GOAT: Automated Concurrency Analysis and Debugging Tool for Go
Paper Type: Tool / Benchmark

***Abstract*—The use of increasing levels of parallelism and concurrency in the system design—especially in a feature-rich language such as Go—demands effective concurrency debugging techniques that are easy to deploy in practice. We present GOAT, a combined static and dynamic concurrency testing and analysis tool that facilitates the process of debugging for real-world programs. Key ideas in GOAT include 1) automated dynamic tracing to capture the behavior of concurrency primitives, 2) systematic schedule space exploration to accelerate the bug occurrence and 3) deadlock detection with supplementary visualizations and reports. We also propose a set of coverage requirements that characterize the dynamic behavior of concurrency primitives and provide metrics to measure the quality of tests. Evaluation of GOAT on 68 curated real-world bug scenarios demonstrates that GOAT is significantly effective in detecting rare bugs, and its schedule perturbation method based on schedule yielding detects these bugs with less than three yields. These results together with the ease of deploying GOAT on real-world Go programs hold significant promise in the field-debugging of Go programs.**

***Index Terms*—golang, concurrency, testing coverage analysis**

I. INTRODUCTION

Go [1] is a statically typed language initially developed by Google. It employs channel-based Hoare's Communicating Sequential Processes (CSP) [2] semantics in its core and provides a productivity-enhancing environment for concurrent programming. Go enjoys accelerating acceptance in a wide variety of communities including container software systems [3], [4], distributed key-value databases [5], [6], and web server libraries [7]. It involves shared memory, message passing, non-deterministic message reception and selection, dynamic process creation, and programming styles that tend to create thousands of *goroutines* (i.e., application-level threads) and discard them to be garbage collected when they reach their final state. The combination of these features is well known for Go's popularity, yet they also make Go challenging to debug. Our work is especially relevant considering that there are no widely practical tools for debugging concurrent Go; even well-curated concurrency bug benchmark suites are only just now beginning to appear [8], [9].

In general, concurrent bugs are notoriously difficult to find and reproduce due to the non-deterministic choices that the scheduler makes during execution. In Go, constructs like *select* and buffered channels entangle the process of debugging by introducing extra randomness to the dynamic behavior of the program. Recent static [10]–[13] and dynamic [14]–[18] techniques have been proposed to address these challenges. GoBench [9] gathers a collection of real concurrency bugs (GoReal) and simplified bug kernels (GoKer) from the top 9 open-source projects written in Go and evaluated the effectiveness of such techniques in detecting the bug collection. Although static methods are proved to be rigorously effective in detecting flaws in small programs, they are not practical for realistic programs and often produce false positives. On the other hand, dynamic analysis approaches cover a more significant subset of real-world programs by constructing and analyzing an *execution model*. However, they focus on a specific class of bugs based on the symptom or cause of the bug. Also, for large codebases with thousands of LOC, it is non-trivial to capture an accurate dynamic execution model using source instrumentation or source-to-source translation. Furthermore, our experiments (section IV) observe that some buggy programs take more than 1,000 runs under different schedules before the bug is hit. Concurrent testing methods [19] are proposed to complement static and dynamic approaches in tackling challenges of concurrent debugging. To the best of our knowledge, there exist no such testing methods applicable to Go.

We implemented GOAT (**Go A**nalysis and **T**esting), a debugging framework for concurrent Go applications to address this lack. GOAT (figure 1) combines static and dynamic approaches to automatically analyze the behavior of concurrent components and facilitate the process of testing and debugging Go applications. Several classic ideas from literature are combined with novel ones to support modern concepts of Go in GOAT, which pursues three primary objectives:

Objective 1: *Accurate Dynamic Execution Modeling*— In order to study the behavior of concurrent components and track the state of the program during execution, a dynamic execution model has to be constructed and compared against a predefined model (e.g., formally defined specifications or the developer's mental representation of the program). Since a bug might occur at various levels of abstraction, *whole-program dynamic tracing* provides a practical and uniform way to track multiple facets of the program during execution [20]. We have enhanced the built-in tracing mechanism of Go to capture the dynamic behavior of concurrency primitives in the form of a *sequence of events*, namely *execution concurrency trace* or ECT. Each event in ECT represents an *action* that corresponds to exactly one statement in the source code. An ECT provides a detailed model of how a concurrent program behaves dynamically and assists debugging procedures (e.g., bug detection, root-cause analysis, execution visualization).

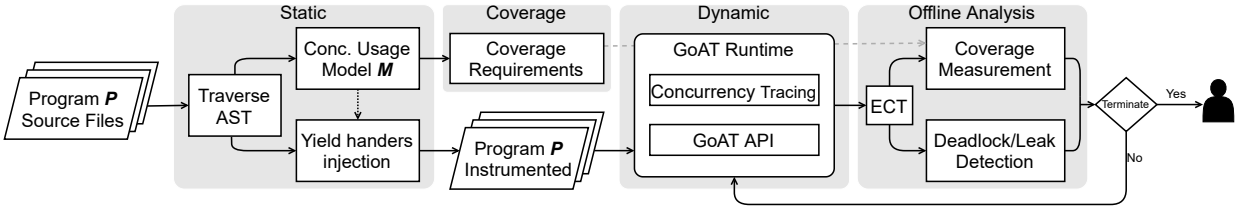


Figure 1: GOAT Overview

Our experiments show that by replaying the program's ECT, GOAT detects all blocking bugs of GoKer [9] many of which are undetected by existing debugging tools.

Objective 2: *Systematic Exploration of Schedule-Space*— Since the scheduler's non-deterministic behavior is the primary reason for Heisenbugs (i.e., errors that are uncommon to occur and hard to reproduce), these bugs may not manifest during conventional testing. By adopting ideas from *systematic concurrency testing* approaches [21]–[32], we perturb the native scheduler of Go to explore the unconventional but feasible execution interleaving. First, we statically identify the source location of concurrency primitive usages in a given program. We then inject handlers these locations to randomly (with a certain probability and within a bound) decide if the current goroutine should continue executing or *yield* to other goroutines to execute first. Such yields change the blocking behavior of the program within the space of feasible states and exercise untested interleavings, consequently heighten the propensity for bug detection. The results of our experiments indicate that just a few random schedule perturbations can accelerate the exposure of rare bugs.

Objective 3: *Testing Quality Measurement*— A test suite's thoroughness is often judged by the coverage of certain aspects of the software, such as its source-code statements (a higher statement coverage indicates more thorough testing). In the context of concurrent software, exisiting coverage metrics [29], [30], [33], [34] characterize (quantify) the behavior of concurrency primitives which enables the quality measurement of schedule-space exploration. Such characterizations involve defining an initial set of requirements and a method for assessing whether or not those requirements are met during testing. Since Go combines traditional synchronization and serialization primitives (mutex, conditional variables) with message-passing and introduces new concepts such as *select-case*, new coverage requirements are required to characterize the behavior of Go concurrency. Using the GOAT's infrastructure, we studied the underlying causes of bugs in GoKer benchmark [9] and proposed a set of coverage requirements that 1) coherently characterize the dynamic behavior of concurrency primitives under various scheduling scenarios and 2) enable measurement of schedule-space exploration until reaching a threshold, or exposing the bug. By analyzing the test's ECT, we can identify if coverage requirements are met during testing. We demonstrate that our novel coverage metric is effective in measuring the schedule-space exploration progress.

To summarize, here are our main contributions:

- We introduce GOAT, a testing and analysis framework that facilitates whole-program trace collection (via an enhancement to the standard tracer package) and knowledge discovery about the program's dynamic behavior.
- We show the effectiveness of controlled preemptions for concurrency bug exposure in the context of a real-world language
- We propose a set of coverage requirements that characterize the dynamic behavior of concurrency primitives, enabling measurement of quality and progress of schedule-space exploration.

The rest of this paper is as follows: Section II discusses the fundamentals about concurrency debugging in Go and ideas behind GOAT. Section III illustreate the design and implementation of GOAT's components. The evaluation of GOAT on GoKer bug benchmark is illustrated in section IV. Section V discusses the related work and finally, section VI summarizes and concludes.

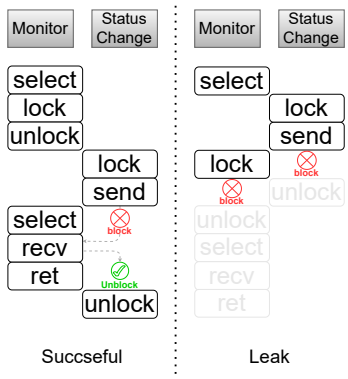
II. BACKGROUND

A. Go Concurrency

Go introduces a new concurrency model, mixing shared-memory features of languages like Java/C/C++ and message-passing concepts such as Erlang's, with an ad-hoc scheduler that orchestrates Go's concurrent components interactions while shielding the user from many low-level aspects of the runtime. The language is equipped with a rich vocabulary of *serialization* features to facilitate the memory model constraints [35]; they include synchronous and asynchronous communication, memory protection, and barriers for efficient synchronization:

- **Goroutines** are functions that execute concurrently on logical processors having their own stacks.
- **Channels** are typed conduits through which goroutines communicate. Channels are unbuffered by default, providing synchronous (rendezvous) or asynchronous (via buffered channels) messaging between goroutines.
- **Synchronization** features such as *(RW)mutex*, *waitGroup*, *conditional variables*, and *select* are included in the language to provide more and flexible synchronization, data access serialization, and memory protection.
- **Scheduler** maintains goroutines in FIFO queues and binds them on OS threads to execute on processing cores.

```
1  package main
2  import "sync"
3
4  type Container struct{
5    sync.Mutex
6    stop  chan struct{}
7  }
8
9  func main() {
10   container := &Container{
11         stop:make(chan struct{})}
12   go Monitor(container)
13   go StatusChange(container)
14 }
15 func Monitor(cnt *Container){
16   for{
17     select{
18     case <- cnt.stop:
19       return
20     default:
21       cnt.Lock()
22       cnt.Unlock()
23 }}}
24 func StatusChange(cnt *Container){
25   cnt.Lock()
26   defer cnt.Unlock()
27   cnt.stop <- struct{}{}
28 }
```



Listing 1: Simplified version of bug `moby28462`

This design facilitates the construction of data flow models that efficiently utilize multiple CPU cores and encourages developers to *share memory through communication* for safe and straightforward concurrency and parallelism. This rich mixture of features has, unfortunately, greatly exacerbated the complexity of debugging. In fact, the popularity of Go has outpaced its debugging support [8], [9], [36]. There are some encouraging developments in support of debugging, such as a data race checker [14] that has now become a standard feature of Go and has helped catch many a bug. However, the support for blocking bugs such as deadlocks and Go-specific bug-hunting support for Go idioms (e.g., misuse of channels and locks) remain insufficiently addressed.

Listing 1 shows a simplified version of a reported bug in Docker [37]. An instance of the `Container` type (lines 4-7) is created in the `main` function (lines 10-11). In line 12, a goroutine is spawned to execute function `Monitor` that continuously checks the container status and returns once it receives from the container's channel (lines 18-19). The default case of the `select` statement (line 20) allows `Monitor` to continue monitoring without getting blocked on the channel receive (line 18). Concurrent to the `main` and `Monitor` goroutines, another goroutine is created in line 13 to execute function `StatusChange` which changes the status of the container by sending to the container's channel. The container's lock is released after the send action completes and function returns (`defer` statement in line 26).

Native execution of this program terminates successfully without issuing any error or warning. Based on the Go specification and memory model, there is no constraint on the goroutines spawned from the `main` function to join back before the `main` goroutine[1] terminates. A deadlock detector within the runtime periodically checks that the scheduler queues of all *runnable* goroutines never become empty until the `main` goroutine terminates. In other words, the runtime throws a deadlock exception when the `main` goroutine is blocked, and no other goroutine is in the queue to execute (i.e., *global deadlock*). Since there is no blocking instruction in the `main` goroutine in listing 1, the program terminates successfully regardless of other goroutines' statuses. However, this program suffers from a common bug in concurrent Go where one or more goroutines *leak*(i.e., *partial deadlock*) from the execution (i.e., never reach their end states).

The right side of the listing displays a successful run and a leak situation of the program. In the leak situation, first, the `Monitor` goroutine executes the `select` statement and, based on the available cases, picks the default case to execute. Right before the execution of mutex lock (line 21), the scheduler context-switches and the `StatusChange` goroutine starts its execution through which it holds the lock and blocks on sending to the channel (line 27) since there is no receiver on that channel. Upon blocking on send, the scheduler transfers back the control to the `Monitor` goroutine that tends to acquire the mutex, but because the mutex is already held by `StatusChange`, the `Monitor` goroutine also blocks. The circular wait between the container mutex and channel prevents both spawned goroutines from reaching their end states and leaves the program in an unnoticed deadlock situation.

B. Concurrency Bugs in Go

Based on a proposed bug taxonomy for Go [8], bugs are categorized separately based on their *causes* (shared-memory vs. message-passing) and *symptoms* (blocking vs. non-blocking). Blocking bugs historically refer to situations where one or more processing units (e.g., goroutines) are blocked, waiting for an external signal to resume (e.g., leak situation in listing 1). The observed causes of such blocking flaws in the context of Go are as follows:

- *Resource deadlocks*: Go inherits resource deadlocks from multithreaded languages like Java and C/Pthreads where goroutines are trapped in a circular wait for the resource (e.g., mutex) that is held by other goroutines.
- *Communication deadlocks*: Synchronized (unbuffered) channels transmit values from one goroutine to another in a rendezvous fashion. The sender (receiver) blocks until the receiver (sender) is ready to receive (send). Misuse of channel operations might result in one or more goroutines waiting for a sender/receiver to unblock them forever.

[1] In the remainder of the paper, we use *main function* and *main goroutine* interchangeably.

- *Mixed deadlocks*: The leak situation in listing 1 is the example of such deadlocks where one goroutine is blocked on acquiring a resource that is held by another goroutine which is blocked on communication.

Similar to other concurrent languages, Go has non-blocking bugs such as data races and atomicity violations while introducing new bug idioms due to its new concepts such as anonymous functions [8]. This work focuses on blocking bugs[2].

In addition to the non-deterministic nature of concurrent languages caused by the scheduler and interaction between concurrent components, Go introduces some level of non-determinism at the application level. The *select-case* statement (similar to switch-case) allows the goroutine to wait on multiple channel operations. The runtime picks one case pseudo-randomly among available cases (i.e., channel sends and receives that are ready to execute without blocking). If none of the cases are ready, the executing goroutine is blocked unless there is a *default* case. The default case makes the select non-blocking and prevents the goroutine from waiting for unavailable communications. Such random behavior expands the interleaving space, and it grows exponentially when nested selects are employed in conjunction with nested loops. As a result, tracing the cause of a program's misbehaved execution becomes increasingly tricky. Our observations (section IV) demonstrate that **select statements are involved at the center of many rare bugs**.

C. Accelerating Bug Exposure

Blocking bugs are primarily caused by the non-deterministic decisions that the scheduler makes. Such bugs may not manifest themselves in conventional testing and are difficult to reproduce. Figure 2 displays the histogram of 68 blocking bug kernels grouped by the number of trials that GOAT takes to detect them. Approximately 30% of bugs required more than one execution to happen and be detected by GOAT[3]. *Stress testing* is a common way to detect such rare bugs by exercising the scheduler and examine the program's behavior in many executions. However, such testing is inefficient because some interleavings might get tested repeatedly while other feasible interleavings remain untested. It has been empirically demonstrated that a small amount of randomness in each test execution can drastically reduce the number of iterations needed to find concurrency bugs [22], [23]. For instance, forcing context-switches before synchronization/serialization operations in concurrent programs increases the probability of finding rare concurrent bugs [24]. In listing 1, a rare context-switch after the `select` statement in line 17 causes the lock operation on mutex m in line 21 of goroutine `Monitor` to *block* goroutine `StatusChange` on locking m in line 25 and causing a deadlock. Concurrency primitive usages (e.g., channel send/recv, mutex lock/unlock, select) are the

[2]Throughout this work, leaks (partial deadlocks) and global deadlocks are interchangeably referred as *deadlocks* as a general term for blocking bugs.

[3]The figure and numbers are obtained from trials of GOAT on native execution of bug kernels without any randomization (i.e., $D = 0$).

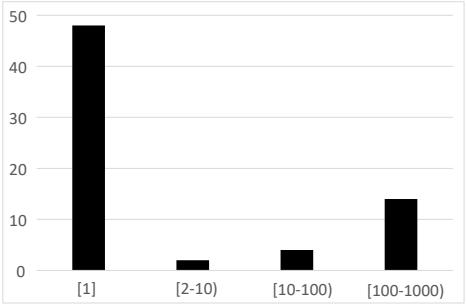


Figure 2: Distribution of number of trials falling into given intervals to detect 68 blocking bugs in GoKer [9] using GOAT ($D0$)

critical points in the program because their behavior directly impacts the blocking behavior of Go programs. In GOAT, we statically identify such critical points and inject *yield* handlers before each concurrency primitive usage. During execution, the handlers randomly decide if the current goroutine should yield to other goroutines to execute. Results in section IV show that such simple yields are effective in detecting rare bugs.

D. Testing Coverage Analysis

To demonstrate that testing has been thorough, *coverage metrics* are defined to measure the progress of tests and specify testing termination conditions. Coverage metric for the set of testing executions $\mathcal{T}$ is a set of *requirements* $\mathcal{R}$ that should get covered during testing iterations. We say requirement $R_i \in \mathcal{R}$ is covered during testing iteration $t_j \in \mathcal{T}$ if we can correlate an *action* during execution of t_j to R_i. For example, in *statement coverage*, which is a widely-used metric in testing sequential software, R is the set of source locations (file and line numbers) in the target program. R_i is covered by test execution t if the statement at location R_i is executed in t. The *coverage percentage* of a test $\mathcal{T}$ is the ratio of the requirements covered by at least one execution over the number of all requirements ($|R|$).

In the context of concurrent software testing, coverage metrics are proposed to quantify the quality of search space exploration. *Synchronization* coverage metrics such as *blocking-blocked* [29], *blocked-pair-follows* [33] and *synchronization-pair* [30] defined requirements to cover during testing for exposing blocking bugs (e.g., deadlocks). For example, the synchronization coverage model in [29] defines *blocking* and *blocked* requirements per each synchronized block (i.e., mutually exclusive section of the code that is protected by a lock). The purpose of this requirement is to check if a test can report when there is a lock contention for two or more threads entering the synchronized block. That is, a thread is either *blocked* from entering the synchronization block or *blocking* other threads from entering by holding the lock.

The existing concurrency coverage metrics are primarily in the context of Java and C/Pthreads. They are not necessarily applicable to languages like Go, as such languages have different concurrency primitives and semantics. Novel coverage

metrics are required to enable the quantification of interleaving space exploration. Bron et al.,[38] enumerates four major characteristics for coverage metrics to gain acceptance:

1) **Static model:** A static model of requirements from the given program should be constructed by instrumenting the source code. The model should be well-understood by the developer or tester before execution. The model should maintain covered requirements during testing executions.
2) **Coverable and measurable requirements:** The absolute majority of requirements should be realistic enough to be *coverable* during testing. For a few that are not coverable (due to program semantics) or not *measurable* (because of technical limitations), the developer should be aware of the reason.
3) **Actions for uncovered requirements:** After testing terminates, every uncovered requirement should yield an action (e.g., extending testing iterations or removing dead code from the program, thus removing uncoverable requirements)
4) **Coverage satisfaction:** Some action should be taken upon reaching a threshold of coverage percentage (e.g., testing phase termination when reaching 100% statement coverage)

Defining a new coverage metric to satisfy the above characteristics requires an accurate and proper mental model of target bugs. Using the GOAT's infrastructure, we studied the underlying causes of many bugs, including GoKer benchmark [9] and propose a set of coverage requirements that enables extensive analysis of dynamic behavior of concurrency primitives under various scheduling scenarios. In section III-C, we describe our proposed coverage metric for testing concurrent Go, which is extensible to all concurrent languages.

III. Design and Implementation

A. Overview

Figure 1 displays the overview of GOAT. Given a program **P** (i.e., a set of Go source files with a *main* function), GOAT automatically instruments **P** and constructs static and dynamic models to facilitate the investigation of *non-deterministic interactions between concurrent components* (i.e., concurrent behavior) of **P**, and achieve objectives introduced in section I.

Static Analysis: (section III-B)— GOAT statically constructs a model **M** which is a table of source locations (files and line numbers) associated with concurrency primitive usages in **P** source files. The primary use of **M** is to identify locations in **P** as potential points for manipulating the schedule to explore possible scenarios and accelerate the discovery of rare bugs. Yield handlers are injected before each entry in **M** to decide if the following concurrency action (e.g., message send or mutex lock) should perform or yield to other goroutines. Such yields effectively perturb the scheduler and execute feasible but rarely taken interleavings of **P**.

Coverage Requirements: (section III-C)— Forcing the schedule perturbation is effective for exploring the feasible interleaving space until the bug is hit. However, a metric is required to evaluate the quality of interleaving space exploration and measure the progress until reaching a threshold. Following the tenets of effective coverage metrics, we employ **M** to emulate the possible behavior of concurrent components of **P** and define a set of *coverage requirements* as indicators for quality and progress of schedule space exploration. The requirements are defined so that, during testing, uncovered requirements demands the user to fix the bug or remove the dead code.

Dynamic Analysis: (section III-D)— To gain insight into the concurrent behavior of **P** and measure the covered requirements, we equipped GOAT with a dynamic tracing mechanism, which is an extension to the Go standard tracer package [39]. When tracing is enabled, an *execution concurrency trace* (ECT) file is generated once the execution of **P** terminates (e.g., successfully exits, fails, times out). ECT is a totally ordered *sequence* of events that contain information about the dynamic behavior concurrent components, enabling offline analysis of **P**'s execution.

Offline Analysis: (section III-E)— In offline, GOAT first separates the application-level events of ECT from the underlying runtime system of Go. Then, it constructs a goroutine tree from application-level goroutines to check if any goroutine has leaked/blocked (i.e., did not reach its final state) after the execution termination. Additionally, GOAT maintains a global goroutine tree for **P** and maps goroutines from run to run to accumulate the covered requirements from each execution of **P**. As soon as a bug is detected or the coverage exceeds a threshold, GOAT stops running and produces reports for manual analysis by the user.

B. Static Analysis

1) Concurrency Usage Model: GOAT statically constructs a model **M** from the usage of concurrency primitives in **P** files which enables uniform analysis during testing iterations. **M** is a table of source locations (files and line numbers) associated with *concurrency usages* (CU). We define CU as a triple of (f, l, k) where f is the file name, l is the line number, and k is the concurrency primitive used in the code location. $k \in$ `Channel` $\cup$ `Sync` $\cup$ `Go` where:

- `Channel` = {`send`, `receive`, `close`}
- `Sync` = {`lock`, `unlock`, `wait`, `add`, `done`, `signal`, `broadcast`}
- `Go` = {`go`, `select`, `range`}

GOAT constructs **M** by traversing the *abstract syntax tree* (AST) for each file in **P** using the Go AST package [40]. The first column of table III shows the CU locations extracted from program in listing 1.

2) Source Instrumentation: We employ **M** entries to instrument **P** with tracing and schedule perturbation mechanisms. First, we traverse the AST of **P** and inject three statements (i.e., AST nodes) to the beginning of **P**'s main function to enable end-to-end tracing:

- `goat_done := goat.Start()` initializes GOAT, enables tracing, and returns a channel as a conduit between application space and GOAT.

- `go goat.Watch(goat_done)` spawns a new goroutine as a watchdog for checking the liveness of the program (in case of global deadlock or infinite loop). The watchdog goroutine either receives from `goat_done` and sends back an ack signal or timeouts (default: 30 seconds). Then it stops tracing, flushing the trace buffer, and terminates.
- `defer goat.Stop(goat_done)` sends a value to the watcher goroutine after main returns and signals that the program is finished. Then GOAT waits to receive the signal from the watcher, then stops tracing and terminates.

Moreover, we inject `goat.handler()` statements before each CU in **M** to manipulate the native scheduler around concurrency primitve usages. `goat.handler()` is a function invocation that randomly calls `runtime.GoSched()` within a bound D to preempt the processing core from current goroutine and push the goroutine to the back of the global queue of runnable goroutines. When $D = 0$, GOAT does not perturb the scheduler and lets **P** to execute natively. For any $D > 0$, GOAT manipulates application-level goroutines from their regular execution D times. Our experiments (section IV) demonstrate that the optimum value for D is not larger than 3, showing that even a small number of yields is effective in exposing the bug (as also shown in [24]).

C. Coverage Requirements

Based on our investigations from the execution of Go applications and bug kernels, we emulate the possible behavior of concurrent components by defining a set of coverage requirements (summarized in table I):

- **Req1 (Send/Recv):** {`blocked`, `unblocking`, `NOP`} – Goroutine G_1 either is *blocked* on a channel send (receive) if the receiver (sender) goroutine G_2 is not ready, or is *unblocking* the waiting receiver (sender) goroutine G_2. A channel send or receive might also be neither blocked nor unblocking (NOP) for buffered channels.
- **Req2 (Select-Case):** {`blocked`, `unblocking`, `NOP`} $\times$ {`case`$_i$} – cases of select statements are channel sends and recives (or default case for non-blocking selects). For all select statements that has no default case, we obtain the cases of each select statement at runtime and maintain an instance of requirement Req1 per case.
- **Req3 (Lock):** {`blocked`, `blocking`} – Goroutine G_i either is *blocked* when locking a mutex because another goroutine has locked the mutex or is *blocking* other goroutines from acquiring the mutex lock.
- **Req4 (Unblocking):** {`unblocking`, `NOP`} – The goroutine that is performing one of the unblocking actions such as channel close, mutex unlock, conditional variable signal and broadcast, waitGroup done, and non-blocking select case (send or receive) either *unblocks* one or more blocked goroutines or has no effect (NOP).
- **Req5-Go:** {`NOP`} – We emit a NOP action for each goroutine creation to indicate its coverage during testing.

Table I: Coverge requirements defined for concurrent Go

Coverage Requirements	Concurrent Action	Coverage Requirement Types			
		Blocked	Unblocking	Blocking	NOP
Req. 1: Send/Recv	SEND	*	*		*
	RECV	*	*		*
Req. 2: Select-Case	CASE$_i$ (SEND)	*	*		*
	CASE$_i$ (RECV)	*	*		*
Req. 3: Lock	LOCK	*		*	
Req. 4: Unblocking	CLOSE		*		*
	UNLOCK		*		*
	SIGNAL		*		*
	BRDCST		*		*
	NB-SELECT		*		*
Req. 5: Go	Go				*

With the help of GOAT's infrastructure, our implemntation of the proposed requirements satisfy the characteristics of an "acceptable" coverage metric because:

1) A *static model* **M** from program **P** is obtained by identifying its CU points. **M** is easy to understand by developers and reflects the concurrent behavior of **P**.
2) The defined requirements are *measurable* by analyzing the test's ECT. A global data structure maintains the covered requirements by each $t \in \mathcal{T}$.
3) Upon completion of $\mathcal{T}$ iterations, the *uncovered* requirements imply some *meaningful* information about the behavior of **P**. For example, if a send is always performing as *unblocking* and never as *blocked*, it means that the receiver always performs receive before the sender reaches its send instruction. In other words, the receive action *always happen-before* send action. This communication pattern might be part of **P**'s semantics and matches the developer's expectations (e.g., a set of goroutines are listening on a channel to perform non-frequent requests). Otherwise, the uncovered requirement "send-`blocked`" reflects a bug or flaw in the program.
4) Since GOAT can detect occurred blocking bugs and maintain a global coverage model, $\mathcal{T}$ iterations terminate either by detecting a bug or reaching a percentage threshold.

D. Dynamic Concurrency Tracing

The standard execution tracer package [39], [41] provides dynamic tracing for the construction of execution models from the interactions of processors, OS threads, goroutines, the scheduler, and the garbage collection mechanism. The tracing mechanism is compiled into all programs always through the runtime and is enabled on demand to study *perfromance bottlenecks* through visualizers like *pprof* [42]. The alphabet of trace events is total of 49 events [43], categorized and summarized in table II. Although the event vocabulary is rich enough to model comprehensive goroutine latency and blocking behavior accurately, the vocabulary lacks concurrency primitive usage events for the construction of concurrency models. We enrich the standard tracing mechanism with 14 additional events to enable the production of dynamic models from the program's concurrency behavior:

- **Channel:** For each channel operation (make, send, receive, close), ECT includes an event with a unique id assigned to each channel.

Table II: Event categories by the Go execution tracer

Category	Description
Process	Indication of process/thread start and stop
GC/Mem	Garbage collection and memory operation events
Goroutine	Goroutines events: create, block, start, stop, end, etc.
Syscall	Interactions with system calls
Users	User annotated regions and tasks (for bounded tracing)
Misc	System related events like futile wakeup or timers

Table III: Concurrency Usages and coverage requirements of program in listing1

CU of list. 1 Line	CU of list. 1 Kind	Coverage Requirements	Covered by run #1	Covered by run #2	Overall Covered
12	go	covered	✓$_{G0}$	✓$_{G0}$	✓
13	go	covered	✓$_{G0}$	✓$_{G0}$	✓
17	select	c-recv-blocked	✓$_{G1}$		✓
		c-recv-unblocking	✓$_{G1}$		✓
21	lock	blocked		✓$_{G1}$	✓
		blocking	✓$_{G1}$		✓
22	unlock	unblocking	✓$_{G1}$		✓
		no_op			
25	lock	blocked	✓$_{G2}$		✓
		blocking		✓$_{G2}$	✓
26	send	blocked	✓$_{G2}$	✓$_{G2}$	✓
		unblocking			
		no_op			
27	unlock	unblocking			
		no_op	✓$_{G2}$		✓
		Coverage %	60%	33%	73%

- **(RW)Mutex, WaitGroup & Conditional Variables:** Similar to channels, we assign a unique id to each concurrency object and emit an event for each of their operations (lock, unlock, rlock, runlock, add, wait, signal, broadcast).
- **Select & Schedule:** The scheduler and the *select* structure introduce non-determinism to the execution. We keep track of the decisions made by the scheduler and select statements to obtain an accurate decision path.

We call the output of enhanced tracer *execution concurrency trace* (ECT). ECT is a totally ordered *sequence* of events in which the order is approximated through a central clock with nanosecond precision. ECT also contains the call-stack for each event, enabling a direct mapping of events to source-line numbers. For each blocking operation (channels *sends*/*recvs*, mutex *locks*, waitGroup/conditional variable *wait* and *select* (when none of the cases are available)), ECT captures a pair of pre-operation and post-operation events to distinguish between the *request for action* and *completion of action*. Hence, ECT is especially effective for debugging because it enables modeling the blocking state of concurrent components at any given step of program execution. The enhanced dynamic tracing also enables the measurement of coverage requirements in offline (section III-E).

E. Offline Analysis

In the lifetime of Go programs' executions, the runtime system creates new goroutines or pick from the pool of dead goroutines to perform various tasks such as bootstrapping the program, garbage marking and sweeping, and tracing. GOAT also adds an extra goroutine to *watch* the program execution in case of the main goroutine blockage. These goroutines are captured during tracing, but our focus is on

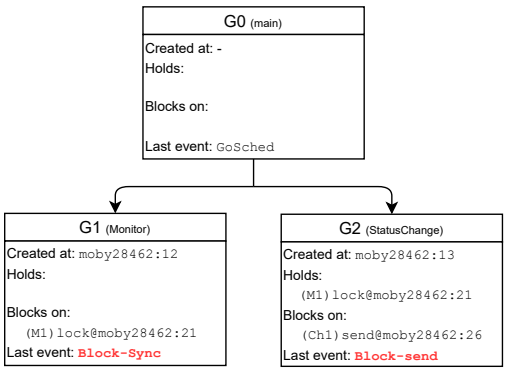


Figure 3: Goroutine tree of the leak situation in listing 1

the goroutines created from within the application. The distinguishment between runtime goroutines and application goroutines is essential to define the boundaries of the application and separate them from the underlying system. We say a goroutine is an *application-level* goroutine if it is the main goroutine (that executes the main function) or it has all of the following conditions: 1) its ancestor is the main goroutine, 2) it is not a Go runtime system goroutine, and 3) it is not a tracer goroutine. These conditions are assessed for every captured goroutine in ECT by checking the call stack of their corresponding $GoCreate$ event.

GOAT constructs a *goroutine tree* (figure 3) of application-level goroutines from the generated ECT. Nodes in the goroutine tree represent a goroutine, and directed edges denote parent-child relationships in which the child is created from a `go` statement that the parent executes. Each node of the tree contains the entire sequence of events that the goroutine executed, information about the goroutine's creation site, the resources it holds at each execution point, and the final executed event right before the program termination. GOAT analyzes this collection of information to check whether any goroutine leaked after termination and whether the coverage requirements are covered.

1) Deadlock Detection: When tracing is enabled, every application goroutine invokes the tracer to capture $GoEnd$ before finishing its execution. Before the main function returns, the main goroutine hands over the control to the root goroutine to finalize the program termination. This context-switch is done through invocation of `runtime.Gosched()` which emits the $GoSched$ event. In GOAT, the main goroutine's final event in a successful execution is $GoSched$ with `runtime.traceStop` on top of its stack.

We call an execution **successful**, if below conditions hold:

1) all goroutines spawned in the main goroutine has $GoEnd$ as their final event,
2) the final event of the main goroutine is $GoSched$ with `runtime.traceStop` on top of its stack.

In the absence of any of these conditions, we conclude that the program suffers from a "deadlock" bug because at least one goroutine did not reach its final state. Therefore, GOAT executes procedure 1 which is a BFS traversal on the goroutine

tree to check if the program suffers from partial or global deadlocks.

```
DeadlockCheck(G):
    if cur.lastEvent≠ GoSched then
        return Global Deadlock
    end
    toVisit = [G.children]
    for |toVisit| ≠ 0 do
        cur = toVisit[0]
        if cur.lastEvent ≠ GoEnd then
            return Partial Deadlock (leak)
        end
        for n in cur.Children do
            append n to toVisit
        end
        toVisit = toVisit[1 :]
    end
    return Pass
```

Procedure 1: `DeadlockCheck` procedure with root node of goroutine tree (main goroutine) as input

When a deadlock is detected, GOAT generates visualizations such as executed interleaving (listing 1) and goroutine tree (figure 3). The detailed report magnifies the scenario under which the bug has occurred and displays the final concurrent state of the program right before the failure. Samples of such reports and visualizations are available in [44].

2) Coverage Measurement: Once the execution terminates, GOAT checks whether the extracted coverage requirements are covered. A mapping between ECT dynamic concurrent events and statically obtained CU points is emitted by matching their respective call-stack and CU source location. Through a BFS traversal of the goroutine tree, we add a *coverage vector* to each goroutine node from the emitted mapping. Each element of the coverage vector is the respective covered value of the coverage requirement for the current goroutine node. During executions of tests $t \in \mathcal{T}$, we maintain and update a global goroutine tree after each t to measure the progress of coverage percentage over tests in $\mathcal{T}$. However, equivalencing between two goroutines and their respective coverage vectors from different executions is non-trivial. We say two goroutines G_m and G_n in the tests t_i and t_j are *equivalent* (i.e., falls into a identical node in the global goroutine tree) if their parents are equivalent and their creation source location (CU of kind `go`) are identical.

$$G_m \equiv G_n \text{ if } \begin{cases} \text{parent}(G_m) \equiv \text{parent}(G_n) \wedge \\ \text{CU}(G_m).\text{file} = \text{CU}(G_n).\text{file} \wedge \\ \text{CU}(G_m).\text{line} = \text{CU}(G_n).\text{line} \end{cases} \quad (1)$$

IV. Evaluation

A. Deadlock Detection

We assess the ability of GOAT and its variations in detecting bugs with the minimum number of executions required to expose the bug. We have compared GOAT against three existing dynamic detectors:

- *Built-in* deadlock detector: It is an embedded mechanism in the standard Go runtime. The mechanism periodically makes sure that the queue of *runnable* goroutines is never empty until the main goroutine terminates. If the queue is empty and the main goroutine has not terminated yet (i.e., main is blocked), it throws a runtime error.
- *LockDL* [45]: This tool intercepts with all mutex locks and unlocks of the target application to maintain a "lock-set" data structure. *LockDL* issues warning during runtime when it finds a circular wait in the lock-set or double-locking the same lock. It has a timeout mechanism for the application that traps into global deadlocks (30 seconds).
- *goleak* [46]: This leak detector from Uber checks the program stack at the end of the main goroutine's execution to find the application-level goroutines that remained in the stack (i.e., leaked).

All experiments are performed on a server with Intel(R) Xeon(R) CPU E7 processor (64 total cores with two threads per core and eight cores per socket), 512 GB of RAM with Ubuntu 5.4.0 and Go version 1.15.6. Table IV shows the details of results obtained from 1000 executions of each tool per bug.

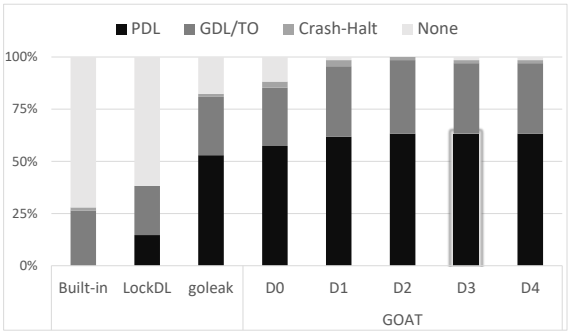


Figure 4: Histogram of detected bugs by each tool performed on 68 GoKer blocking bugs. PDL: partial deadlock, GDL/TO: global deadlock, Crash/Halt: causes the program to crash or halt during detection.

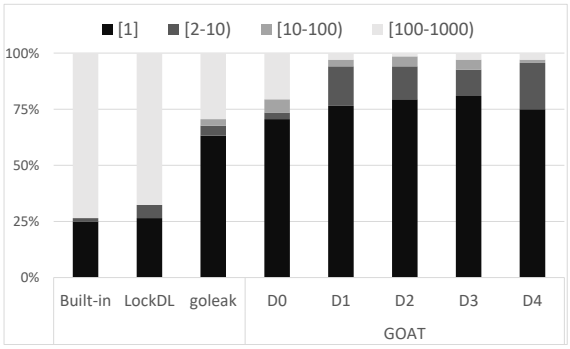
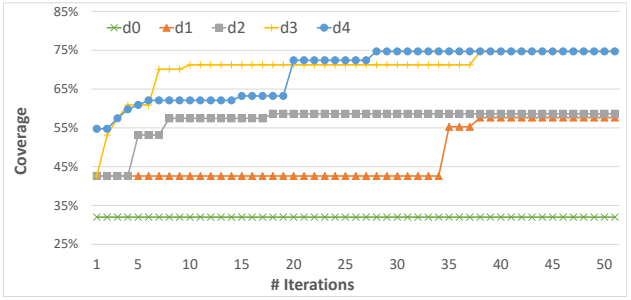


Figure 5: Percentage distribution for the average required number of iterations (falling into each of the four given intervals) by each tool to detect 68 GoKer blocking bugs.

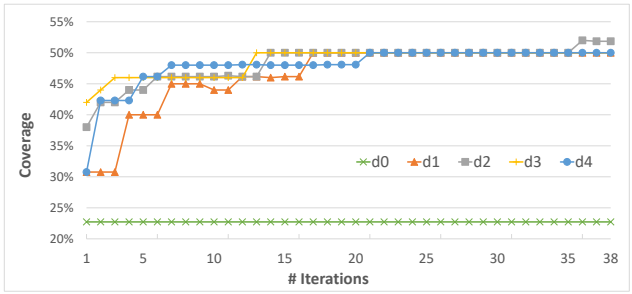
Table IV: Output of each tool on GoKer [9] blocking bugs. Detected bug (minimum # of executions required) – **X (1000)**: the tool is not able to detect any bug after 1000 executions. **PDL**: Partial Deadlock, **GDL**: Global Deadlock, **PDL-k**: Partial Deadlock with *k* number of goroutines leaked. **DL**: A warning for potential deadlock is issued. **TO/GDL**: The global deadlock is detected because none of goroutines made any progress after 30 seconds, **CRASH**: The execution paniced because of a flaw in the execution (e.g., send on closed channel panic), **HANG**: The tool halt for more than 10 minutes.

Bug Description			Debugging Tools							
Cause	SubCause	Bug ID	BUILTINDL	GOLEAK	LOCKDL	GOAT				
						D0	D1	D2	D3	D4
Communication Deadlock	Channel	cockroach_2448	X (1000)	X (1000)	X (1000)	CRASH (1)	CRASH (1)	CRASH (1)	CRASH (1)	CRASH (1)
		cockroach_24808	GDL (1)	GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		cockroach_25456	GDL (1)	GDL(1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		cockroach_35073	GDL (1)	GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		cockroach_35931	GDL (1)	GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		etcd_6857	X (1000)	PDL (325)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (11)	PDL-1 (3)	PDL-1 (3)
		grpc_1275	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		grpc_1424	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		grpc_660	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		istio_17860	X (1000)	PDL (1)	X (1000)	PDL-1 (2)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		kubernetes_38669	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		kubernetes_5316	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-2 (1)	PDL-1 (1)	PDL-2 (1)	PDL-2 (1)
		kubernetes_70277	GDL (1)	GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		moby_21233	X (1000)	PDL (1)	X (1000)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)
		moby_33293	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (3)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		moby_4395	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		syncthing_5795	GDL (1)	GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
	Channel & Conditional Variable	kubernetes_11298	X (1000)	X (1000)	TO/GDL (179)	X (1000)	TO/GDL (352)	TO/GDL (158)	TO/GDL (179)	TO/GDL (179)
		moby_27782	X (1000)	PDL (741)	X (1000)	X (1000)	PDL-2 (1)	PDL-2 (1)	PDL-2 (4)	PDL-2 (4)
	Channel & Context	cockroach_10790	X (1000)	PDL (3)	X (1000)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)
		cockroach_13197	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		cockroach_13755	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		cockroach_18101	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		grpc_862	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		istio_18454	X (1000)	PDL (13)	X (1000)	PDL-2 (5)	PDL-1 (14)	PDL-1 (4)	PDL-1 (6)	PDL-1 (6)
		kubernetes_25331	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		moby_33781	X (1000)	PDL (1)	X (1000)	PDL-1 (221)	PDL-1 (10)	PDL-1 (8)	PDL-1 (10)	PDL-1 (10)
	Condition Variable	moby_29733	GDL (1)	GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		moby_30408	GDL (1)	GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
Mixed Deadlock	Channel & Lock	etcd_6873	X (1000)	PDL (371)	X (1000)	PDL-2 (1)	PDL-2 (2)	PDL-2 (7)	PDL-2 (6)	PDL-2 (6)
		etcd_7443	X (1000)	X (1000)	X (1000)	X (1000)	PDL-1 (9)	PDL-1 (15)	PDL-1 (14)	PDL-1 (14)
		etcd_7492	HANG (1)	HANG (1)	TO/GDL (4)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		etcd_7902	X (1000)	PDL (1)	X (1000)	PDL-4 (1)	PDL-4 (1)	PDL-4 (1)	PDL-4 (1)	PDL-4 (1)
		grpc_1353	X (1000)	PDL (1)	X (1000)	CRASH (1)	CRASH (1)	PDL-3 (1)	PDL-3 (1)	PDL-3 (1)
		grpc_1460	X (1000)	PDL (1)	X (1000)	PDL-2 (135)	PDL-2 (1)	PDL-2 (2)	PDL-2 (1)	PDL-2 (1)
		istio_16224	GDL (1)	GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		kubernetes_10182	X (1000)	PDL (44)	X (1000)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)
		kubernetes_1321	X (1000)	PDL (307)	X (1000)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		kubernetes_26980	GDL (375)	GDL (131)	X (1000)	TO/GDL (191)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		kubernetes_6632	X (1000)	X (1000)	X (1000)	PDL-2 (1)	PDL-2 (1)	PDL-2 (2)	PDL-2 (1)	PDL-2 (1)
		moby_28462	X (1000)	PDL (5)	X (1000)	PDL-2 (39)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)
		serving_2137	X (1000)	X (1000)	X (1000)	X (1000)	X (1000)	TO/GDL (88)	X (1000)	X (1000)
	Channel & WaitGroup	cockroach_1055	GDL (1)	GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		cockroach_1462	X (1000)	X (1000)	TO/GDL (1)	X (1000)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
	Misuse WaitGroup	moby_25384	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
Resource Deadlock	Double locking	cockroach_584	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		cockroach_9935	X (1000)	PDL (1)	DL (721)	PDL-1 (1)	PDL-1 (2)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		etcd_10492	GDL (1)	GDL (1)	DL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		etcd_5509	X (1000)	GDL (766)	TO/GDL (426)	X (1000)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		etcd_6708	GDL (1)	GDL (1)	DL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		grpc_3017	GDL (4)	GDL (4)	TO/GDL (3)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		grpc_795	GDL (1)	GDL (1)	DL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		hugo_5379	GDL (1)	GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		moby_17176	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		moby_36114	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		moby_7559	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		syncthing_4829	GDL (1)	GDL (1)	DL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
	RWR deadlock	cockroach_16167	X (1000)	X (1000)	DL (1)	X (1000)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		cockroach_3710	X (1000)	X (1000)	DL (123)	PDL-2 (28)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)
		cockroach_6181	X (1000)	PDL (1)	X (1000)	PDL-4 (1)	PDL-4 (1)	PDL-3 (1)	PDL-1 (1)	PDL-1 (1)
		hugo_3251	GDL (1)	GDL (1)	DL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		kubernetes_58107	X (1000)	X (1000)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		kubernetes_62464	X (1000)	X (1000)	DL (6)	PDL-2 (161)	PDL-2 (7)	PDL-2 (2)	PDL-2 (3)	PDL-2 (3)
	AB-BA deadlock	cockroach_10214	X (1000)	X (1000)	X (1000)	PDL-2 (368)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)
		cockroach_7504	X (1000)	X (1000)	X (1000)	PDL-2 (199)	PDL-2 (1)	PDL-2 (7)	PDL-2 (1)	PDL-2 (1)
		kubernetes_13135	X (1000)	PDL (1)	DL (4)	PDL-2 (1)	PDL-2 (5)	PDL-2 (1)	PDL-2 (22)	PDL-2 (22)
		kubernetes_30872	X (1000)	PDL (338)	X (1000)	PDL-3 (50)	PDL-3 (2)	PDL-3 (1)	PDL-3 (6)	PDL-3 (6)
		moby_4951	X (1000)	PDL (120)	X (1000)	PDL-2 (15)	PDL-2 (2)	PDL-2 (2)	PDL-2 (1)	PDL-2 (1)
Total Bugs:	68	Total Detected:	19	56	26	60	67	68	67	67

Figure 4 and table IV show that variations of GOAT outperforms other detector by discovering the bug in 100% of the GoKer blocking benchmark. For example, the bug `kubernetes_6632` that is caused by misuse of channels and locks, only detected by GOAT after a couple of executions, while other tools were unable to detect it after 1000 executions. Figure 5 and highlighted cells of table IV show that the idea of injecting random delays around concurrency usage points in the program drastically reduces the required number of testing iterations until the bug occurs. $D0$ means GOAT did not delay the program at any point and $D4$ means that the target program has been delayed up to four times around its CU points. Figures 4 and 5 also state that the increase in the delay bound of GOAT does not necessarily increase the chance of exposing the bug. For example, the row of bug `serving_2137` in table IV show that only GOAT $D2$ were able to detect the bug.



(a) `etcd7443`



(b) `kuberenetes11298`

Figure 6: Coverage percentage after each iteration of GOAT with various D values for two representative bugs. Iterations on the X axis of figures end when the respective bug is first detected. E.g., GOAT ($D2$) detects the bug in `kuberenetes11298` after 38 executions at 52.23% coverage percentage.

B. Coverage Analyis

We picked two representative bug kernels `etcd7443` and `kubernetes11298` to evaluate the coverage idea as they both have extensive use of channels, mutexes, conditional variables, nested selects within nested for loops, and the buggy interleaving is proved to be rare to happen. Figures 6a and 6b show the gradual increase in coverage percentage during testing iterations for different values of D. Recall that D is the bound on the number of yields that we inject to the native execution of a given program to perturb the scheduler around concurrency usages. With the increase of D, the coverage percentage increase rate also grows. With lower values of D, the coverage percentages start at lower values and increase more slowly over iterations. The reason is that the scheduler does not get to explore different interleavings (thus different coverage scenarios), and over iterations, the program executes more deterministic regarding coverage requirements. However, higher D does not necessarily increase the coverage ($D2$ and $D4$ in figure 6b). The gradual increase of the coverage percentage and non-uniform increase rates for different values of D reflects the effectiveness of our proposed coverage metric. The drop in coverage for $D1$ in figure 6b is because of the new coverage requirements (e.g., a new goroutine is spawned and executing some concurrency primitives) that were encountered during testing execution.

V. Related Work

For correctness of CSP-based concurrent languages like Go, Ng and Yoshida [10] proposed a static tool to detect global deadlock in Go programs using choreography synthesis. Later, Stadtmuller et al. [11] proposed a static trace-based global detection approach based on forkable regular expressions. Lange et al. proposed more static verification frameworks for checking channel safety, and liveness [12], and behavioral model checking [13]. Both methods approximate Go programs with session types and behavioral contracts extracted from their SSA intermediate representation. The mentioned work has limitations for handling dynamic (e.g., in-loop) goroutine or channel creation, and programs with many goroutines. Besides, the rate of generated false positives is high. As dynamic (runtime-level) analysis approaches, Zhao et al. [15] introduced a heuristic-based runtime monitoring approach for deadlock detection in Occam programs. Sulzmann and Stadtmuller proposed a dynamic verification approach for synchronous (unbuffered) channels [16], and a vector-clock-based approach for asynchronous channels [17]. Although they may support a larger subset of the Go language, they only focus on channels as the root cause of deadlocks and evaluated on relatively small examples. Also, they usually do not scale for applications with thousands of goroutines and LOC [47].

Researchers have applied different systematic testing methods [22] to reduce the interleaving space to explore effectively and efficiently. Delay-bounded [23], [24] and preemption-bounded [25] techniques systematically "fuzz" the scheduler to equally and fairly cover feasible interleaving. Other tools like Maple [26], CalFuzzer [27], and ConTest [29] *actively* control the scheduler to maximise a pre-defined concurrency coverage criterion [30] or the probability of bug exposure [24].

VI. Summary & Future Work

We presented GOAT, a testing framework for concurrent Go applications to assist concurrency debugging of real-world applications. GOAT combines static and dynamic methods to model and explores application execution. GOAT detects all 68 blocking bugs of GoKer benchmark, which are the bug kernels of the top nine open-source projects written in Go. The scheduler behavior is perturbed with automatically injected random delays to accelerate the bug exposure. By dynamic measurement of a set of coverage requirements, we quantify the quality of schedule-space exploration of GOAT. Proposed coverage requirements accurately reflect the dynamic behavior of program executions and testing iterations.

The engineering of GOAT is flexible and extensible to more advanced components. For example, the current minimal GOAT engine can be extended to take full control over the Go scheduler and "guide" testing towards untested interleaving. We are dockerizing GOAT for easy and public use, making it practical for production testing.

REFERENCES

[1] "Effective Go." [Online]. Available: https://golang.org/doc/effective_go.html

[2] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, p. 666–677, Aug. 1978. [Online]. Available: https://doi.org/10.1145/359576.359585

[3] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[4] "Kubernetes Reference." [Online]. Available: https://kubernetes.io/docs/reference/

[5] "Etcd: A distributed, reliable key-value store for the most critical data of a distributed system." [Online]. Available: https://github.com/coreos/etcd

[6] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis, "Cockroachdb: The resilient geo-distributed sql database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1493–1509. [Online]. Available: https://doi.org/10.1145/3318464.3386134

[7] "A high performance, open source, general RPC framework that puts mobile and HTTP/2 first." [Online]. Available: https://github.com/grpc/grpc-go

[8] T. Tu, X. Liu, L. Song, and Y. Zhang, "Understanding real-world concurrency bugs in go," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 865–878. [Online]. Available: https://doi.org/10.1145/3297858.3304069

[9] T. Yuan, G. Li, J. Lu, C. Liu, L. Li, and J. Xue, "Gobench: A benchmark suite of real-world go concurrency bugs," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 187–199.

[10] N. Ng and N. Yoshida, "Static deadlock detection for concurrent go by global session graph synthesis," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 174–184. [Online]. Available: https://doi.org/10.1145/2892208.2892232

[11] K. Stadtmüller, M. Sulzmann, and P. Thiemann, "Static trace-based deadlock analysis for synchronous mini-go," in *Programming Languages and Systems*, A. Igarashi, Ed. Cham: Springer International Publishing, 2016, pp. 116–136.

[12] J. Lange, N. Ng, B. Toninho, and N. Yoshida, "Fencing off go: Liveness and safety for channel-based programming," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 748–761. [Online]. Available: https://doi.org/10.1145/3009837.3009847

[13] J. Lange, N. Ng, B. Toninho, and N. Yoshida, "A static verification framework for message passing in go using behavioural types," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1137–1148. [Online]. Available: https://doi.org/10.1145/3180155.3180157

[14] D. Vyukov and A. Gerrand, "Introducing the go race detector," 2013. [Online]. Available: https://blog.golang.org/race-detector

[15] J. Zhao, H. Abe, Y. Nomura, J. Cheng, and K. Ushijima, "Runtime detection of communication deadlocks in occam 2 programs," pp. 97–107, 1997.

[16] M. Sulzmann and K. Stadtmüller, "Trace-based run-time analysis of message-passing go programs," *CoRR*, vol. abs/1709.01588, 2017. [Online]. Available: http://arxiv.org/abs/1709.01588

[17] M. Sulzmann and K. Stadtmüller, "Two-phase dynamic analysis of message-passing go programs based on vector clocks," ser. PPDP '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3236950.3236959

[18] N. Dilley and J. Lange, "Bounded verification of message-passing concurrency in go using promela and spin," *Electronic Proceedings in Theoretical Computer Science*, vol. 314, p. 34–45, Apr 2020. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.314.4

[19] V. Arora, R. K. Bhatia, and M. Singh, "A systematic review of approaches for testing concurrent programs," *Concurr. Comput. Pract. Exp.*, vol. 28, no. 5, pp. 1572–1611, 2016. [Online]. Available: https://doi.org/10.1002/cpe.3711

[20] S. Taheri, I. Briggs, M. Burtscher, and G. Gopalakrishnan, "Difftrace: Efficient whole-program trace analysis and diffing for debugging," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, 2019, pp. 1–12.

[21] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 110–121. [Online]. Available: https://doi.org/10.1145/1040305.1040315

[22] P. Thomson, A. F. Donaldson, and A. Betts, "Concurrency testing using schedule bounding: An empirical study," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 15–28. [Online]. Available: https://doi.org/10.1145/2555243.2555260

[23] M. Emmi, S. Qadeer, and Z. Rakamarić, "Delay-bounded scheduling," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 411–422. [Online]. Available: https://doi.org/10.1145/1926385.1926432

[24] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: Association for Computing Machinery, 2010, p. 167–178. [Online]. Available: https://doi.org/10.1145/1736020.1736040

[25] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 446–455. [Online]. Available: https://doi.org/10.1145/1250734.1250785

[26] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," ser. OOPSLA '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 485–502. [Online]. Available: https://doi.org/10.1145/2384616.2384651

[27] P. Joshi, M. Naik, C.-S. Park, and K. Sen, "Calfuzzer: An extensible active testing framework for concurrent programs," in *Computer Aided Verification*, A. Bouajjani and O. Maler, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 675–681.

[28] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded java program test generation," in *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, ser. JGI '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 181. [Online]. Available: https://doi.org/10.1145/376656.376848

[29] O. Edelstein, E. Farchi, E. Goldin, Y. Nir-Buchbinder, G. Ratsaby, and S. Ur, "Framework for testing multithreaded java programs," *Concurrency and Computation: Practice and Experience*, vol. 15, 2003.

[30] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold, "Testing concurrent programs to achieve high synchronization coverage," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: Association for Computing Machinery, 2012, p. 210–220. [Online]. Available: https://doi.org/10.1145/2338965.2336779

[31] M. Christakis, A. Gotovos, and K. Sagonas, "Systematic testing for detecting concurrency errors in erlang programs," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013, pp. 154–163.

[32] X. Yuan and J. Yang, "Effective concurrency testing for distributed systems," ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1141–1156. [Online]. Available: https://doi.org/10.1145/3373376.3378484

[33] E. Trainin, Y. Nir-Buchbinder, R. Tzoref-Brill, A. Zlotnick, S. Ur, and E. Farchi, "Forcing small models of conditions on program interleaving for detection of concurrent bugs," in *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, ser. PADTAD '09. New York, NY, USA: Association for Computing Machinery, 2009. [Online]. Available: https://doi.org/10.1145/1639622.1639629

[34] J. Yu and S. Narayanasamy, "A case for an interleaving constrained shared-memory multi-processor," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 325–336. [Online]. Available: https://doi.org/10.1145/1555754.1555796
[35] "The go memory model," 2014. [Online]. Available: https://golang.org/ref/mem
[36] "Go developer survey 2019 results." [Online]. Available: https://blog.golang.org/survey2019-results
[37] [Online]. Available: https://github.com/moby/moby/issues/28405
[38] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur, "Applications of synchronization coverage," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 206–212. [Online]. Available: https://doi.org/10.1145/1065944.1065972
[39] Golang, "Command trace." [Online]. Available: https://golang.org/cmd/trace/
[40] Golang, "Package ast." [Online]. Available: https://golang.org/pkg/go/ast/
[41] Golang, "Package trace." [Online]. Available: https://golang.org/pkg/runtime/trace/
[42] S. M. Russ Cox, "Profiling go programs," 2013. [Online]. Available: https://blog.golang.org/pprof
[43] [Online]. Available: https://golang.org/src/internal/trace/parser.go
[44] GOAT, "Supporting material for iiswc submission." [Online]. Available: https://anonymous.4open.science/r/iiswc_goat_visualizations
[45] "Online deadlock detection in go (golang)." [Online]. Available: https://github.com/sasha-s/go-deadlock
[46] "Uber goleak." [Online]. Available: https://github.com/uber-go/goleak
[47] N. Dilley and J. Lange, "An empirical study of messaging passing concurrency in go projects," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 377–387.