Gobra: Modular Specification and Verification of Go Programs

Felix A. Wolf[1], Linard Arquint[1], Martin Clochard[1], Wytse Oortwijn[2], João C. Pereira[1(✉)], and Peter Müller[1]

[1] Department of Computer Science, ETH Zurich, Zurich, Switzerland
{felix.wolf,linard.arquint,martin.clochard, joao.pereira,peter.mueller}@inf.ethz.ch
[2] ESI (TNO), Eindhoven, The Netherlands
wytse.oortwijn@tno.nl

Abstract. Go is an increasingly-popular systems programming language targeting, especially, concurrent and distributed systems. Go differentiates itself from other imperative languages by offering structural subtyping and lightweight concurrency through goroutines with message-passing communication. This combination of features poses interesting challenges for static verification, most prominently the combination of a mutable heap and advanced concurrency primitives.

We present Gobra, a modular, deductive program verifier for Go that proves memory safety, crash safety, data-race freedom, and user-provided specifications. Gobra is based on separation logic and supports a large subset of Go. Its implementation translates an annotated Go program into the Viper intermediate verification language and uses an existing SMT-based verification backend to compute and discharge proof obligations.

Keywords: Separation logic · Program logics · Channel-based concurrency · Interfaces · Deductive verification · Automated verification

1 Introduction

Go is an increasingly popular systems programming language targeting, especially, concurrent and distributed systems such as web applications. It combines standard features of imperative languages, such as mutable heap data structures, with less common concepts, such as structural subtyping and lightweight concurrency through goroutines with message-passing communication.

This combination of features poses interesting challenges for static verification, most prominently the combination of a mutable heap and advanced concurrency primitives. Prior research on Go verification handles some of these features, but not their combination. For instance, Lange et al. [14,15] verify safety and

A. Silva and K. R. M. Leino (Eds.) CAV 2021, LNCS 12759, pp. 367–379, 2021.
https://doi.org/10.1007/978-3-030-81685-8_17

liveness of Go's message-passing, but do not consider functional properties about the heap state, whereas Perennial [4] supports heap data structures, but neither channels nor interfaces.

We present Gobra, an automated, modular verifier for heap-manipulating, concurrent Go programs. Gobra supports a large subset of Go, including Go's interfaces and primitive data structures, both of which have not been fully supported in previous work. Gobra verifies memory safety, crash safety, data-race freedom, and user-provided specifications. It takes as input a Go program annotated with assertions such as pre and postconditions and loop invariants. Verification proceeds by encoding the annotated programs into the intermediate verification language Viper [17] and then applying an existing SMT-based verifier. In case verification fails, Gobra reports at the level of the Go program which assertions it could not verify.

Gobra's assertion language builds on established concepts: Gobra uses separation logic style permissions [19] to reason locally about heap data structures. It supports recursive predicates and specification methods to abstract over (possibly unbounded) data structures and their contents. In particular, Gobra has first-class predicates that enable a natural specification of concurrency primitives, for instance, to parameterize a lock by an invariant.

Gobra is intended for the verification of substantial, real-world code, and is currently used to verify the Go implementation of the SCION internet architecture [23]. Our tool paper makes the following technical contributions:

(1) We present the Gobra tool, an automated modular verifier for annotated Go programs. Our evaluation demonstrates that Gobra can verify non-trivial examples with good performance. Our artifact is available online [21].
(2) We define a specification language for functional properties of Go programs. Our specification language provides a consistent abstraction at the level of Go and does not leak details of the underlying encoding.
(3) We present the first specification and verification technique for structural subtyping via Go interfaces.
(4) Our Viper encoding supports, among other features, Go's broad range of built-in data types, such as slices and channels. A lightweight annotation allows it to apply separation logic to reason soundly about addressable memory locations, but use a more efficient encoding for others.

Outline. We demonstrate key features of Gobra on examples (Sect. 2), give an overview of the encoding into Viper (Sect. 3), and provide an experimental evaluation of Gobra (Sect. 4). Lastly, Sect. 5 discusses related work and concludes.

2 Gobra in a Nutshell

This section illustrates Gobra's specification language on simple examples and shows how we handle interfaces and concurrency.

2.1 Basics

Gobra uses a variant of separation logic [19] in order to reason about mutable heap data structures and concurrency. Separation logics associate an access permission with each heap location. Access permissions are held by method executions and transferred between methods upon call and return. A method may access a location only if it holds the associated permission. Permission to a shared location `v` is denoted in Gobra by `acc(&v)`, which is analogous to separation logic's $v \mapsto \_$. Gobra provides an expressive permission model supporting fractional permissions [3] to allow concurrent read accesses while still ensuring exclusive writes, (recursive) predicates to denote access to unbounded data structures, and quantified permissions (also called iterated separating conjunction) to express permissions to random-access data structures such as arrays and slices.

```
1  requires ∀ k int :: 0 ≤ k < len(s) ⟹ acc(&s[k])
2  ensures  ∀ k int :: 0 ≤ k < len(s) ⟹ acc(&s[k])
3  ensures  ∀ k int :: 0 ≤ k < len(s) ⟹ s[k] == old(s[k]) + n
4  func incr (s []int, n int) {
5    invariant 0 ≤ i ≤ len(s)
6    invariant ∀ k int :: 0 ≤ k < len(s) ⟹ acc(&s[k])
7    invariant ∀ k int :: i ≤ k < len(s) ⟹ s[k] == old(s[k])
8    invariant ∀ k int :: 0 ≤ k < i ⟹ s[k] == old(s[k]) + n
9    for i := 0; i < len(s); i += 1 {
10     s[i] = s[i] + n
11   }
12 }
```

Fig. 1. A simple Gobra example showing method and loop contracts.

The example in Fig. 1 illustrates the use of permissions. Method `incr` increases all elements of a given slice `s` by an amount `n`. (Slices are data types that can intuitively be seen as shared arrays of variable length.) The method requires permission to all slice elements (via its precondition) and returns them to the caller (via its first postcondition).

Functional properties are expressed via standard assertions, which include side-effect free Go expressions (including calls to pure methods, as we explain below) as well as universal quantification and old-expressions to refer to the value an expression had in the pre-state of a method. In our example, the second postcondition uses these assertions to express the functional behavior of the method. The loop invariants are analogous to the method contracts and are needed for verification.

In Go, any memory location can either be *shared* or *exclusive*. Shared locations reside on the heap and can, thus, be accessed by multiple methods and threads; reasoning about shared locations requires permissions to ensure race freedom and to enable framing, i.e., preserving information across heap changes. On the other hand, exclusive locations are accessed exclusively by one method execution and may be allocated on the stack; they can be reasoned about as local variables. The Go compiler determines automatically whether a location is

shared or exclusive, for instance by determining whether its address is taken at some point of the execution. To make verification independent of a particular compiler analysis, Gobra requires shared locations to be decorated with an extra annotation `@` at the declaration point, as illustrated by the following client of `incr`:

```
1  a@ := [4]int { 1, 2, 4, 8 }
2  incr(a[2:], 2)
3  assert a == [4]int { 1, 2, 6, 10 }
```

The first line declares a Go array `a` of fixed length 4, with values 1, 2, 4, and 8. This array is sliced on line 2 using the syntax `a[2:]`, thereby omitting the first two elements of `a` from the created slice. Since `a` is used in a context in which it is sliced, it is a shared location, which is made explicit via the `@` annotation. Consequently, the array creation will produce permissions to the array elements, which are required by `incr`'s precondition. Omitting the `@` annotation will cause a verification error.

2.2 Interfaces

Go supports polymorphism through *interfaces*, named sets of method signatures. Subtyping for interfaces is structural: a type implements an interface iff every method of the interface is implemented by the type. The subtype relationship is determined by the type checker, without any declarations from the programmer[1].

Calls on an interface value are dynamically dispatched. In settings with nominal subtyping, dynamic dispatch is handled by proving behavioral subtyping [16]: each subtype declaration requires a proof that the specifications of subtype methods refine the specifications of the corresponding supertype methods. Since structural subtypes are not declared explicitly, we adapt this approach as follows.

Whenever a Go program assigns a value to a variable of an interface type, Gobra requires an *implementation proof*, that is, a proof that each method of the subtype satisfies the specification of the corresponding method in the interface. Implementation proofs are inferred automatically by Gobra in simple cases; user-provided implementation proofs are required especially when they include ghost operations, for instance, to manipulate predicates.

The example in Fig. 2 illustrates this approach. Interface `stream` (lines 1–8) declares an interface with two methods, `hasNext` and `next`. The latter may return values of an arbitrary type, which is denoted by an empty interface. Since interfaces do not contain an implementation, their specification must be fully abstract. To this end, `stream` introduces an abstract predicate `memory`, whose definition is provided by the subtypes of the interface. The functional behavior of interface methods can be expressed in terms of pure (that is, side-effect free) abstract methods, here, `hasNext`, which will also be defined in subtypes.

Next, lines 10–16 show an implementation of the interface in the form of a counter. The counter has a current `f` and maximum `max` value. As long as the

[1] For the sake of simplicity, we omit *embeddings*, Go's construct for delegation; an extension is straightforward.

```
1   type stream interface{
2     pred memory()
3     requires acc(memory(), _) // arbitrary fraction of memory()
4     pure hasNext() bool
5     requires memory() && hasNext()
6     ensures  memory()
7     next() interface{}
8   }
9
10  type counter struct{ f int; max int }
11  requires acc(&x.f, _) && acc(&x.max, _)
12  pure func (x *counter) hasNext() bool { return x.f < x.max }
13  requires acc(&x.f) && acc(&x.max, 1/2) && x.hasNext()
14  ensures  acc(&x.f) && acc(&x.max, 1/2) && x.f == old(x.f)+1
15  ensures  typeOf(y) == int && y.(int) == old(x.f)
16  func (x *counter) next() (y interface{}) { x.f++;return x.f-1 }
17
18  pred (x *counter) memory() { acc(&x.f) && acc(&x.max) }
19  (*counter) implements stream {
20    pure (x *counter) hasNext_PROOF() bool {
21      return unfolding acc(x.memory(), _) in x.hasNext()
22    }
23    (x *counter) next_PROOF() (res interface{}) { ... }
24  }
```

Fig. 2. An interface specification for a stream (lines 1–8) together with an implementation (lines 10–16) and an implementation proof (lines 18–24). We write `acc(p, _)` to denote an arbitrary, positive amount of predicate `p`, and simply `p` for `acc(p, 1/1)`. At line 14, the fractional permission to `&x.max` entails that `x.max` is not modified.

maximum value is not reached, `next` will increase the current value. At line 16, an integer can be assigned to the empty interface since behavioral subtyping holds trivially. The specification at line 15 expresses that the returned interface value contains an integer with the old value of the `f` field.

The counter implementation is completely independent of the `stream` interface. Their connection is established only in the implementation proof (lines 18–24). This proof defines the `memory` predicate from the `stream` interface for receivers of type `counter` (line 18). Moreover, an implementation proof verifies that the specification of each method implementation refines the specification of the corresponding interface method. This proof checks that, assuming the precondition of an interface method, a call to the implementation method with identical arguments establishes the postcondition of the interface method. This format is enforced syntactically and permits ghost operations before and after the call to manipulate predicates. For instance, the proof on line 21 for `hasNext` temporarily unfolds the `memory` predicate to obtain permission to `x`, which is required by the implementation method, and conversely after the call.

Implementation proofs can be written explicitly, imported from other packages, and also inferred automatically when no explicit proof exists in the current scope. Currently, Gobra does not infer ghost operations such as the `unfolding` on line 21; our experiments suggest that already simple heuristics can deal with

many cases occurring in practice. For instance, many implementation proofs we have encountered follow the same pattern: First, the interface predicate instances of the precondition are unfolded. Second, the implementation method is called. Lastly, the interface predicate instances of the postcondition are folded. This pattern can be generated automatically to alleviate the annotation burden.

Gobra's implementation proofs enable one to reason about interfaces without enforcing subtype declarations in either the interface or the declaration, which would defeat the purpose of structural subtyping. This solution allows one to reason about dynamically-dispatched calls. For instance, the following code snippet verifies in Gobra:

```
1  x := &counter{0, 50}
2  var y stream = x
3  fold y.memory()
4  var z interface{} = y.next()
```

In particular, Gobra is able to determine that `next`'s precondition `hasNext()` holds because `y.hasNext()` is equal to `x.hasNext()`, and the latter follows from the definition of `hasNext` (line 12) and the initial value of `x.f`. This intuitive reasoning is enabled by an intricate underlying encoding, which is not exposed to users. Users do not have to know how interface predicates are encoded and can treat interface predicates the same as any other separation-logic predicate.

2.3 Concurrency

Go supports concurrency through *goroutines*, lightweight threads started by prefixing a method call with the `go` keyword. Go offers the usual synchronization primitives, but goroutines idiomatically synchronize via *channels*. Buffered channels provide asynchronous communication, where sending a message blocks only when the buffer is full. Unbuffered channels offer rendez-vouz communication.

Gobra enables verification of concurrent programs by associating Go's synchronization primitives with predicates that do not only express properties of data but also express how permissions to shared memory get transferred between threads. For instance, lock invariants may include properties as well as permissions to the data protected by the lock, and channel invariants include properties and permissions of the data sent over a channel. These invariants are specified via ghost operations when the synchronization primitive is initialized.

Figure 3 illustrates Gobra's concurrency support using an excerpt from a parallel search-and-replace algorithm (see the full paper [22] for the complete example). Method `searchAndReplace` spawns a series of worker threads and then sends each of them a chunk of the input slice to process. The worker threads are joined via a wait group `wg`. Method `worker` implements the worker threads.

Gobra associates channels (like `c` in the example) with a predicate to specify properties and permissions of the sent data. The call `c.Init(...)` on line 10 takes this predicate as an argument. As expressed on line 2, it includes permissions to the chunk a worker operates on. For synchronous channels, an additional predicate can specify permissions transferred in the opposite direction, from the

```
1  pred messagePerm(wg *sync.WaitGroup, chunk []int, x, y int) {
2    ( ∀ i int :: 0 ≤ i < len(chunk) ⟹ acc(&chunk[i]) ) && ...
3  }
4  requires ∀ i int :: 0 ≤ i < len(s) ⟹ acc(&s[i])
5  func searchAndReplace(s []int, x, y int) {
6    var wg@ sync.WaitGroup
7    ghost wg.Init()
8    c := make(chan []int,4)
9    // predicate-name{..., _, ...} is syntax for partial application
10   ghost c.Init(messagePerm{&wg, _, x, y})
11   // Spawn workers
12   invariant acc(c.RecvChannel(), _)
13   invariant c.RecvGotPerm() == messagePerm{&wg, _, x, y}
14   for i := 0; i < numOfWorkers; i++ { go worker(c, wg, x, y) }
15   // Split slice into chunks, which are sent to workers
16   invariant c.SendChannel()
17   invariant c.SendGivenPerm() == messagePerm{&wg, _, x, y}
18   invariant ∀ i int :: offset ≤ i < len(s) ⟹ acc(&s[i])
19   invariant ... // constraints on offset and nextOffset
20   for offset := 0; offset != len(s); offset = nextOffset {
21     nextOffset = ...
22     wg.Add(1)
23     fold messagePerm{&wg, _, x, y}(s[offset:nextOffset])
24     c <- s[offset:nextOffset]
25   }
26   wg.Wait()
27 }
28 requires acc(c.RecvChannel(), _)
29 requires c.RecvGotPerm() == messagePerm{wg, _, x, y};
30 func worker(c <- chan []int, wg *sync.WaitGroup, x, y int) {
31   invariant acc(c.RecvChannel(), _)
32   invariant c.RecvGotPerm() == messagePerm{wg, _, x, y};
33   invariant ok ⟹ messagePerm{wg, _, x, y}(chunk)
34   for chunk, ok := <- c; ok; chunk, ok = <-c {
35     unfold messagePerm{wg, _, x, y}(chunk)
36     ... // replace x with y in chunk
37     wg.Done() // same as wg.Add(-1)
38   }
39 }
```

Fig. 3. Excerpt showing goroutines, channels, and wait groups. The code spawns workers (line 14), sends slice chunks through a channel to the workers (line 24), and then waits on a wait group (line 26). A worker receives a chunk (line 34), processes it, and then notifies the wait group (line 37). For the sake of simplicity, some details were omitted.

receiver to the sender. Initializing a channel also creates send and receive permissions for the channel, which are used to control which threads may access it. In our example, we transfer a fraction of the receive permission to each worker (line 28).

The workers receive permission to the chunk they operate on via a message sent on line 24 and received on line 34. The transfer back is orchestrated through a wait group, which implements an abstract shared counter. Wait groups are used as follows: The main thread adds to the counter the number of units of work to be done in spawned goroutines (line 22). Each spawned goroutine decreases the counter each time a unit of work is done (via a call to `Done`, line 37). The

master can await the counter to reach 0 via a call to `Wait` (line 26). Gobra uses dedicated permissions to express the obligation of a thread to perform units of work before decreasing the counter; each time this happens, permissions are transferred to the wait group and, eventually to the main thread calling `Wait`. We omit the details here for brevity.

In our example, this mechanism allows the main thread to recover the permissions to the entire slice once the workers have terminated. The example in Fig. 3 illustrates only the permission aspect of the verification. Functional correctness can be verified easily based on the explained machinery, by specifying a stronger channel invariant that includes the work obligation for each worker. We omit the details here, but see the full paper [22] for the complete example.

3 Encoding

Gobra encodes an annotated Go program into a Viper program verifying only if the input program is correct. Many features of Gobra are also present in Viper, making parts of the encoding straightforward. For instance, methods, pure methods, and predicates are encoded to their Viper counterpart. Viper's permission model (including fractions, wildcards, and quantifiers) is similar to Gobra's, but memory is represented differently; Viper's heap is object-based, where each object contains all declared fields. Viper's fields store primitive values (including references). To encode Go's compound values such as structs, arrays, slices, and interface values, we use Viper's mechanism to declare mathematical types (such as tuples) using uninterpreted types, uninterpreted functions, and appropriate axioms. Exclusive Go values are directly represented using these mathematical types. For shared values, there is an indirection via the Viper heap to permit aliasing and apply permission-based reasoning.

Interfaces. As explained in Sect. 2.2, our treatment of Go interfaces relies on interface predicates, specification methods, and implementation proofs. We explain how we handle the former two here; based on this encoding, the encoding of implementation proofs is analogous to methods.

Intuitively, we encode interface predicates as a case split over all possible implementations. All implementations not present in the current scope are subsumed by an abstract default case. Consequently, adding an implementation does not invalidate existing proofs, which enables modular reasoning. The predicate for the `stream` example (Fig. 2) is encoded as follows:

```
predicate memory(x: ⟦interface{}⟧) {
  ⟦typeOf(x) == *counter⟧ ? ⟦acc(x.(*counter))⟧ : unknownMemory(x)
}
predicate unknownMemory(x: ⟦interface{}⟧)

function hasNext(x: ⟦interface{}⟧) returns (y: ⟦bool⟧)
  req ⟦acc(x.memory(), _)⟧
  ens ⟦typeOf(x) == *counter⟧ ⟹ y == hasNext_PROOF(⟦x.(*counter)⟧)
```

The body of the predicate branches on the dynamic type of `x`, with a single case for the (only) given implementation. The abstract predicate `unknownMemory` encodes the default case. The encoding of pure methods such as `hasNext` uses an analogous case split, but uses $\texttt{hasNext}_{\text{PROOF}}$, which is part of the implementation proof (Fig. 2 line 20) and couples the interface and implementation method. Our encoding of interface predicates is an instance of an *abstract predicate family* [18]. For Go, we have crafted a variant that is well-suited for implementation proofs, pure interface methods, and structural subtyping.

First-Class Predicates. Our support for concurrency uses first-class predicates, for instance, to specify channel invariants (see Sect. 2.3). We encode first-class predicate values as mathematical types, using defunctionalization. Predicate instances are represented by abstract predicates that take the predicate value as an argument. First-class predicates enable us to use library stubs to support concurrency primitives such as mutexes and wait groups. These stubs allow us to encode the use of these concurrency primitives via standard method calls. Go's native channel operations are represented analogously.

4 Implementation and Evaluation

The Gobra implementation consists of a parser and type checker for annotated Go programs and a translation of those programs into the Viper intermediate verification language. The resulting Viper program is verified using Viper's symbolic execution backend, which in turn uses the Z3 SMT solver [7]. Verification errors are translated back to the Go level, such that users are not exposed to the internal encodings. Users never have to inspect the encoding. Error messages contain the failing assertion and a reason describing why the assertion failed. Gobra's test suite contains 407 verification tests (with and without errors) with a total of 10'030 LOCs (Go code and annotations) that take 14.9 min to verify.

We evaluated Gobra on 14 interesting verification problems, which include well-known algorithms and data structures, and cover Go's main features, such as interfaces (Examples 7–9) and concurrency primitives (Examples 13 and 14), including goroutines, mutexes, wait groups, and channels. For each example, Gobra verifies memory safety and functional correctness properties. To assess Gobra's performance on failing verifications, we have additionally constructed two incorrect variations of each example, one with a seeded error in the specification and one in the implementation.

All experiments were executed on a warmed-up JVM on a MacBook Pro with a 2.3 GHz 8-Core Intel Core i9 CPU and 32 GB of RAM, running macOS 11.1 and OpenJDK 11. For each experiment, we measured its verification time using Viper's symbolic execution backend and averaged the duration of twelve executions, excluding the slowest and fastest outlier.

Figure 4 summarizes the results, including the required annotations and verification times for the three variants of each example. The annotation overhead

#	Example	LOC / Spec.	Viper LOC	T [s]	$T_{\text{spec error}}$ [s]	$T_{\text{impl error}}$ [s]
1	binary search tree	125 / 140	632	10.88	10.50	11.67
2	dutchflag	22 / 16	142	2.02	1.78	1.88
3	heapsort	47 / 93	271	16.72	19.30	15.23
4	dense and sparse matrix	69 / 62	326	10.46	10.55	10.06
5	binary tree	59 / 20	217	2.09	2.08	2.11
6	running ex. (Fig. 1)	10 / 11	164	1.71	1.70	1.70
7	running ex. (Fig. 2)	24 / 16	186	1.04	0.98	1.01
8	list of interfaces	46 / 27	219	1.45	1.41	1.54
9	visitor pattern	76 / 30	475	4.38	4.22	5.45
10	zune	31 / 12	141	1.08	1.07	1.06
11	relaxed prefix	25 / 36	158	7.08	5.36	4.19
12	pair insertion sort	50 / 105	353	15.55	12.64	13.96
13	parallel search replace	35 / 94	565	53.18	51.97	61.54
14	parallel sum	31 / 98	527	58.39	50.25	57.69

Fig. 4. Experimental results. For each experiment, we list the number of lines of Go code (LOC), number of lines of specification and proof annotations (Spec), and the average verification time in seconds for correct examples (T), errors in the specification ($T_{\text{spec error}}$), and errors in the implementation ($T_{\text{impl error}}$). A line containing both, code and annotations, is counted as one line of Go code and one line of annotation.

ranges between 0.3 and 3.1 lines of annotations per line of code, which is typical for SMT-based deductive verifiers. Verification times range between a second and a minute per example. The verification times are significantly higher when the verified code uses concurrency features; these examples require quantitatively more and more-complex specifications, which complicates reasoning. Lastly, there is hardly any difference between successful and failed verification attempts. Consistent performance is crucial when verifiers are used interactively, where users run them frequently, especially on programs that do not yet verify.

5 Related Work and Conclusion

Besides Gobra, we are aware of two other verification approaches for Go. Perennial [4] reasons about concurrent, crash-safe systems. Their core techniques are an extension to the Iris framework [13] and independent of Go. They connect their theory to Go programs with Goose, a shallow embedding of Go into Coq [5], which proves that Go code complies with a given transition system. In contrast to Gobra, Perennial does not support core Go features such as channels and interfaces.

Several prior works [9,14,15] infer behavioral types [12] to reason about Go's channel-based message passing. After they infer behavioral types for a given program, they check safety and liveness properties on the inferred types, using model checkers such as mCRL2 [6]. Some works use additional analyses to strengthen the provided guarantees. Lange et al. [15] add a termination analysis to enable one to verify unbounded properties under certain conditions. Gabet

and Yoshida [9] extend this work by inferring behavioral types on shared variables and locks to additionally reason about data-race freedom, lock safety, and lock liveness. The approaches by Lange et al. [15] and Gabet and Yoshida [9] are vastly different from Gobra. They do not verify code contracts, but instead verify global properties such as deadlock and data-race freedom. Their automation is high and annotation overhead minimal, but their analyses are not modular and do not verify functional properties of code. Furthermore, they do not verify properties about the state of the heap.

There are some prior works that can handle channel-based concurrency and heap-manipulating programs, but these do not apply directly to Go. Villard et al. [20] introduce a powerful contract mechanism to specify protocols that channels must adhere to. Their channel specification language is more expressive than the one presented in this paper. Their contracts are finite state machines and thus can have multiple phases. However, their channels are always shared between two peers whereas Go supports more advanced concurrency patterns where both channel endpoints are shared between an unbounded number of peers. Actris [10,11] is a concurrent separation logic built on top of the Iris framework to reason about session types in an interactive theorem prover. Actris can go beyond two peers, but to do so, it requires a memory model that is incompatible with Go's memory model. Actris models the sharing of channel endpoints via Iris' ghost locks, which to our knowledge, implies sequentialization of sends, and dually receives, which is not guaranteed by Go's memory model.

Gobra's verification logic and encoding into Viper have been inspired by several other Viper-based verifiers, such as Nagini [8] for Python, Prusti [1] for Rust, and VerCors [2] for Java. None of these verifiers address the Go-specific features that Gobra supports.

Conclusion. We introduced Gobra, the first modular verifier for Go that supports reasoning about a crucial aspect of the language: the combination of channel-based concurrency and heap-manipulating constructs. Moreover, Gobra is the first verifier to support Go's version of interfaces and structural subtyping. In future work, we will expand the properties that can be verified with Gobra, in particular to liveness and hyper-properties. Furthermore, we are applying Gobra to verify the implementation of a full-fledged network router [23]. Gobra is hosted on Github at https://github.com/viperproject/gobra.

Acknowledgements. This project has received funding from the European Union's Horizon 2020 research and innovation program within the framework of the NGI-POINTER Project funded under grant agreement No. 871528.

References

1. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging rust types for modular specification and verification. In: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), vol. 3, pp. 147:1–147:30. ACM (2019)

2. Blom, S., Huisman, M.: The VerCors tool for verification of concurrent programs. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 127–131. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_9
3. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44898-5_4
4. Chajed, T., Tassarotti, J., Kaashoek, M.F., Zeldovich, N.: Verifying concurrent, crash-safe systems with Perennial. In: SOSP, pp. 243–258. ACM (2019)
5. Coq consortium, T.: The Coq proof assistant. https://coq.inria.fr/
6. Cranen, S., et al.: An overview of the mCRL2 toolset and its recent advances. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 199–213. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_15
7. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
8. Eilers, M., Müller, P.: Nagini: a static verifier for Python. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018, Part I. LNCS, vol. 10981, pp. 596–603. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_33
9. Gabet, J., Yoshida, N.: Static race detection and mutex safety and liveness for go programs (extended version) (2020)
10. Hinrichsen, J.K., Bengtson, J., Krebbers, R.: Actris: session-type based reasoning in separation logic. Proc. ACM Program. Lang. **4**(POPL), 1–30 (2019)
11. Hinrichsen, J.K., Bengtson, J., Krebbers, R.: Actris 2.0: Asynchronous session-type based reasoning in separation logic. arXiv preprint arXiv:2010.15030 (2020)
12. Hüttel, H., et al.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 1–36 (2016)
13. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: a modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, e20 (2018)
14. Lange, J., Ng, N., Toninho, B., Yoshida, N.: Fencing off Go: liveness and safety for channel-based programming, pp. 748–761 (2017)
15. Lange, J., Ng, N., Toninho, B., Yoshida, N.: A static verification framework for message passing in Go using behavioural types. In: ICSE, pp. 1137–1148. ACM (2018)
16. Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6), 1811–1841 (1994)
17. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
18. Parkinson, M., Bierman, G.: Separation logic and abstraction. ACM SIGPLAN Not. **40**(1), 247–258 (2005)
19. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: LICS, pp. 55–74. IEEE Computer Society (2002)
20. Villard, J., Lozes, É., Calcagno, C.: Proving copyless message passing. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 194–209. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10672-9_15
21. Wolf, F.A., Arquint, L., Clochard, M., Oortwijn, W., Pereira, J.C., Müller, P.: Gobra: Modular Specification and Verification of Go Programs (2021). https://doi.org/10.5281/zenodo.4716664

22. Wolf, F.A., Arquint, L., Clochard, M., Oortwijn, W., Pereira, J.C., Müller, P.: Gobra: Modular specification and verification of go programs (extended version). CoRR arXiv:2105.13840 (2021)
23. Zhang, X., Hsiao, H.C., Hasker, G., Chan, H., Perrig, A., Andersen, D.G.: Scion: scalability, control, and isolation on next-generation networks. In: IEEE Symposium on Security and Privacy, pp. 212–227. IEEE (2011)