurrency parameters $\mathbb{X}$, and its output consists of the final program counters (represented by $\mathsf{P}(\pi)$ for each process identifier $\pi \in \Pi$, where π_0 denotes the main process).

The scheduler Sch is an uninterpreted function that decides which process should be scheduled next. The precondition P (line 4) is generated by the different strategies presented in Section 4.4. The postcondition Q (line 5) is generated automatically and corresponds to the termination of all processes (Section 4.3). It essentially requires that each program counter, $\mathsf{P}(\pi)$, has reached its termination point, $\mathsf{T}(\pi)$, assuming process π has started.

The body of the method consists of two parts: (1) some variable initialization (lines 7–11) which we will describe further below, and (2) a **while**-loop. The **while**-loop models all possible interleavings of the VirGo program and executes as long as at least one process can perform an action ($enabled(V)$ holds). The loop is annotated with an invariant $inv(V)$ (line 15), described in Section 4.3. The body consists of nested **match**-constructs and is generated via a mapping $\Xi\colon \Pi \mapsto N \mapsto \mathbb{S}$ that maps each process identifier to a map from program points N to CoreDafny statements $\mathbb{S}$. We detail how Ξ is built in Section 4.2. The outer **match**-construct selects which *process* is executed next as determined by Sch. For each $\pi \in \Pi$ there is a case whose body contains an inner **match**-construct which chooses the *instruction* to be executed next. This is determined by the value of π's program counter (i.e., variable $\mathsf{P}(\pi)$). There is a case for each program point n in the encoding of process π.

4.2 Step 1: Program Point Decomposition

The first step to transform a VirGo program into CoreDafny is to decompose it into a set of sequential components, each of which becomes a counter-automaton where counters keep track of the states of processes and concurrency primitives (e.g., the number of messages in a channel).

Conventions and notations

$\mathbb{N} \supset \Pi \ni \pi$ process identifiers
$\mathbb{Z} \supset N \ni n$ program points
$\mathbb{E} \ni \mathsf{P}(\pi)$ prog. counter variable
$\mathbb{I} \ni i$ loop variables

$\Phi \ni \phi : N \mapsto \mathbb{S}$ process encoding
$\Xi : \Pi \mapsto \Phi$ process encoding environment
$\kappa : \mathbb{C} \mapsto \mathbb{E}$ channel capacity environment
$\uplus$ disjoint union of maps
$\Xi[\pi \Rightarrow \phi]$ short for $\Xi[\pi \mapsto \Xi(\pi) \uplus \phi]$

Translation of atomic operations $\boxed{\kappa \vdash \langle \phi : a \rangle^n \overset{\pi}{\Longrightarrow} \langle \phi \rangle^n}$

SEND
$$\frac{S = \texttt{if}\ c < \kappa(c)\ \{c \coloneqq c{+}1;\ \mathsf{P}(\pi) \coloneqq n{+}1\}}{\kappa \vdash \langle \phi : c! \rangle^n \overset{\pi}{\Longrightarrow} \langle \phi[n \mapsto S] \rangle^{n+1}}$$

RECEIVE
$$\frac{S = \texttt{if}\ c > 0\ \{c \coloneqq c{-}1;\ \mathsf{P}(\pi) \coloneqq n{+}1\}}{\kappa \vdash \langle \phi : c? \rangle^n \overset{\pi}{\Longrightarrow} \langle \phi[n \mapsto S] \rangle^{n+1}}$$

SKIP
$$\frac{}{\kappa \vdash \langle \phi : \texttt{skip} \rangle^n \overset{\pi}{\Longrightarrow} \langle \phi \rangle^n}$$

A-SEQ
$$\frac{\kappa \vdash \langle \phi : a_1 \rangle^n \overset{\pi}{\Longrightarrow} \langle \phi_1 \rangle^{n_1} \qquad \kappa \vdash \langle \phi_1 : a_2 \rangle^{n_1} \overset{\pi}{\Longrightarrow} \langle \phi_2 \rangle^{n_2}}{\kappa \vdash \langle \phi : a_1; a_2 \rangle^n \overset{\pi}{\Longrightarrow} \langle \phi_2 \rangle^{n_2}}$$

Translation of structured control flow $\boxed{\langle \kappa, \Xi : V \rangle^n \overset{\pi}{\Longrightarrow} \langle \kappa, \Xi \rangle^n}$

ATOMIC
$$\frac{\kappa \vdash \langle [] : a \rangle^n \overset{\pi}{\Longrightarrow} \langle \phi \rangle^{n'}}{\langle \kappa, \Xi : a \rangle^n \overset{\pi}{\Longrightarrow} \langle \kappa, \Xi[\pi \Rightarrow \phi] \rangle^{n'}}$$

CHAN
$$\frac{\phi = \left[\ n \mapsto \texttt{if}\ e < 1\ \{\texttt{return}\};\ \mathsf{P}(\pi) \coloneqq n{+}1\ \right]}{\langle \kappa, \Xi : c = \texttt{chan}\ [e] \rangle^n \overset{\pi}{\Longrightarrow} \langle \kappa[c \mapsto e], \Xi[\pi \Rightarrow \phi] \rangle^{n+1}}$$

GO
$$\frac{\phi = [n \mapsto\ \mathsf{P}(\pi) \coloneqq n+1;\ \mathsf{P}(\pi') \coloneqq 0] \qquad \langle \kappa, [] : V \rangle^0 \overset{\pi'}{\Longrightarrow} \langle \kappa', \Xi' \rangle^{n'} \qquad \pi' \text{ fresh}}{\langle \kappa,\ \Xi : \texttt{go}\ \{\ V\ \}\rangle^n \overset{\pi}{\Longrightarrow} \langle \kappa',\ \Xi[\pi \Rightarrow \phi]\ \uplus\ \Xi'[\pi' \Rightarrow [-1 \mapsto \{\}, n' \mapsto \{\}]] \rangle^{n+1}}$$

V-SEQ
$$\frac{\langle \kappa, \Xi : V_1 \rangle^n \overset{\pi}{\Longrightarrow} \langle \kappa_1, \Xi_1 \rangle^{n_1} \qquad \langle \kappa_1, \Xi_1 : V_2 \rangle^{n_1} \overset{\pi}{\Longrightarrow} \langle \kappa_2, \Xi_2 \rangle^{n_2}}{\langle \kappa, \Xi : V_1;\ V_2 \rangle^n \overset{\pi}{\Longrightarrow} \langle \kappa_2, \Xi_2 \rangle^{n_2}}$$

FOR
$$\frac{\kappa \vdash \langle [] : a \rangle^{n+1} \overset{\pi}{\Longrightarrow} \langle \phi \rangle^{n'} \qquad \phi' = \left[\begin{array}{ll} n \mapsto \texttt{if}\ i < e_2\ \{\mathsf{P}(\pi) \coloneqq n+1\}\ \texttt{else}\ \{\mathsf{P}(\pi) \coloneqq n'+1\} \\ n' \mapsto i \coloneqq i+1;\ \mathsf{P}(\pi) \coloneqq n & \text{with } i \text{ fresh} \end{array}\right]}{\langle \kappa, \Xi : \texttt{for}\ e_1\ ..\ e_2\ \{\ a\ \}\rangle^n \overset{\pi}{\Longrightarrow} \langle \kappa, \Xi[\pi \Rightarrow \phi \uplus \phi'] \rangle^{n'+1}}$$

Translation of VirGo programs $\boxed{V \Longrightarrow \Xi, \kappa}$

PROGRAM
$$\frac{\langle [], [] : V \rangle^0 \overset{\pi_0}{\Longrightarrow} \langle \kappa, \Xi \rangle^n \qquad \phi = [n \mapsto \{\}]}{V \Longrightarrow \Xi[\pi_0 \Rightarrow \phi], \kappa}$$

Fig. 6. Translation to sequential processes.

Concretely, each process becomes a mapping from each program point to a CoreDafny statement that models the semantics of the next instruction.

We use the following notation (see top of Figure 6): $\pi \in \Pi$ and $n \in N$ range over finite sets of, respectively, process identifiers and program points (i.e., possible values of the program counters). The program points are represented as integers $\mathbb{Z}$ according to their order in the textual program code, using -1 to denote the program point for processes that have not yet been created. Process identifiers π can similarly be represented as natural numbers $\mathbb{N}$ so that $0 \leq \pi < |\Pi|$. $\mathsf{P}(\pi)$ refers to the name of the CoreDafny variable holding the current value of the program counter of process π, and similarly, $\mathsf{T}(\pi)$ is the name of the CoreDafny variable holding the process exit program point for π.

Process encoding. We formalize the translation in Figure 6 using three syntax-driven inductive judgments. The first judgment handles atomic VirGo statements: $\kappa \vdash \langle \phi : a \rangle^n \overset{\pi}{\Longrightarrow} \langle \phi' \rangle^{n'}$, where environment κ binds channel names to expressions denoting their capacity, π is the identifier of the process being translated, ϕ is the current encoding of the process π (mapping program points to CoreDafny statements), a is the statement to be translated, n is the program point immediately before a, ϕ' is the result of adding the translation of a to ϕ, and n' is the next program point. Rule SEND translates a send on c into an increment of variable c (capacity allowing). Rule RECEIVE translates a receive from c as a decrement of c (if $c > 0$). Rules SKIP and A-SEQ are straightforward.

Judgment $\langle \kappa, \Xi : V \rangle^n \overset{\pi}{\Longrightarrow} \langle \kappa, \Xi' \rangle^{n'}$ deals with the translation of general statements (V in Figure 2). Here, Ξ and Ξ' are mappings from process identifiers to process encodings (input and output, respectively) and the other parameters are identical to the previous judgment. Rule ATOMIC models atomic statements using the corresponding judgment. We use $\uplus$ to denote union of maps (with disjoint keys) and we write $\Xi[\pi \Mapsto \phi]$ to append ϕ to the value corresponding to π in Ξ. Rule CHAN models channel creation by recording the channel capacity e in environment κ and inserting a capacity check to ensure it is a buffered channel (the program terminates if the capacity is <1).

Rule GO models the spawning of a new concurrent process: it increments the program counter of the parent process (π) and sets the program counter of the child process (π', a globally fresh identifier) to 0, thus allowing it to start its execution. The encoding of the child is extended with two additional bindings: -1 as a value indicating the process has not yet been spawned, and n' as the last program point, to mark goroutine termination. Process π' is encoded separately in environment Ξ'. Rules V-SEQ and FOR are straightforward.

Finally, judgment $V \Longrightarrow \Xi, \kappa$ expresses the translation of a whole VirGo program V with main process π_0. In rule PROGRAM, n becomes the final (noop) instruction point of the main program.

To generate the nested **match**-constructs of Figure 5, we produce a case in the the outer **match**-construct for each $\pi \in dom(\Xi)$, then we iterate over the elements of each $\Xi(\pi)$ to generate the cases of each inner **match**-construct.

Syntactic occurrences. Figure 7 (top) defines some sets of syntactic occurrences that compactly gather relevant information about the representation of higher-level constructs in the process encoding. They are used to annotate the generated CoreDafny program with logical formulae. $\mathcal{P}$ is the set of all occurrences of a channel declaration. Each element $p \in \mathcal{P}$ is a tuple (π, n, c) where π is the process identifier declaring the channel, n is the program point of the declaration, and c is the declared channel name. O is the set of all occurrences of a channel operation in the translated program. Each element $o \in O$ is a tuple (π, n, c, d) where π is the process identifier executing that operation, n is the program point where that operation is executed, c is the channel name on which the operation is invoked, and d is the direction of the operation: send (!) or receive (?). $\mathcal{G}$ is the set of all occurrences of a goroutine spawning operation. Each element $g \in \mathcal{G}$ is a

Conventions and notations

Given function $J(r)$ and set $R = \{r_1, \ldots, r_k\}$ we write $J(R)$ for $J(r_1) \wedge \ldots \wedge J(r_k)$

$\mathbb{N} \ni \mathsf{T}(\pi)$ termination point

Syntactic occurrences

$$p = (\pi, n, c, e) \in \mathcal{P} \subseteq \Pi \times N \times \mathbb{C} \times \mathbb{E} \qquad \text{Occurrences of } \texttt{chan}\ [e]$$

$$o = (\pi, n, c, d) \in \mathcal{O} \subseteq \Pi \times N \times \mathbb{C} \times \{!, ?\} \qquad \text{Occurrences of } c! \text{ and } c?$$

$$g = (\pi, n, \pi') \in \mathcal{G} \subseteq \Pi \times N \times \Pi \qquad \text{Occurrences of } \texttt{go}$$

$$l = (\pi, n, n', i, e, e', O) \in \mathcal{L} \subseteq \Pi \times N \times N \times \mathbb{I} \times \mathbb{E} \times \mathbb{E} \times \wp(\mathcal{O}) \qquad \text{Occurrences of } \texttt{for}$$

$$o = (\pi, n, c, d) \in \mathcal{O}_{\mathcal{L}} = \bigcup\{O \mid (\pi, n, n', i, e, e', O) \in \mathcal{L}\} \qquad \text{Operations in loops}$$

Enabled and Post-condition

$$enabled(\pi) = \quad -1 < \mathsf{P}(\pi) < \mathsf{T}(\pi) \wedge \bigwedge_{(\pi,n,c,d) \in O} \mathsf{P}(\pi) = n \implies (d =! \implies c < \kappa(c)) \wedge (d =? \implies c > 0)$$

$$enabled(V) = \bigvee_{\pi \in \Pi} enabled(\pi) \qquad \frac{V \Longrightarrow \Xi, \kappa \qquad \Pi = dom(\Xi)}{terminated(V) = \bigwedge_{\pi \in \Pi} \mathsf{P}(\pi) \neq -1 \implies \mathsf{P}(\pi) = \mathsf{T}(\pi)}$$

Invariant

$$\frac{o = (\pi, n, c, d) \qquad o \notin \mathcal{O}_{\mathcal{L}}}{Icom(o) = \texttt{if}\ n < \mathsf{P}(\pi)\ \texttt{then}\ 1\ \texttt{else}\ 0} \qquad \frac{o \in \mathcal{O}_{\mathcal{L}}}{Icom(o) = \emptyset}$$

$$\frac{l = (\pi, n_1, n_2, i, e, e', O) \qquad N = \{n \in N \mid (\pi, n, c, d) \in O\} \qquad E = \sum_{n \in N} \texttt{if}\ n < \mathsf{P}(\pi) < n_2\ \texttt{then}\ 1\ \texttt{else}\ 0}{IforCom(c, l, d) = (i - e) * |N| + E}$$

$$\frac{p = (\pi', n', c) \qquad E(d) = \sum_{l \in \mathcal{L}} IforCom(c, l, d) \ + \sum_{(\pi,n,c,!) \in O} Icom((\pi, n, c, d))}{Ichan(p) = \mathsf{P}(\pi') > n' \implies 0 \leq c \leq \kappa(c) \ \wedge \ c = E(!) - E(?)}$$

$$\frac{\pi = 0 \implies k = 0 \quad \pi > 0 \implies k = -1}{Iproc(\pi) = k \leq \mathsf{P}(\pi) \leq \mathsf{T}(\pi)} \qquad \frac{g = (\pi, n, \pi')}{Igo(g) = n < \mathsf{P}(\pi) \iff \mathsf{P}(\pi') = -1}$$

$$\frac{l = (\pi, n, n', i, e, e', O)}{\begin{array}{l} Ifor(l) = \texttt{if}\ e \leq e' \\ \quad \texttt{then}\ e \leq i \leq e' \ \wedge \ (\mathsf{P}(\pi) < n \implies i = e) \wedge \ (n < \mathsf{P}(\pi) < n' \implies i < e') \\ \qquad \wedge \ (n' \leq \mathsf{P}(\pi) \implies i = e') \\ \quad \texttt{else}\ i = e \wedge\ !(n < \mathsf{P}(\pi) < n') \end{array}}$$

$$\frac{V \Longrightarrow \Xi, \kappa \qquad \Pi = dom(\Xi) \qquad V :: \mathcal{P}, \mathcal{G}, \mathcal{L}, \mathcal{O}}{inv(V) = Ichan(\mathcal{P}) \ \wedge \ Iproc(\Pi) \ \wedge \ Icom(\mathcal{O}) \ \wedge \ IforCom(\mathbb{C} \times \mathcal{L} \times \{!.?\}) \ \wedge \ Igo(\mathcal{G}) \ \wedge \ Ifor(\mathcal{L})}$$

Fig. 7. Generation of termination condition, postcondition, and invariant.

triple (π, n, π') where π is the identifier of the parent process, n is the program point where the spawning is executed, and π' is the process identifier of the newly spawned process. $\mathcal{L}$ is the set of all occurrences of a **for**-loop in the source program. Each element $l \in \mathcal{L}$ is a tuple $(\pi, n, n', i, e, e', O)$ where π is the identifier of the process executing the loop, n is the loop initialization program point, n' is the termination program point, i is the name of the loop variable (in CoreDafny), e (resp. e') is the lower (resp. upper) bound expression, and $O \subseteq \mathcal{O}$ is the set of all channel operations in the loop body. Finally, the set $\mathcal{O}_{\mathcal{L}} \subseteq \mathcal{O}$ contains all operations that occur in loops. Hereafter, we write $V :: \mathcal{P}, \mathcal{G}, \mathcal{L}, \mathcal{O}$ when the encoding of program V produces occurrence set $\mathcal{P}$, $\mathcal{G}$, $\mathcal{L}$ and $\mathcal{O}$. We assume these sets to be globally available.

Initialization. The initialization phase (lines 7–11) in Figure 5 sets all state variables to their initial values. The program counter of all processes is set to -1 (except for the main thread set to 0). Each channel initially holds 0 messages, and each loop variable is set to its lower bound.

Progress. The termination condition of the **while**-loop, $enabled(V)$ (line 14 in Figure 5), is formally defined in Figure 7. The aim of the $enabled(V)$ property is to model when the whole program can *progress*, i.e., at least one process is enabled. A process with identifier π is enabled ($enabled(\pi)$) if it has started and has not terminated ($-1 < \mathsf{P}(\pi) < \mathsf{T}(\pi)$), and any channel action it is currently due to execute ($\mathsf{P}(\pi) = n$) is enabled: a send operation is enabled when the number of messages in the channel has not reached its capacity, and a receive operation is enabled when the channel is not empty.

4.3 Step 2: Termination Property and Invariants

We aim to verify that a program V gracefully terminates under certain conditions. We define graceful termination as $terminated(V)$, formally defined in Figure 7 as: all processes that have started ($\mathsf{P}(\pi) \neq -1$) have terminated ($\mathsf{P}(\pi) = \mathsf{T}(\pi)$).

Dafny is only able to reason about the generated programs when annotated with suitable invariants. We generate invariant properties using function $inv(V)$, invoked at line 15 of Figure 5, and defined in Figure 7. It is defined as a conjunction over predicates that enforce properties for each high-level language construct, correlating program points to process counters and channel buffer values. We use the convention in Figure 7 (top) to extend any function J to sets by taking the conjunction of the application of J on each element in the set.

Predicate $I_{chan}(c)$ states the properties related to channel c once it has been created (left-hand-side of the implication). The number of messages in the channel must not be negative and must not exceed its capacity. Additionally, the number of messages in the channel must be equal to the difference between the number of send and receive operations performed until the current execution point. To symbolically compute the number of operations we use $I_{com}(o)$ and $I_{forCom}(c, l, d)$. Function $I_{com}(o)$ deals with operations that occur outside of loops, returning 1 if the operation has executed, 0 otherwise. Function $I_{forCom}(c, l, d)$ models operations on channel c occurring in loop l. It adds the number of operations in completed iterations and the number of operations executed in the current iteration (E).

Predicate $I_{proc}(\pi)$ states properties for each process (with identifier π) whose program counter ($\mathsf{P}(\pi)$) should be within the expected range. Similarly, predicate $I_{go}(g)$ states that the program counter of a spawned process ($\mathsf{P}(\pi')$) must be set to -1 (not yet started) if its parent π' has not yet reached its spawning instruction.

Predicate $I_{for}(l)$ deals with state variables associated with loops. Essentially, if the loop is executable, then the loop variable (i) must be within the bounds, and if the program counter is within the range of the loop ($n < \mathsf{P}(\pi) < n'$), then the loop variable must be less than the upper bound (e').

$$\frac{l = (\pi, n_1, n_2, i, e_1, e_2, O) \qquad N = \{n \mid (\pi, n, c, d) \in O\} \qquad e = \texttt{if } e_1 > e_2 \texttt{ then } 0 \texttt{ else } e_2 - e_1}{for(c, l, d) = e * |N|}$$

$$\frac{O = \{n \mid (\pi, n, c, d) \in O \setminus O_{\mathcal{L}}\}}{ops(c, d) = |O| + \sum_{l \in \mathcal{L}} for(c, l, d)} \qquad \frac{E_! = ops(c, !) \qquad E_? = ops(c, ?)}{com(c) = E_? \leq E_! \leq E_? + \kappa(c)}$$

$$\frac{V \Longrightarrow \Xi, \kappa}{balanced(V) = \bigwedge_{c \in dom(\kappa)} \kappa(c) > 0 \wedge com(c)}$$

Fig. 8. Precondition formula.

4.4 Step 3: Strategies for Precondition Discovery

The precondition P and postcondition Q at lines 4–5 of Figure 5 depend on the strategy being used. We outline three strategies below, which we use in the given order (until success or all strategies have been attempted).

Strategy 1: Attempt to prove $\{\texttt{true}\}enc(V)\{terminated(V)\}$, i.e., that the CoreDafny encoding of a VirGo programs V guarantees $terminated(V)$ without any precondition. If this succeeds, then all processes terminate for all inputs and all schedules.

Otherwise, we synthesize a formula, $balanced(V)$, defined in Figure 8, which is used in the next two strategies. For each channel, the formula enforces that its capacity is strictly positive and (corresponding to **C1** and **C2** in Section 3) that the number of send ($E_!$) and receive ($E_?$) actions match, up to the capacity of the channel ($\kappa(c)$). The function that generates symbolic expressions for $E_!$ and $E_?$ works similarly to the invariant formula $I_{forCom}(c, l, d)$ in Figure 7.

Strategy 2: Check $\{\texttt{true}\}enc(V)\{balanced(V) \iff terminated(V)\}$, i.e., that the CoreDafny encoding of a VirGo program V satisfies $terminated(V)$ if and only if $balanced(V)$ holds. If this succeeds, then all processes terminate for all schedules and inputs for which $balanced(V)$ holds, and $balanced(V)$ is the weakest precondition that satisfies this requirement. This follows from the fact that all concurrency parameters are *immutable* in VirGo. Note that this check can be applied to any precondition suggestion strategy.

Strategy 3: Attempt to prove $\{balanced(V)\}enc(V)\{terminated(V)\}$, i.e., that the CoreDafny encoding of a VirGo program V satisfies $terminated(V)$ when $balanced(V)$ holds. If this succeeds, then all processes terminate for all inputs and all schedules whenever $balanced(V)$ holds (but this may not be the weakest precondition).

5 Translating All VirGo Features to CoreDafny

This section describes the rest of the translation from VirGo to CoreDafny, including all features omitted from the previous section. These detailed extensions are not essential for understanding the main ideas of the approach but are given for the sake of completeness. We keep explanations brief as the rules are mostly simple extensions of the ones in Section 4. The extensions presented here preserve the sound approach presented earlier and are reflected in our implementation.

We describe the encoding to program points of each VirGo construct (Section 5.1), followed by invariant and precondition generation (Sections 5.2 and 5.3, respectively).

Conventions

Constants for synchronous channel encoding: $\mathsf{RDY} = 0$ $\mathsf{SND} = 1$ $\mathsf{ACK} = -1$

Atomic operations

$$\boxed{\kappa \vdash \langle \phi : a \rangle^n \overset{\pi}{\Longrightarrow} \langle \phi \rangle^n}$$

SEND

$$\frac{\begin{array}{l} S_1 = \texttt{if}\ \kappa(c) > 0\ \{ \\ \quad\quad \texttt{if}\ c < \kappa(c)\ \{c \coloneqq c{+}1;\ \mathsf{P}(\pi) \coloneqq n{+}2\} \\ \quad \}\ \texttt{else}\ \{ \\ \quad\quad \texttt{if}\ c == \mathsf{RDY}\ \{c \coloneqq \mathsf{SND};\ \mathsf{P}(\pi) \coloneqq n{+}1\} \\ \quad \} \end{array} \qquad \begin{array}{l} S_2 = \texttt{if}\ c == \mathsf{ACK}\ \{ \\ \quad\quad c \coloneqq \mathsf{RDY}; \\ \quad\quad \mathsf{P}(\pi) \coloneqq n{+}2 \\ \quad \} \end{array}}{\kappa \vdash \langle \phi : c! \rangle^n \overset{\pi}{\Longrightarrow} \langle \phi[n \mapsto S_1,\ n+1 \mapsto S_2] \rangle^{n+2}}$$

RECEIVE

$$\frac{\begin{array}{l} S = \texttt{if}\ \kappa(c) > 0\ \{ \\ \quad\quad \texttt{if}\ c > 0\ \{c \coloneqq c{-}1;\ \mathsf{P}(\pi) \coloneqq n{+}1\} \\ \quad \}\ \texttt{else}\ \{ \\ \quad\quad \texttt{if}\ c == \mathsf{SND}\ \{c \coloneqq \mathsf{ACK};\ \mathsf{P}(\pi) \coloneqq n{+}1\} \\ \quad \} \end{array}}{\kappa \vdash \langle \phi : c? \rangle^n \overset{\pi}{\Longrightarrow} \langle \phi[n \mapsto S] \rangle^{n+1}}$$

ADD

$$\frac{\begin{array}{l} S = \texttt{if}\ w < -e\ \{\ \texttt{return}\ \}; \\ \quad\quad w \coloneqq w + e;\ \mathsf{P}(\pi) \coloneqq n{+}1 \end{array}}{\kappa \vdash \langle \phi : w.\mathsf{Add}(e) \rangle^n \overset{\pi}{\Longrightarrow} \langle \phi[n \mapsto S] \rangle^{n+1}}$$

Additional control flow constructs

$$\boxed{\langle \kappa, \Xi : V \rangle^n \overset{\pi}{\Longrightarrow} \langle \kappa, \Xi \rangle^n}$$

WG

$$\frac{}{\langle \kappa, \Xi : w\ =\ \texttt{WaitGroup} \rangle^n \overset{\pi}{\Longrightarrow} \langle \kappa, \Xi \rangle^n}$$

WAIT

$$\frac{\phi = [n \mapsto \texttt{if}\ w == 0\ \{\ \mathsf{P}(\pi) \coloneqq n{+}1\ \}]}{\langle \kappa, \Xi : w.\mathsf{Wait}() \rangle^n \overset{\pi}{\Longrightarrow} \langle \kappa, \Xi[\pi \Mapsto \phi] \rangle^{n+1}}$$

RETURN

$$\frac{\phi = [n \mapsto \mathsf{P}(\pi) \coloneqq \mathsf{T}(\pi)]}{\langle \kappa, \Xi : \texttt{return} \rangle^n \overset{\pi}{\Longrightarrow} \langle \kappa, \Xi[\pi \Mapsto \phi] \rangle^{n+1}}$$

IF

$$\frac{\langle \kappa, \Xi : V_1 \rangle^{n+1} \overset{\pi}{\Longrightarrow} \langle \kappa_1, \Xi_1 \rangle^{n'} \qquad \langle \kappa_1, \Xi_1 : V_2 \rangle^{n'+1} \overset{\pi}{\Longrightarrow} \langle \kappa_2, \Xi_2 \rangle^{n_2} \qquad \phi = \left[\begin{array}{l} n \mapsto \mathsf{P}(\pi) \coloneqq 1 + \texttt{if}\ e\ \texttt{then}\ n\ \texttt{else}\ n' \\ n' \mapsto \mathsf{P}(\pi) \coloneqq n_2 \end{array} \right]}{\langle \kappa, \Xi : \texttt{if}\ e\ \{\ V_1\ \}\ \texttt{else}\ \{\ V_2\ \} \rangle^n \overset{\pi}{\Longrightarrow} \langle \kappa_2, \Xi_2[\pi \Mapsto \phi] \rangle^{n_2}}$$

Fig. 9. Translation rules for additional syntactic features and synchronous channel operations.

5.1 Encoding

The rules to transform a VirGo program into CoreDafny processes are given in Figure 9. Rules send and receive now check whether the channel is buffered ($\kappa(c) > 0$) or synchronous. The asynchronous case is as in Section 4. In the synchronous case ($\kappa(c) == 0$), a

Enabled

$$
\begin{aligned}
& & & \textbf{if } \kappa(c) > 0 \\
& enabled_!(\pi, n, c) & = & \textbf{then } \mathsf{P}(\pi) = n \implies c < \kappa(c) \\
& & & \textbf{else } (\mathsf{P}(\pi) = n \implies c = \mathsf{RDY}) \wedge (\mathsf{P}(\pi) = n + 1 \implies c = \mathsf{ACK}) \\
& enabled_?(\pi, n, c) & = & \mathsf{P}(\pi) = n \implies (\kappa(c) > 0 \implies c > 0) \wedge (\kappa(c) == 0 \implies c = \mathsf{SND}) \\
& enabled_C(\pi, n, c, d) & = & (d = ! \implies enabled_!(\pi, n, c)) \wedge (d = ? \implies enabled_?(\pi, n, c)) \\
& enabled_W(\pi, n, w) & = & \mathsf{P}(\pi) = n \implies w = 0 \\
& enabled(\pi) & = & -1 < \mathsf{P}(\pi) < \mathsf{T}(\pi) \wedge \\
& & & \bigwedge_{(\pi,n,c,d) \in O} enabled_C(\pi, n, c, d) \wedge \bigwedge_{(\pi,n,w) \in \mathcal{W}_W} enabled_W(\pi, n, w) \\
& enabled(V) & = & \bigvee_{\pi \in \Pi} enabled(\pi)
\end{aligned}
$$

Fig. 10. Revised enabled predicate with support for synchronous channels and waitgroups.

channel is only available when it is *ready* (RDY), and the first process to send a message sets the value of the channel to *sending* (SND). The sender is then blocked until a receiver acknowledges the synchronization by setting the channel to ACK, and continuing to the next instruction.

Waitgroup declarations do not require new program points as the corresponding Dafny variables modeling waitgroup counters are initialized to 0. Waitgroup operations are modeled via the ADD and WAIT rules. Adding to the waitgroup simply increments the value with the given expression e. Note that the Go waitgroup method `w.Done()` is translated to `w.Add(-1)` in VirGo. The guard $w < -e$ checks whether the new value is negative, in which case the execution of the entire CoreDafny model is terminated immediately (without reaching its expected end state). This models the run-time error thrown when a waitgroup counters become negative in Go. Waiting on a waitgroup blocks until the counter is zero, just like in Go.

VirGo **`return`** statements translate to setting the process counter to its termination point. Two program points are reserved for **`if`** statements: one for the branching instruction that decides the continuation, and one at the end of the **`true`** branch that transfers control flow to the continuation, skipping the **`else`** branch instructions.

Progress. In Figure 10, we give a revised $enabled(V)$ predicate, used to identify whether some process can make some progress. It is extended to model the blocking behavior of unbuffered channel and `w.Wait()` operations. For channels, an additional case is added for processes in the rendezvous state, while send or receive operation cases take the capacity into account. For waitgroups, a process is blocked on a wait operation until the waitgroup counter is 0.

5.2 Invariants with Support for Waitgroups, Conditionals, and Return

We proceed in two steps as the invariant construction that supports **`if`** statements need first to keep track of the conditions that make a given program point reachable in the source VirGo program.

Reachability. A *reachability environment*, $\psi : (\Pi \times N) \mapsto \mathbb{E}$ binds a process program point to a Boolean expression modeling the path conditions under which the program point is reachable.

Conventions

$$\Psi \ni \psi : (\Pi \times N) \mapsto \mathbb{E} \quad \text{Reachability environment}$$

Reachability for atomic operations $\boxed{e \vdash \langle s \rangle_n \overset{\pi}{\rightsquigarrow} \langle \psi \mid e \rangle_n}$

SEND
$$e \vdash \langle c! \rangle_n \overset{\pi}{\rightsquigarrow} \langle [(\pi, n) \mapsto e] \mid \textbf{false} \rangle_{n+2}$$

RECEIVE
$$e \vdash \langle c? \rangle_n \overset{\pi}{\rightsquigarrow} \langle [(\pi, n) \mapsto e] \mid \textbf{false} \rangle_{n+1}$$

ADD
$$e \vdash \langle w.\texttt{Add}(e) \rangle_n \overset{\pi}{\rightsquigarrow} \langle [(\pi, n) \mapsto e] \mid \textbf{false} \rangle_{n+1}$$

Reachability for statements $\boxed{e \vdash \langle V \rangle_n \overset{\pi}{\rightsquigarrow} \langle \psi \mid e \rangle_n}$

RETURN
$$\frac{\psi = [(\pi, n) \mapsto e]}{e \vdash \langle \textbf{return} \rangle_n \overset{\pi}{\rightsquigarrow} \langle \psi \mid e \rangle_{n+1}}$$

WAITGROUP
$$\frac{}{e \vdash \langle w = \textbf{WaitGroup} \rangle_n \overset{\pi}{\rightsquigarrow} \langle [] \mid \textbf{false} \rangle_n}$$

CHAN
$$\frac{\psi = [(\pi, n) \mapsto e]}{e \vdash \langle c = \textbf{chan}\ [e] \rangle_n \overset{\pi}{\rightsquigarrow} \langle \psi \mid \textbf{false} \rangle_{n+1}}$$

WAIT
$$\frac{\psi = [(\pi, n) \mapsto e]}{e \vdash \langle w.\texttt{Wait}() \rangle_n \overset{\pi}{\rightsquigarrow} \langle \psi \mid \textbf{false} \rangle_{n+1}}$$

GO
$$\frac{e \vdash \langle V \rangle_0 \overset{next(\pi)}{\rightsquigarrow} \langle \psi \mid e' \rangle_{n'}}{e \vdash \langle \textbf{go}\ \{\ V\ \} \rangle_n \overset{\pi}{\rightsquigarrow} \langle \psi [(\pi, n) \mapsto e] \mid \textbf{false} \rangle_{n+1}}$$

SEQ
$$\frac{e \vdash \langle V_1 \rangle_n \overset{\pi}{\rightsquigarrow} \langle \psi_1 \mid e_1 \rangle_{n_1} \qquad e \ \wedge\ !e_1 \vdash \langle V_2 \rangle_{n_1} \overset{\pi}{\rightsquigarrow} \langle \psi_2 \mid e_2 \rangle_{n_2}}{e \vdash \langle V_1\,;\ V_2 \rangle_n \overset{\pi}{\rightsquigarrow} \langle \psi_1 \uplus \psi_2 \mid e_1 \ \vee\ e_2 \rangle_{n_2}}$$

FOR
$$\frac{e \vdash \langle a \rangle_{n+1} \overset{\pi}{\rightsquigarrow} \langle \psi \mid \textbf{false} \rangle_{n'}}{e \vdash \langle \textbf{for}\ e_1\ ..\ e_2\ \{\ a\ \} \rangle_n \overset{\pi}{\rightsquigarrow} \langle \psi [n \mapsto e] \mid \textbf{false} \rangle_{n'+1}}$$

IF
$$\frac{e \ \wedge\ e' \vdash \langle V_1 \rangle_{n+1} \overset{\pi}{\rightsquigarrow} \langle \psi_1 \mid e_1 \rangle_{n_1} \qquad e \ \wedge\ !e' \vdash \langle V_2 \rangle_{n_1+1} \overset{\pi}{\rightsquigarrow} \langle \psi_2 \mid e_2 \rangle_{n_2}}{e \vdash \langle \textbf{if}\ e'\ \{\ V_1\ \}\ \textbf{else}\ \{\ V_2\ \} \rangle_n \overset{\pi}{\rightsquigarrow} \langle (\psi_1 \uplus \psi_2)[n \mapsto e] \mid e_1 \ \vee\ e_2 \rangle_{n_2}}$$

Reachability for programs $\boxed{V \rightsquigarrow \psi}$

PROGRAM
$$\frac{\textbf{true} \vdash \langle V \rangle_0 \overset{\pi_0}{\rightsquigarrow} \langle \psi | e \rangle_n}{V \rightsquigarrow \psi}$$

Fig. 11. Rules for constructing the path reachability for each program point.

Figure 11 gives the syntax-directed rules to construct such an environment. The main judgement is of the form $e \vdash \langle V \rangle_n \overset{\pi}{\rightsquigarrow} \langle \psi \mid e' \rangle_n$ where e is a Boolean expression that needs to be satisfied for statement V to be executed, ψ is the reachability environment under-construction, and e' represents a Boolean condition that is true when the program has returned early. We write $V \rightsquigarrow \psi$ for "the reachability environment ψ is derived from V". Reachability is affected by **if** statements, where the path condition for the program points in each branch statements are extended with the guard expression (negated for the **else** branch). Reachability also keeps track of the (negated) path conditions of preceding **return** statements, since instructions syntactically following a **return** statement may not execute if the function has returned early.

Invariant generation. The rules to construct invariants from Section 4.3 are extended with predicates modeling all features, and extended with program point reachability, $\psi(\pi, n)$. Figure 12 gives the revised rules for channel operations which deal with buffered and synchronous channels. In the unbuffered case, the channel state is constrained wrt. the number of concurrency operations performed as well as restrictions enforcing that at most one process is executing a rendezvous at any given point. Conversely, the invariant models the rendezvous instruction as unreachable for buffered channels.

In Figure 13, we introduce additional sets of syntactic occurrences to deal with waitgroup operations, conditionals, and return statements. The information about loops is extended to include waitgroup operations in their body. The sets $\mathcal{W}_W$ and $\mathcal{W}_A$ represent the occurrences of each w`.Wait()` and w`.Add(`e`)` operations, respectively, keeping track of the executing process, program point and, for the latter, the added expression e. Waitgroup invariants model the waitgroup counter relative to process states. The expression value of every `Add` operation outside a loop is added to the waitgroup counter, while expressions in loops are multiplied by the number of loop iterations. The invariant also models run-time errors caused by negative waitgroup counters. The set $\mathcal{R}$ contains all **return** statements, represented as process and program points pairs, (π, n). The generated invariant $I_{ret}(r)$, $r \in \mathcal{R}$, models how a process executing a reachable **return** statement skips to termination without visiting any successor program points. The set $\mathcal{I}$ contains all **if** statements, represented as tuples (π, n_1, n_2, n_3, e), where e is the guard expression, π the process, and program points n_1, n_2 and n_3 represent the guard program point, the beginning of the else branch, and the successor of the **if** statement, respectively. The generated invariant $I_{if}(i)$, $i \in \mathcal{I}$, asserts which branch program points are unreachable, depending on the value of the guard.

5.3 Preconditions

Figure 14 details how the precondition predicate is extended to support waitgroups, as used in Strategies 2 and 3. Essentially, it generates an expression representing the sum of all expressions across add operations and requires this sum to amount to 0 for each waitgroup. This matches a programming pattern often observed in practice where w`.Wait()` is used only once, and syntactically preceded by all w`.Add(`e`)` operations.

6 Implementation

The approach has been implemented in the tool Ginger. It reuses Gomela to parse Go in the front-end and Dafny as a verifier in the back-end. We now present how the behaviors of Go program fragments are over-approximated with VirGo, and how to handle additional Go features.

6.1 Over-Approximating Go Programs with VirGo

To apply Ginger on real-world programs, they must be broken down into VirGo-expressible fragments. A program fragment is a set of related functions that cover the scope of a set of

Invariant generation with support for synchronous channels

$$\frac{o = (\pi, n, c, d) \qquad o \notin O_{\mathcal{L}}}{I_{com}(o) = \textbf{if } \psi(\pi, n) \wedge n < \mathsf{P}(\pi) \textbf{ then } 1 \textbf{ else } 0} \qquad \frac{o \in O_{\mathcal{L}}}{I_{com}(o) = 0}$$

$$\frac{o = (\pi, n, c, !) \qquad o \notin O_{\mathcal{L}}}{I_{send}(o) = \begin{array}{ll} & \textbf{if } \psi(\pi, n) \wedge \quad n \quad < \mathsf{P}(\pi) \textbf{ then } 1 \textbf{ else } 0 \\ + & \textbf{if } \psi(\pi, n) \wedge \quad 1 + n \quad < \mathsf{P}(\pi) \textbf{ then } 1 \textbf{ else } 0 \end{array}} \qquad \frac{o \in O_{\mathcal{L}}}{I_{send}(o) = 0}$$

$$\frac{o = (\pi, n, c, ?) \qquad o \notin O_{\mathcal{L}}}{I_{recv}(o) = \textbf{if } \psi(\pi, n) \wedge n < \mathsf{P}(\pi) \textbf{ then } 2 \textbf{ else } 0} \qquad \frac{o \in O_{\mathcal{L}}}{I_{recv}(o) = 0}$$

$$\frac{\begin{array}{c} l = (\pi, n, n', x, e, e', O, W) \\ N = \{\hat{n} \in N \mid (\pi, \hat{n}, c, d) \in O\} \qquad E = \sum_{\hat{n} \in N} \textbf{if } \hat{n} < \mathsf{P}(\pi) < n' \textbf{ then } 1 \textbf{ else } 0 \end{array}}{I_{forCom}(c, l, d) = \textbf{if } \psi(\pi, n) \textbf{ then } (x - e) * |N| + E \textbf{ else } 0}$$

$$\frac{\begin{array}{c} l = (\pi, n, n', x, e, e', O, W) \\ N = \{\hat{n} \in N \mid (\pi, \hat{n}, c, !) \in O\} \qquad E = \sum_{\hat{n} \in N} \begin{array}{ll} & \textbf{if} \quad \hat{n} \quad < \mathsf{P}(\pi) < n' \textbf{ then } 1 \textbf{ else } 0 \\ + & \textbf{if} \quad 1 + \hat{n} \quad < \mathsf{P}(\pi) < n' \textbf{ then } 1 \textbf{ else } 0 \end{array} \end{array}}{I_{forSend}(c, l) = \textbf{if } \psi(\pi, n) \textbf{ then } (x - e) * |N| + E \textbf{ else } 0}$$

$$\frac{\begin{array}{c} l = (\pi, n, n', x, e, e', O, W) \\ N = \{\hat{n} \in N \mid (\pi, \hat{n}, c, ?) \in O\} \qquad E = \sum_{\hat{n} \in N} \textbf{if } \hat{n} < \mathsf{P}(\pi) < n' \textbf{ then } 2 \textbf{ else } 0 \end{array}}{I_{forRecv}(c, l) = \textbf{if } \psi(\pi, n) \textbf{ then } (x - e) * |N| + E \textbf{ else } 0}$$

$$\frac{E(d) = \sum_{(\pi,n,c,d) \in O} I_{com}(\pi, n) \; + \; \sum_{l \in \mathcal{L}} I_{forCom}(c, l, d) \qquad I_{noRendezvous} = \bigwedge_{(\pi,n,c,!) \in O} \mathsf{P}(\pi) \neq n + 1}{I_{async}(c) = 0 \leq c \leq \kappa(c) \wedge c = E(!) - E(?) \wedge I_{noRendezvous}}$$

$$\frac{\begin{array}{c} E_1 = \sum_{(\pi,n,c,!) \in O} I_{send}(\pi, n) \; + \; \sum_{l \in \mathcal{L}} I_{forSend}(c, l, d) \qquad E_2 = \sum_{(\pi,n,c,?) \in O} I_{recv}(\pi, n) \; + \; \sum_{l \in \mathcal{L}} I_{forRecv}(c, l, d) \\ I_{maxOneRendezvous} = \bigwedge_{(\pi,n,c,!) \in O} \mathsf{P}(\pi) = n + 1 \implies \bigwedge_{(\pi',n',c,!) \in O} \pi' \neq \pi \implies \mathsf{P}(\pi') \neq n' + 1 \end{array}}{I_{sync}(c) = \mathsf{ACK} \leq c \leq \mathsf{SND} \wedge c = E_1 - E_2 \wedge I_{maxOneRendezvous}}$$

$$\frac{p = (\pi', n', c)}{I_{chan}(p) = \psi(\pi', n') \implies \textbf{if } \kappa(c) > 0 \textbf{ then } I_{async}(c) \textbf{ else } I_{sync}(c)}$$

Fig. 12. Invariant generation rules for buffered and unbuffered channels operations.

concurrency primitives [8, 22, 38]. Functions are related if they are connected in the call graph and share concurrency primitives. A fragment is constructed from a function that allocates concurrency primitives. All callees that use concurrency primitives are transitively included. Concurrency

Syntactic occurrences

$wa = (\pi, n, w, e) \in \mathcal{W}_{\mathcal{A}} \subseteq \Pi \times N \times \mathbb{W} \times \mathbb{E}$ — Occurrences of $w.\texttt{Add}(e)$

$ww = (\pi, n, w) \in \mathcal{W}_{\mathcal{W}} \subseteq \Pi \times N \times \mathbb{W}$ — Occurrences of $w.\texttt{Wait}()$

$l = (..., W) \in \mathcal{L} \subseteq ... \times \wp(\mathcal{W}_{\mathcal{A}})$ — Occurrences of **for** extended with $w.\texttt{Add}(e)$

$wa = (\pi, n, w, e) \in \mathcal{W}_{\mathcal{A}\mathcal{L}} = \bigcup \{W \mid (..., W) \in \mathcal{L}\}$ — $w.\texttt{Add}(e)$ operations in loops

$i = (\pi, n, n', n'', e) \in \mathcal{I} \subseteq \Pi \times N \times N \times N \times \mathbb{E}$ — Occurrences of **if**

$r = (\pi, n) \in \mathcal{R} \subseteq \Pi \times N$ — Occurrences of **return**

Invariant generation with support for waitgroups

$$\frac{wa = (\pi, n, w, e) \qquad o \notin \mathcal{W}_{\mathcal{A}\mathcal{L}}}{I_{add}(wa) = \textbf{if } \psi(\pi, n) \wedge n < \mathsf{P}(\pi) \textbf{ then } e \textbf{ else } 0} \qquad \frac{wa \in \mathcal{W}_{\mathcal{A}\mathcal{L}}}{I_{add}(wa) = 0}$$

$$\frac{\begin{array}{c} l = (\pi, n, n', x, e, e', O, W) \qquad A = \{(\hat{n}, \hat{e}) \mid (\pi, \hat{n}, w, e) \in W\} \\ E_1 = \sum_{(\pi, \hat{n}, w, \hat{e}) \in W} \hat{e} \qquad E_2 = \sum_{(\hat{n}, \hat{e}) \in A} \textbf{if } \hat{n} < \mathsf{P}(\pi) < n' \textbf{ then } \hat{e} \textbf{ else } 0 \end{array}}{I_{forAdd}(w, l) = \textbf{if } \psi(\pi, n) \textbf{ then } (x - e) * E_1 + E_2 \textbf{ else } 0}$$

$$\frac{E = \sum_{l \in \mathcal{L}} I_{forAdd}(w, l) \; + \sum_{(\pi, n, w, e) \in \mathcal{W}_{\mathcal{A}}} I_{add}(\pi, n)}{I_{wg}(w) = 0 \leq w \wedge w = E} \qquad \frac{r = (\pi, n)}{I_{ret}(r) = \psi(\pi, n) \implies !(n < \mathsf{P}(\pi) < \mathsf{T}(\pi))}$$

$$\frac{g = (\pi, n, \pi')}{I_{go}(g) = \psi(\pi, n) \wedge n < \mathsf{P}(\pi) \iff \mathsf{P}(\pi') \neq -1}$$

$$\frac{i = (\pi, n_1, n_2, n_3, e)}{I_{if}(i) = \textbf{if } e \textbf{ then } !(n_2 \leq \mathsf{P}(\pi) < n_3) \textbf{ else } !(n_1 < \mathsf{P}(\pi) < n_2)}$$

$$\frac{l = (\pi, n, n', x, e, e', O, W)}{\begin{array}{l} I_{for}(l) = \textbf{if } \psi(\pi, n) \wedge e \leq e' \\ \qquad \textbf{then } e \leq x \leq e' \;\wedge\; (\mathsf{P}(\pi) < n \implies x = e) \wedge\; (n < \mathsf{P}(\pi) < n' \implies x < e') \\ \qquad\qquad \wedge\; (n' \leq \mathsf{P}(\pi) \implies x = e') \\ \qquad \textbf{else } x = e \wedge !(n < \mathsf{P}(\pi) < n') \end{array}}$$

Fig. 13. Invariant generation rules for waitgroups.

parameters are recorded while the fragment is being identified. Fragments are ruled out if precision is degraded in the discovery phase due to, e.g., escaping channels.

We adopt Gomela's front-end to excise each fragment from its calling context, identify concurrency parameters, and strip it of non-concurrent operations. These fragments are passed to Ginger for VirGo translation. The crux of the translation technique is to over-approximate the Go semantics by preserving the control flow and the interactions with concurrency primitives but

Precondition generation with support for waitgroups

$$\frac{l = (\pi, n_1, n_2, x, e_1, e_2, O, W)}{for(w, l) = \sum_{(\pi,n,w,e)\in W} e * (\textbf{if } \psi(\pi, n_1) \wedge e_1 < e_2 \textbf{ then } e_2 - e_1 \textbf{ else } 0)}$$

$$wg(w) = \sum_{(\pi,n,w,e)\in \mathcal{W}_{\mathcal{A}}\setminus\mathcal{W}_{\mathcal{AL}}} e + \sum_{l\in\mathcal{L}} for(w, l) \qquad \frac{V \Longrightarrow \Xi, \kappa}{balanced(V) = \bigwedge_{w\in\mathbb{W}} wg(w) = 0}$$

Fig. 14. Precondition generation for waitgroups.

abstracting away from non-concurrency related constructs, following techniques used in [8, 17, 18]. Note that since VirGo does not support named function calls and definitions, all process spawning and function calls are inlined. Additionally, we perform two syntactical transformation on the internal representation of Go programs to widen the applicability of Ginger.

First, to support the pattern of spawning goroutines in loops, we "commute" nested `for-go` patterns to `go-for` whenever possible, i.e., we transform the fragment on the left below to the one on the right.

```
for 0..x { go { P } }        ⟹        go { for 0..x { P } }
```

When `P` is `c!;...;c!` or `c?;...;c?`, this transformation is sound since we transform k identical processes each performing n identical operations into one process performing $k \cdot n$ operations. The transformation can also be soundly applied when `P` executes "atomically", e.g., if `P` is `mu.lock(); Q; mu.unlock()` (for some `Q`), and `mu.unlock()` does not appear anywhere else in the fragment (incl. `Q`). Ginger raises an alarm when the transformation cannot be applied soundly.

Second, to provide partial support for conditionals in loops, we merge branches whenever possible, i.e., when they perform the same sequence of concurrent actions. For instance, we transform the fragment on the left below to the one on the right while preserving behavior equivalence.

```
for 0..x { if e { c! } else { c! } }        ⟹        for 0..x {  c! }
```

6.2 Additional Language Features

The VirGo language, as presented in Figure 2, is fully supported by the translation to CoreDafny implemented in Ginger.

As described in Section 5, synchronous channels (with capacity 0) are encoded using an additional alternative for send and receive operations. For each operation on c, if $\kappa(c) = 0$ then we use an additional rendezvous state to encode synchronous communication. Waitgroup operations are encoded by allowing `w.Add(e)` to increment the corresponding state variable by e, and `w.Wait()` by blocking until the counter is set to 0. We also support standard mutexes by encoding them as channels of capacity one. If-then-else are supported by mapping each program point to a conjunction of conditionals that must be met for the instruction to be reachable. Every variable in conditionals that does not correspond to known concurrency parameters produces a fresh Dafny variable. Return statements are encoded as a jump to the termination point of the relevant process ($\mathsf{T}(\pi)$). Using these techniques, Ginger can analyze programs such as the fragment in Figure 1 (right), which includes a conditional and a return statement. We use a similar technique to support standard patterns such as blocking operations with timeouts, e.g., the Go program on the left becomes the VirGo program on the right below:

Table 1. Expressible fragments, and results of their verification.

Primitive	Fragments	Expressible	First successful strategy			
			Total	Strat. 1	Strat. 2	Strat. 3
Channels	258	159 (61.6%)	157	41	116	0
WaitGroup	205	171 (83.4%)	170	86	67	17
Both	120	48 (40%)	46	18	25	3
Total	**583**	**378** (64.8%)	**373**	**145**	**208**	**20**

```
select {                                                    if timeout_123 { c1?;... }
    case res := <-c1:...                          ⟹        else { ... }
    case <-time.After(time.Second):... }
```

The generated precondition may constrain fresh variable `timeout_123`.

7 Evaluation

We evaluate the approach by running Ginger on code fragments extracted from the large enterprise code base of Uber, which includes thousands of thoroughly tested microservices. The code base consists of ∼3 million LoC across packages that contain concurrency features. From these packages, we extracted 583 *parameterized* fragments ($|\mathbb{X}| > 0$) discovered using a simple syntactical analysis on the code base. The syntactical analysis selects fragments that are parameterized (they include parameterized loop bounds or a concurrency operation such as `make(chan T, k)` or `wg.Add(k)`, where `k` is a variable). Overall these parameterized fragments total 30,777 lines of code, where individual fragments are 53 lines on average, ranging from 7 to 309 lines. The number of distinct concurrency parameters per fragment has an average of 1.08 and ranges from 1 to 3, and concurrency parameters are used 2.20 times on average, between 1 and 6 times. Although typical fragments are small, they often exhibit complex concurrent behavior as in the motivating example from Section 2 and Examples 7.1 and 7.2 below.

We answer the following questions:

RQ1 How many parametric fragments can be expressed in VirGo via Ginger?
RQ2 How many expressible fragments can be successfully verified?
RQ3 How much CPU time does it take to verify VirGo encodings?

RQ1 - VirGo expressivity Table 1 shows the expressivity of VirGo wrt. the 583 parameterized fragments we extracted from Uber's code base. We classify fragments based on the types of concurrency primitives affected by parameters and measure the relative number of VirGo-expressible fragments. Note that 294 (out of 378) fragments are expressible thanks to a sound application of the `go-for` commute transformation; 32 additional fragments necessitated a potentially unsound `go-for` commute transformation. For all of these we manually confirmed that the analysis is sound. When translation fails, it is due to unsupported features, e.g., general select statements, or unsupported loops (nested, non-terminating, etc.). Features such as premature loop exit (via `break` or `return`) are suitable candidates for future work, while others are more challenging, e.g., non-deterministic `select` statements in loops.

RQ2 - Verifiability Table 1 also shows the precondition strategy that succeeds first, applied in the same order as presented in Section 4.4. The number of fragments verifiable with Strategy 1 relative to Strategies 2 and 3 suggests that developers often write fragments with some implicit assumptions. Strategies 2 and 3 help make these assumptions explicit. Suggested preconditions validated by Strategy 2 can be used by the developer to devise counter-examples e.g., by challenging assumptions about the sign of values used for channel capacities. When the verification fails with Strategy 2,

```
done = chan [0];
go { done!; };
go {
  for 0 .. readers / 2 {
    done!;
  }
};
go { done!; };
go {
  for readers / 2 .. readers {
    done!;
  }
}
go {
  for 0 .. readers + 2 {
    done?;
  }
}
```

(a) Successfully verified example.

```
if numLevels == 0 { return } else { skip }
w = WaitGroup;
c = chan [numLevels];
for 0 .. complianceLevels { w.Add(1); };
go {
  for 0 .. complianceLevels {
    c!;
    w.Add(-1);
  }
};
w.Wait();
for 0 .. numLevels {
  c?;
}
```

(b) Example that fails to verify.

Fig. 15. Complex examples of VirGo-expressible programs.

the invariant constructed in Section 4.3 is too weak to support the proof. Failure to verify with any strategy is either caused by unsuccessful applications of the `go-for` commute transformation presented in Section 6.2 (three cases) or by loss of vital constraints over concurrency parameters in the front-end.

Although we manually validated the fragments verified by Ginger, the impact of these bugs on the overall system's functionality would need to be evaluated in a wider context, and ideally in conjunction with the developers. Additionally, our manual validation process, may not capture all contextual dependencies and interactions within the entire code base, potentially overlooking the broader implications of the detected bugs.

Next we discuss two VirGo programs (adapted from representative Go code) that highlight the challenge addressed by the technique.

Example 7.1. Figure 15a is a program for which the absence of partial deadlocks depends the value of `readers`. At first glance, the number of send operations on `done` appears equal to the number of receive operations, making the program partial deadlock free. However, `readers` corresponds to an unconstrained function parameter, and thus may be instantiated to a negative number, in which case there will be unmatched (blocking) send operations.

If the number of loop iterations is defined as $\texttt{iter}(x, y)$, where $\texttt{iter}(x, y) = max(y - x, 0)$, Ginger infers the following precondition and validates it with Strategy 2:

$$\texttt{iter}(0, \texttt{readers}/2) + \texttt{iter}(\texttt{readers}/2, \texttt{readers}) + 2 = \texttt{iter}(0, \texttt{readers} + 2)$$

When `readers` is negative, the equality may not be satisfied (e.g., if $\texttt{readers} = -2$ the formula reduces to 2=0), and some of the sending goroutines get stuck.

For this example, Goat [38] and GCatch [22] unconditionally report partial deadlocks as they either over-approximate or unsoundly fix the number of loop iterations without considering the arithmetic connection between loop bounds. By contrast, Gomela [8] fails to detect the partial deadlock because `readers` is not tested by default for negative values.

Example 7.2. The program in Figure 15b is an example of a correct program that Ginger is unable to verify. In the source program, `complianceLevels` is assigned to `numLevels`, but the front-end fails

Table 2. Verification time per CoreDafny program per strategy (including failures). All experiments were performed on a MacBook Pro™ with a M1 Pro CPU (10 cores), and 32GB of memory.

Precondition strategy	Number of fragments	Verification time (s)			
		Avg.	P50	P90	Max
Strategy 1	378	2.19	1.73	3.08	4.3
Strategy 2	233	2.85	2.98	3.17	5.33
Strategy 3	25	2.86	2.98	3.21	3.32

to capture this equality. For this reason, all precondition strategies fail because send operations cannot be unblocked by receive operations, as all receive operations may only be executed after waiting on the waitgroup has been unblocked. The inferred *balanced* precondition is:

$$0 < \texttt{numLevels} \leq \texttt{complianceLevels} \leq \texttt{numLevels} + \texttt{numLevels}$$

It is easy to see that this condition is satisfied when `numLevels == complianceLevels`. As in the previous example, other tools fail to precisely reason about this example, as it is parameterized. Notably Gomela reports that some valuations fail and others succeed.

RQ3 - Performance In Table 2, we measure the verification time for each precondition strategy. We omit the execution times of the Gomela and Ginger translation phases as they are negligible. The majority of examples can be verified in under four seconds. This shows the technique can be applied to many real-world examples in a practical amount of time.

8 Related Work

Several approaches have been developed to detect or rule out bugs in concurrent Go programs. Besides the static verifiers we mentioned earlier (Goat [38], Gomela [8], GCatch [22]), earlier works include static checkers based on behavioral types [11, 17, 18, 27], abstract interpretation [24], and forkable regular expressions [34]. Except Gomela, which instantiates concurrency parameters based on user-provided values, all these approaches over-approximate concurrency parameters, or otherwise rely on unsound heuristics to handle imprecision introduced by loops, failing to perform a trustworthy analysis of Figure 1. Additionally, they cannot suggest preconditions that enforce partial deadlock freedom. Another static verifier for Go programs is Gobra [39], an extension of the Viper [26] framework, which enables the verification of programs for memory safety and partial correctness based on user-provided specifications (e.g., pre- and post-condition annotations within Go Programs). Gobra cannot detect blocking bugs such as partial deadlocks.

Dynamic partial deadlock detection techniques have also been proposed, such as [35], and GoLeak and LeakProf [30], two dynamic tools introduced by Uber that leverage an extensive testing and profiling infrastructure. GFuzz [21] relies on random exploration of test execution paths by fuzzing select case choices. Yuan et al. [40] have curated a benchmark suite of Go programs that targets dynamic verifiers and does not include parameterized communication patterns.

The automated generation of invariants has been studied extensively with particular focus on (sequential) transition systems rather than concurrent program fragments. Well-known techniques such as Craig interpolation [14, 23] and IC3 [3] (aka. PDR [9]) require exact knowledge of the underlying transition system, with the latter notably being adapted for software verification [5]. Some techniques rely on the existence of templates from which to derive invariants [4, 6, 33]. Other techniques use a black-box approach based on learning [12, 28] or large language models [15]. The discovery of invariants at runtime has also been studied, most notably with Daikon [10] which aims at discovering a (sequential) program's *likely* invariants by analyzing its executions. Our

technique uses a white-box approach and generates invariants following patterns informed by the specific communication mechanisms and their usage in Go. The techniques above could complete ours when it fails to generate a strong enough invariant.

Parameterized verification [2] addresses the analysis of concurrent system with an arbitrary number of processes, where some verification techniques generate invariants, e.g., [29]. While VirGo does not natively allow the arbitrary spawning of processes, it supports arbitrary parameters for loop bounds, path conditions and communication primitives.

Dafny has generally been used to write correct-by-construction programs, including concurrent systems [20], but not often as the backend verifier. We conjecture that using Dafny's backend, Boogie [1], would improve CPU-time at the cost of reduced transparency for the intermediate steps. The Civl verifier [16] (a recent extension of Boogie) could serve as an alternative back-end. Civl natively supports concurrency, and hence might allow us to simplify our encoding.

9 Conclusion

We have presented a novel approach for automatically verifying partial deadlock freedom in program fragments that feature parametric communication, by translating Go fragments to Dafny programs. The experimental results demonstrate that the approach can be used on an enterprise code base to verify a large number of such fragments that are out of reach for other verification techniques. For future work, it may be interesting to explore opportunities for generalizing the approach to cover a larger family of parameterized fragments and to adapt the approach to related languages that support channel-based process communication, such as Rust and Kotlin.

Data Availability Statement

The source code of Ginger is available at https://github.com/VladSaioc/ginger. An artifact that contains Ginger, the illustrated examples, and some additional fragments representative of real life coding patterns is archived on Zenodo [32]. Due to their proprietary nature, the fragments used in the evaluation are not publicly available.

References

[1] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures (Lecture Notes in Computer Science, Vol. 4111)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.). Springer, 364–387. https://doi.org/10.1007/11804192_17

[2] Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. 2015. *Decidability of Parameterized Verification.* Morgan & Claypool Publishers. https://doi.org/10.2200/S00658ED1V01Y201508DCT013

[3] Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6538)*, Ranjit Jhala and David A. Schmidt (Eds.). Springer, 70–87. https://doi.org/10.1007/978-3-642-18275-4_7

[4] Jason Breck, John Cyphert, Zachary Kincaid, and Thomas W. Reps. 2020. Templates and recurrences: better together. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 688–702. https://doi.org/10.1145/3385412.3386035

[5] Alessandro Cimatti and Alberto Griggio. 2012. Software Model Checking via IC3. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science, Vol. 7358)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, 277–293. https://doi.org/10.1007/978-3-642-31424-7_23

[6] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. 2003. Linear Invariant Generation Using Non-linear Constraint Solving. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July*

8-12, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2725), Warren A. Hunt Jr. and Fabio Somenzi (Eds.). Springer, 420–432. https://doi.org/10.1007/978-3-540-45069-6_39

[7] Nicolas Dilley and Julien Lange. 2019. An Empirical Study of Messaging Passing Concurrency in Go Projects. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*. IEEE, 377–387. https://doi.org/10.1109/SANER.2019.8668036

[8] Nicolas Dilley and Julien Lange. 2021. Automated Verification of Go Programs via Bounded Model Checking. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 1016–1027. https://doi.org/10.1109/ASE51524.2021.9678571

[9] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. 2011. Efficient implementation of property directed reachability. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, Per Bjesse and Anna Slobodová (Eds.). FMCAD Inc., 125–134. http://dl.acm.org/citation.cfm?id=2157675

[10] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1-3 (2007), 35–45. https://doi.org/10.1016/J.SCICO.2007.01.015

[11] Julia Gabet and Nobuko Yoshida. 2020. Static Race Detection and Mutex Safety and Liveness for Go Programs (Artifact). *Dagstuhl Artifacts Ser.* 6, 2 (2020), 12:1–12:3. https://doi.org/10.4230/DARTS.6.2.12

[12] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 69–87. https://doi.org/10.1007/978-3-319-08867-9_5

[13] C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (1978), 666–677. https://doi.org/10.1145/359576.359585

[14] Ranjit Jhala and Kenneth L. McMillan. 2006. A Practical and Complete Approach to Predicate Refinement. In *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3920)*, Holger Hermanns and Jens Palsberg (Eds.). Springer, 459–473. https://doi.org/10.1007/11691372_33

[15] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. 2023. Finding Inductive Loop Invariants using Large Language Models. https://doi.org/10.48550/arXiv.2311.07948

[16] Bernhard Kragl and Shaz Qadeer. 2021. The Civl Verifier. In *FMCAD*. IEEE, 143–152. https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_23

[17] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2017. Fencing off go: liveness and safety for channel-based programming. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 748–761. https://doi.org/10.1145/3009837.3009847

[18] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2018. A static verification framework for message passing in Go using behavioural types. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 1137–1148. https://doi.org/10.1145/3180155.3180157

[19] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6355)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[20] K. Rustan M. Leino. 2017. Modeling Concurrency in Dafny. In *Engineering Trustworthy Software Systems - Third International School, SETSS 2017, Chongqing, China, April 17-22, 2017, Tutorial Lectures (Lecture Notes in Computer Science, Vol. 11174)*, Jonathan P. Bowen, Zhiming Liu, and Zili Zhang (Eds.). Springer, 115–142. https://doi.org/10.1007/978-3-030-02928-9_4

[21] Ziheng Liu, Shihao Xia, Yu Liang, Linhai Song, and Hong Hu. 2022. Who Goes First? Detecting Go Concurrency Bugs via Message Reordering. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. ACM, 888–902. https://doi.org/10.1145/3503222.3507753

[22] Ziheng Liu, Shuofei Zhu, Boqin Qin, Hao Chen, and Linhai Song. 2021. Automatically detecting and fixing concurrency bugs in go software systems. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 616–629. https://doi.org/10.1145/3445814.3446756

[23] Kenneth L. McMillan. 2003. Interpolation and SAT-Based Model Checking. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2725)*, Warren A. Hunt Jr. and Fabio Somenzi (Eds.). Springer, 1–13. https://doi.org/10.1007/978-3-540-45069-6_1

[24] Jan Midtgaard, Flemming Nielson, and Hanne Riis Nielson. 2018. Process-Local Static Analysis of Synchronous Processes. In *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11002)*, Andreas Podelski (Ed.). Springer, 284–305. https://doi.org/10.1007/978-3-319-99725-4_18

[25] Marvin L. Minsky. 1967. *Computation: finite and infinite machines*. Prentice-Hall, Inc., USA.

[26] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

[27] Nicholas Ng and Nobuko Yoshida. 2016. Static deadlock detection for concurrent go by global session graph synthesis. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 174–184. https://doi.org/10.1145/2892208.2892232

[28] Saswat Padhi, Rahul Sharma, and Todd D. Millstein. 2016. Data-driven precondition inference with learned features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 42–56. https://doi.org/10.1145/2908080.2908099

[29] Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. 2001. Automatic Deductive Verification with Invisible Invariants. In *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2031)*, Tiziana Margaria and Wang Yi (Eds.). Springer, 82–97. https://doi.org/10.1007/3-540-45319-9_7

[30] Georgian-Vlad Saioc, Dmitriy Shirchenko, and Milind Chabbi. 2024. Unveiling and Vanquishing Goroutine Leaks in Enterprise Microservices: A Dynamic Analysis Approach. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2024, Edinburgh, United Kingdom, March 2-6, 2024*, Tobias Grosser, Christophe Dubach, Michel Steuwer, Jingling Xue, Guilherme Ottoni, and ernando Magno Quintão Pereira (Eds.). IEEE, 411–422. https://doi.org/10.1109/CGO57630.2024.10444835

[31] Georgian-Vlad Saioc and Milind Chabbi. 2022. LeakProf: Featherlight In-Production Goroutine Leak Detection. https://www.uber.com/en-GB/blog/leakprof-featherlight-in-production-goroutine-leak-detection/.

[32] Georgian-Vlad Saioc, Julien Lange, and Anders Møller. 2024. *VladSaioc/oopsla-24-artifact: OOPSLA 24 Artifact*. https://doi.org/10.5281/zenodo.13825844

[33] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. 2004. Constraint-Based Linear-Relations Analysis. In *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3148)*, Roberto Giacobazzi (Ed.). Springer, 53–68. https://doi.org/10.1007/978-3-540-27864-1_7

[34] Kai Stadtmüller, Martin Sulzmann, and Peter Thiemann. 2016. Static Trace-Based Deadlock Analysis for Synchronous Mini-Go. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10017)*, Atsushi Igarashi (Ed.). 116–136. https://doi.org/10.1007/978-3-319-47958-3_7

[35] Martin Sulzmann and Kai Stadtmüller. 2018. Two-Phase Dynamic Analysis of Message-Passing Go Programs Based on Vector Clocks. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018*, David Sabel and Peter Thiemann (Eds.). ACM, 22:1–22:13. https://doi.org/10.1145/3236950.3236959

[36] The Go Team. 2010. Share Memory by Communicating. https://go.dev/doc/effective_go.

[37] The Go Team. 2024. Go. https://go.dev/.

[38] Oskar Haarklou Veileborg, Georgian-Vlad Saioc, and Anders Møller. 2022. Detecting Blocking Errors in Go Programs using Localized Abstract Interpretation. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 32:1–32:12. https://doi.org/10.1145/3551349.3561154

[39] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller. 2021. Gobra: Modular Specification and Verification of Go Programs. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12759)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 367–379. https://doi.org/10.1007/978-3-030-81685-8_17

[40] Ting Yuan, Guangwei Li, Jie Lu, Chen Liu, Lian Li, and Jingling Xue. 2021. GoBench: A Benchmark Suite of Real-World Go Concurrency Bugs. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 187–199. https://doi.org/10.1109/CGO51591.2021.9370317

Received 2024-04-05; accepted 2024-08-18