

Dynamic Analysis of Concurrent Go-Programs

Bachelorarbeit - Kolloquium

Erik Kassubek

Institut für Informatik
Albert-Ludwigs-Universität Freiburg

14.02.2023

- Go-Routine
 - leichtgewichtiger Thread
 - ermöglicht Nebenläufigkeit
- Mutexe (Locks)
 - Synchronisationsmechanismus
 - Löst das Problem des gegenseitigen Ausschluss
- Channel
 - Synchronisationsmechanismus
 - Ermöglichen Kommunikation zwischen Routinen

```
func main() {  
    var m sync.Mutex  
  
    go func() {  
        m.Lock()  
        a() // kritischer Abschnitt  
        m.Unlock()  
    }()  
  
    m.Lock()  
    b() // kritischer Abschnitt  
    m.Unlock()  
}
```

Mutex

```
func main() {  
    var m sync.Mutex  
  
    go func() {  
        m.Lock()  
        a() // kritischer Abschnitt  
        m.Unlock()  
    }()  
  
    m.Lock()  
    b() // kritischer Abschnitt  
    m.Unlock()  
}
```

```
func main() {  
    var m sync.RWMutex  
  
    go func() {  
        m.RLock()  
        read()  
        m.RUnlock()  
    }()  
  
    go func() {  
        m.RLock()  
        read()  
        m.RUnlock()  
    }()  
  
    m.Lock()  
    write()  
    m.Unlock()  
}
```

Mutex

```
func main() {  
    m sync.Mutex  
    n sync.Mutex  
  
    go func() {  
        m.Lock()  
        n.Lock()  
        n.Unlock()  
        m.Unlock()  
    }()  
  
    n.Lock()  
    m.Lock()  
    m.Unlock()  
    n.Unlock()  
}
```

Mutex

```
func main() {  
    m sync.Mutex  
    n sync.Mutex  
  
    go func() {  
        m.Lock()  
        n.Lock()  
        n.Unlock()  
        m.Unlock()  
    }()  
  
    n.Lock()  
    m.Lock()  
    m.Unlock()  
    n.Unlock()  
}
```

```
func main() {  
    var m sync.Mutex  
  
    m.Lock()  
    m.Lock()  
}
```

Channel - Unbuffered

```
func main() {  
    c := make(chan int)  
  
    go func() {  
        c <- 1  
    }()  
  
    <- c  
}
```

Channel - Unbuffered

```
func main() {  
    c := make(chan int)  
  
    go func() {  
        c <- 1  
    }()  
  
    <- c  
}
```

```
func main() {  
    c := make(chan int)  
  
    go func() {  
        c <- 1  
    }()  
  
    go func() {  
        <- c  
    }()  
  
    <- c  
}
```


Channel - Buffered

```
func main() {  
    c := make(chan int, 2)  
    d := make(chan int)  
  
    go func() {  
        c <- 1  
        c <- 1  
        d <- 1  
    }()  
  
    <- d  
    <- c  
    <- c  
}
```

Channel - Buffered

```
func main() {  
    c := make(chan int, 2)  
    d := make(chan int)  
  
    go func() {  
        c <- 1  
        c <- 1  
        d <- 1  
    }()  
  
    <- d  
    <- c  
    <- c  
}
```

```
func main() {  
    c := make(chan int, 1)  
    d := make(chan int)  
  
    go func() {  
        c <- 1  
        c <- 1  
        d <- 1  
    }()  
  
    <- d  
    <- c  
    <- c  
}
```

- Schließt Channel \Rightarrow keine weiter Kommunikation möglich
- Send auf geschlossenem Channel \Rightarrow Laufzeitfehler

Channel - Close

- Schließt Channel \Rightarrow keine weitere Kommunikation möglich
- Send auf geschlossenem Channel \Rightarrow Laufzeitfehler

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

Channel - Select

- Wartet gleichzeitig auf mehrere Channel-Operationen
- Erste ausführbare Funktion wird ausgeführt
- Wenn mehrere gleichzeitig \Rightarrow Zufälliger Case
- Ohne Default \Rightarrow blockiert

```
go func() {  
    select {  
        case c <- 1:  
            func1()  
        case a := <- d:  
            func2(a)  
        default:  
            func3()  
    }  
}()
```

- Entwicklung und Implementierung eines Detektors zur
 - Erkennung von problematischen Situationen
 - Analyse von problematischen Situationen

- Entwicklung und Implementierung eines Detektors zur
 - Erkennung von problematischen Situationen
 - Analyse von problematischen Situationen
- Dynamische Analyse
 - Instrumentierung
 - Programm (mehrfach) ausführen
 - Verhalten aufzeichnen \Rightarrow Trace
 - Trace analysieren

- Speichert Ablauf der notwendigen Informationen
- Ein Trace pro Routine
- Ermöglicht Rekonstruktion des Programmablaufs

- Veränderung des Programmcodes
- Ersetze Mutexe/Channel durch eigene Objekte
- Funktionen auf Objekten:
 - Ausführung der eigentlichen Operation
 - Aufzeichnung der Operation
- Aufzeichnung von Fork
- Aufzeichnung / Kontrolle über Select

- `signal(ts, nr)` in Trace
- Erzeuge neuen Trace
- Starte der neuen Routine
- `wait(ts, nr)` in neuen Trace
- Ausführen der Routine

- (Try)(R)Lock:
 - `lock(ts, mu.id, <-|t|r|tr>, <0|1>)`
 - Führe Lock aus
- (Try)(R)Unlock:
 - `unlock(ts, mu.id)` in Trace
 - Führe Lock aus

- Send: (`c <- i`):
 - `pre(ts, c.id, true)` in Trace
 - `c <- {i, r.id, ts}`
 - `post(ts', c.id, true, ts)` in Trace
- Receive (`a := <- c`):
 - `pre(ts, c.id, false)` in Trace
 - `{i, id_send, ts_send} := <-c`
 - `post(ts', ch.id, false, id_send, ts_send)` in Trace
 - `a := i`
- Close (`close(c)`)
 - `close(ts, c.id)` in Trace
 - `close(c)`

```
go func() {  
    select {  
        case a := <- c:  
            func1(a)  
        case d <- 1:  
            func2()  
    }  
}()
```

Instrumentierung - Select

```
goChan.PreSelect(false, c.GetIdPre(true),
                  d.GetIdPre(false))
sel_RAjWwhTH := goChan.BuildMessage(1)
switch goChan.FetchOrder[1] {
case 0:
    select {
    case sel_VIBzgbaiCM := <-c.GetChan():
        c.Post(true, sel_VIBzgbaiCM)
        a := sel_VIBzgbaiCM.GetInfo()
        func1(a)
    case <-time.After(time.Second):
        ....
    }
}
```

```
case 1:
    select {
    case d.GetChan() <- sel_RAjWwhTH:
        d.Post(false, sel_RAjWwhTH)
        func2()
    case <-time.After(time.Second):
        ....
    }
default:
    ....
}
```

- Funktionen in Package implementiert
- Automatisierte Ersetzung im Programmcode
 - Traversierend des AST
 - Veränderung des AST wenn notwendig
- Erzeugung einer neuen Main-Datei zur Ausführung

- Mehrfache Ausführung für verschiedene Select-Pfade
- Zufällige Wahl der Pfade (ohne Doppelung)
- Analyse nach jeder Ausführung
- Konsolidierung der Ergebnisse am Ende
- Automatischer Abbruch wenn Laufzeit zu lange

- Suche nach doppeltem Locking
 - Lock-Operation als letztes Element in Trace
 - Traversiere Trace rückwärts
 - Wenn Lock auf selben Mutex vor Unlock auf selben Mutex \Rightarrow doppeltes Locking
 - Kein Deadlock wenn beide RLock

- Suche nach zyklischem Locking
 - Aufbau von Lock-Bäumen
 - Suchen nach Zyklen in Lock-Bäumen

Detektion von Deadlocks: Lock-Bäume

```
func lockTree(){  
    var x,y,z sync.Mutex  
  
    // R1  
    go func(){  
        y.Lock()  
        z.Lock()  
        z.Unlock()  
        x.Lock()  
        x.Unlock()  
        y.Unlock()  
    }  
  
    // R2  
    go func(){  
        x.Lock()  
        z.Lock()  
        z.Unlock()  
        x.Unlock()  
    }  
  
    // R3  
    go func(){  
        z.Lock()  
        y.Lock()  
        y.Unlock()  
        z.Unlock()  
    }  
}
```

Detektion von Deadlocks: Lock-Bäume

```
func lockTree(){  
    var x,y,z sync.Mutex  
  
    // R1  
    go func(){  
        y.Lock()  
        z.Lock()  
        z.Unlock()  
        x.Lock()  
        x.Unlock()  
        y.Unlock()  
    }  
  
    // R2  
    go func(){  
        x.Lock()  
        z.Lock()  
        z.Unlock()  
        x.Unlock()  
    }  
  
    // R3  
    go func(){  
        z.Lock()  
        y.Lock()  
        y.Unlock()  
        z.Unlock()  
    }  
}
```

R1

R2

R3

Detektion von Deadlocks: Lock-Bäume

```
func lockTree(){  
    var x,y,z sync.Mutex
```

```
    // R1
```

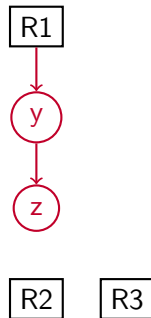
```
    go func(){  
        y.Lock()  
        z.Lock()  
        z.Unlock()  
        x.Lock()  
        x.Unlock()  
        y.Unlock()  
    }
```

```
    // R2
```

```
    go func(){  
        x.Lock()  
        z.Lock()  
        z.Unlock()  
        x.Unlock()  
    }
```

```
    // R3
```

```
    go func(){  
        z.Lock()  
        y.Lock()  
        y.Unlock()  
        z.Unlock()  
    }  
}
```



Detektion von Deadlocks: Lock-Bäume

```
func lockTree(){  
    var x,y,z sync.Mutex
```

```
    // R1
```

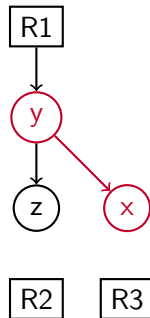
```
    go func(){  
        y.Lock()  
        z.Lock()  
        z.Unlock()  
        x.Lock()  
        x.Unlock()  
        y.Unlock()  
    }
```

```
    // R2
```

```
    go func(){  
        x.Lock()  
        z.Lock()  
        z.Unlock()  
        x.Unlock()  
    }
```

```
    // R3
```

```
    go func(){  
        z.Lock()  
        y.Lock()  
        y.Unlock()  
        z.Unlock()  
    }
```



Detektion von Deadlocks: Lock-Bäume

```
func lockTree(){  
    var x,y,z sync.Mutex
```

```
// R1
```

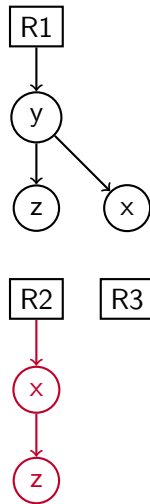
```
go func(){  
    y.Lock()  
    z.Lock()  
    z.Unlock()  
    x.Lock()  
    x.Unlock()  
    y.Unlock()  
}
```

```
// R2
```

```
go func(){  
    x.Lock()  
    z.Lock()  
    z.Unlock()  
    x.Unlock()  
}
```

```
// R3
```

```
go func(){  
    z.Lock()  
    y.Lock()  
    y.Unlock()  
    z.Unlock()  
}
```



Detektion von Deadlocks: Lock-Bäume

```
func lockTree(){  
    var x,y,z sync.Mutex
```

```
// R1
```

```
go func(){  
    y.Lock()  
    z.Lock()  
    z.Unlock()  
    x.Lock()  
    x.Unlock()  
    y.Unlock()  
}
```

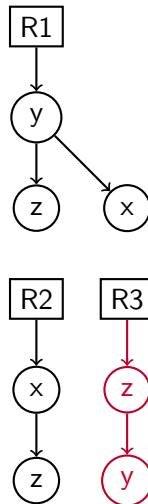
```
// R2
```

```
go func(){  
    x.Lock()  
    z.Lock()  
    z.Unlock()  
    x.Unlock()  
}
```

```
// R3
```

```
go func(){  
    z.Lock()  
    y.Lock()  
    y.Unlock()  
    z.Unlock()  
}
```

```
}
```



Detektion von Deadlocks: Lock-Bäume

```
func lockTree(){  
    var x,y,z sync.Mutex
```

```
// R1
```

```
go func(){  
    y.Lock()  
    z.Lock()  
    z.Unlock()  
    x.Lock()  
    x.Unlock()  
    y.Unlock()  
}
```

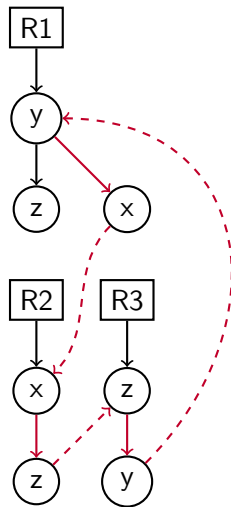
```
// R2
```

```
go func(){  
    x.Lock()  
    z.Lock()  
    z.Unlock()  
    x.Unlock()  
}
```

```
// R3
```

```
go func(){  
    z.Lock()  
    y.Lock()  
    y.Unlock()  
    z.Unlock()  
}
```

```
}
```



- Zyklus nur gültig wenn:
 - keine zwei R-Lock hintereinander
 - Locks in Zyklus dürfen keine gemeinsamen Vorfahren haben, außer beide Vorfahren sind RLock

- Suche nach potenziellem Ablauf mit
 - Send/Receive ohne Kommunikationspartner
 - potenzielles Send auf geschlossenem Channel

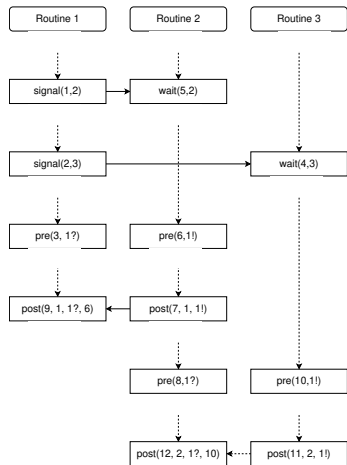
- Berechne Pre- und Post-Vectorclocks
- Bestimme vectorclock-annotated Trace
- Bestimme Operationen mit unvergleichbaren Vectorclocks
- Betrachte potenzielle Abläufe
- Suche nach Send/Receive ohne Kommunikationspartner und Send auf Closed

Analyse - Channel

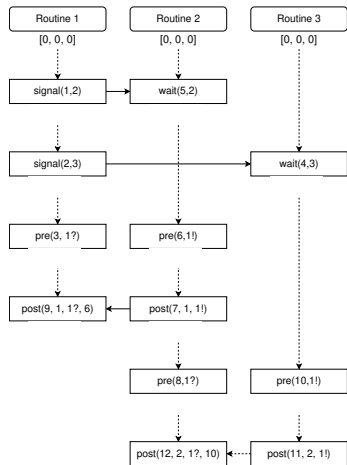
```
func main() {  
    x := make(chan int)  
  
    go func() {  
        x <- 1  
        <- x  
    }()  
  
    go func() {  
        x <- 1  
    }()  
  
    <-x  
}
```

$[[\text{signal}(1, 2), \text{signal}(2, 3), \text{pre}(3, 1?), \text{post}(9, 1, 1?, 6)]$
 $[\text{wait}(5, 2), \text{pre}(6, 1!), \text{post}(7, 1, 1!), \text{pre}(8, 1?), \text{post}(12, 2, 1?, 10)]$
 $[\text{wait}(4, 3), \text{pre}(10, 1!), \text{post}(11, 2, 1!)]]$

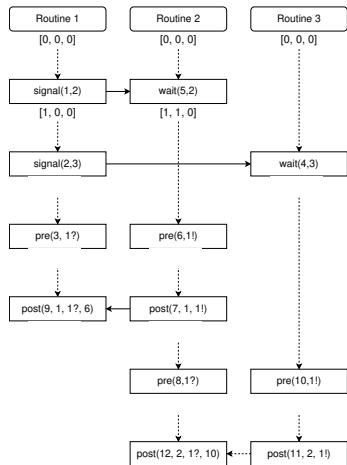
Analyse - Channel



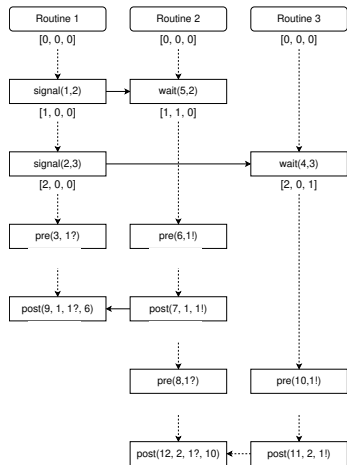
Analyse - Channel



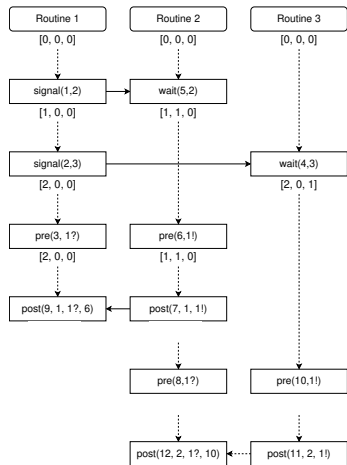
Analyse - Channel



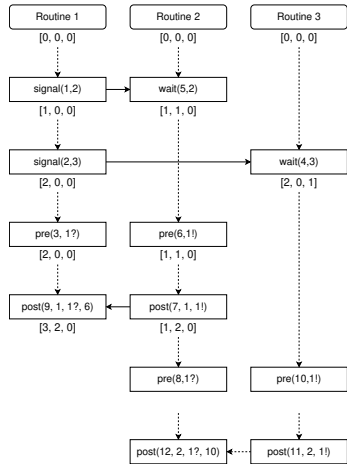
Analyse - Channel



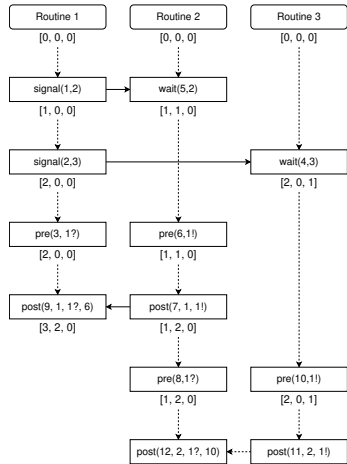
Analyse - Channel



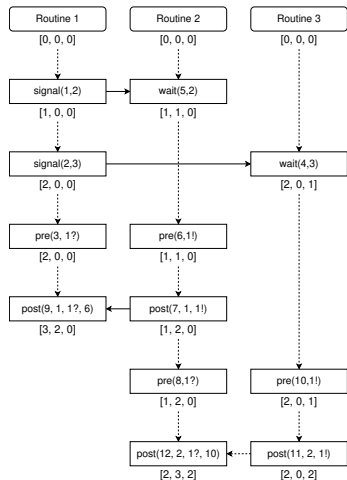
Analyse - Channel



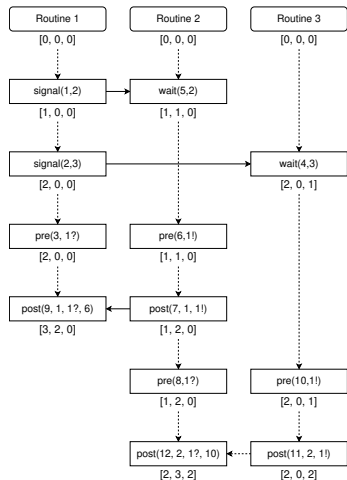
Analyse - Channel



Analyse - Channel



Analyse - Channel



$$[[[2,0,0] x? [3,2,0]]$$

$$[[1,1,0] x! [1,2,0], [1,2,0] x? [2,3,2]]$$

$$[[2,0,1] x! [2,0,2]]]$$

- x ist Ursache von y wenn
 - $x \rightarrow y \Leftrightarrow VC(x) < VC(y) \Leftrightarrow \forall z (VC(x)_z \leq VC(y)_z) \wedge \exists z' (VC(x)_{z'} < VC(y)_{z'})$
- x ist nebenläufig/unvergleichbar mit y wenn
 - $x || y \Leftrightarrow VC(x) \not\leq VC(y) \Leftrightarrow VC(x) \not\leq VC(y) \wedge VC(y) \not\leq VC(x)$

- Potenzielle Kommunikationspartner auf ungepufferten Channels:
 - Paar von Send-Receive auf selbem Channel
 - Pre- oder Post-Vectorclock unvergleichbar
- Potenzielle Kommunikationspartner auf gepufferten Channels:
 - Paar von Send-Receive auf selbem Channel
 - $\#s_{<} + \#s_{\not<} \geq \#r_{<} + \#r_{\not<}$
 - $\#s_{<} \leq \#r_{<}$

$$\begin{aligned} & [[[2,0,0] x_1 ? [3,2,0]] \\ & \quad [[1,1,0] x_2 ! [1,2,0] , [1,2,0] x_3 ? [2,3,2]] \\ & \quad [[2,0,1] x_4 ! [2,0,2]]] \end{aligned}$$

$$\begin{aligned} & [[[2,0,0] x_1 ?^{[3,2,0]}] \\ & \quad [[1,1,0] x_2 !^{[1,2,0]}, [1,2,0] x_3 ?^{[2,3,2]}] \\ & \quad [[2,0,1] x_4 !^{[2,0,2]}]] \end{aligned}$$
$$x_2! \rightarrow x_1?$$
$$x_4! \rightarrow x_1?$$
$$x_4! \rightarrow x_3?$$

$$\begin{aligned} & [[2,0,0]x_1?[3,2,0]] \\ & [[1,1,0]x_2![1,2,0], [1,2,0]x_3?[2,3,2]] \\ & [[2,0,1]x_4![2,0,2]] \end{aligned}$$
$$x_2! \rightarrow x_1?$$
$$x_4! \rightarrow x_1?$$
$$x_4! \rightarrow x_3?$$
$$x_2! \rightarrow x_1?$$
$$x_4! \rightarrow x_3?$$

$$\begin{aligned} & [[[2,0,0] x_1 ?^{[3,2,0]}] \\ & \quad [[1,1,0] x_2 !^{[1,2,0]}, [1,2,0] x_3 ?^{[2,3,2]}] \\ & \quad [[2,0,1] x_4 !^{[2,0,2]}]] \end{aligned}$$
$$x_2! \rightarrow x_1?$$
$$x_4! \rightarrow x_1?$$
$$x_4! \rightarrow x_3?$$
$$x_2! \rightarrow x_1?$$
$$x_4! \rightarrow x_3?$$
$$x_4! \rightarrow x_1?$$
$$x_2! \rightarrow \text{false}$$

$$[[[2,0,0]x_1?[3,2,0]]$$
$$[[1,1,0]x_2![1,2,0], [1,2,0]x_3?[2,3,2]]$$
$$[[2,0,1]x_4![2,0,2]]]$$
$$x_2! \rightarrow x_1?$$
$$x_4! \rightarrow x_1?$$
$$x_4! \rightarrow x_3?$$
$$x_2! \rightarrow x_1?$$
$$x_4! \rightarrow x_3?$$
$$x_1? \leftarrow x_2!$$
$$x_3? \leftarrow x_4!$$
$$x_4! \rightarrow x_1?$$
$$x_2! \rightarrow \text{!}$$

$$[[[2,0,0]x_1?[3,2,0]]$$
$$[[1,1,0]x_2![1,2,0], [1,2,0]x_3?[2,3,2]]$$
$$[[2,0,1]x_4![2,0,2]]]$$
$$x_2! \rightarrow x_1?$$
$$x_4! \rightarrow x_1?$$
$$x_4! \rightarrow x_3?$$
$$x_2! \rightarrow x_1?$$
$$x_4! \rightarrow x_3?$$
$$x_4! \rightarrow x_1?$$
$$x_2! \rightarrow \downarrow$$
$$x_1? \leftarrow x_2!$$
$$x_3? \leftarrow x_4!$$
$$x_1? \leftarrow x_4!$$
$$x_3? \leftarrow \downarrow$$

- Potenzielles Send auf Closed Channel:
 - Pre- oder Post-VC von Send unvergleichbar mit VC von Close
 - Wenn Send ist Ursache von Close \Rightarrow Send auf Close tritt während Durchlauf auf \Rightarrow erzeugt Laufzeitfehler

- Dynamischer Detektor zur Erkennung von Concurrency-Bugs
 - Mutex-Bugs mit Lock-Bäumen
 - Channel-Bugs mit Vectorclock
- Implementierung für Instrumenter und Detektor:
 - <https://github.com/ErikKassubek/GoChan>

- [1] M. Sulzmann und K. Stadtmüller, „Two-Phase Dynamic Analysis of Message-Passing Go Programs based on Vector Clocks,” *CoRR*, Jg. abs/1807.03585, 2018. arXiv: 1807.03585. Adresse: <http://arxiv.org/abs/1807.03585>.
- [2] Z. Liu, S. Zhu, B. Qin, H. Chen und L. Song, „Automatically Detecting and Fixing Concurrency Bugs in Go Software Systems,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Ser. ASPLOS '21, Virtual, USA: Association for Computing Machinery, 2021, S. 616–629, ISBN: 9781450383172. DOI: 10.1145/3445814.3446756. Adresse: <https://doi.org/10.1145/3445814.3446756>.
- [3] Z. Liu, S. Xia, Y. Liang, L. Song und H. Hu, „Who Goes First? Detecting Go Concurrency Bugs via Message Reordering,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Ser. ASPLOS '22, Lausanne, Switzerland: Association for Computing Machinery, 2022, S. 888–902, ISBN: 9781450392051. DOI: 10.1145/3503222.3507753. Adresse: <https://doi.org/10.1145/3503222.3507753>.
- [4] S. Taheri und G. Gopalakrishnan, „Automated Dynamic Concurrency Analysis for Go,” *arXiv e-prints*, arXiv:2105.11064, arXiv:2105.11064, Mai 2021. arXiv: 2105.11064 [cs.DC].
- [5] S. Taheri und G. Gopalakrishnan, „GoAT: Automated Concurrency Analysis and Debugging Tool for Go,” in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, 2021, S. 138–150. DOI: 10.1109/IISWC53511.2021.00023.
- [6] The Go Team, *Go Documentation: trace*, <https://pkg.go.dev/cmd/trace>, 2022.
- [7] J. Zhou, S. Silvestro, H. Liu, Y. Cai und T. Liu, „UNDEAD: Detecting and preventing deadlocks in production software,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2017, S. 729–740. DOI: 10.1109/ASE.2017.8115684.
- [8] P. Joshi, C.-S. Park, K. Sen und M. Naik, „A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks,” *SIGPLAN Not.*, Jg. 44, Nr. 6, S. 110–120, Juni 2009, ISSN: 0362-1340. Adresse: <https://doi.org/10.1145/1543135.1542489>.

- [9] C. J. Fidge, „Timestamps in Message-Passing Systems That Preserve the Partial Ordering,“, 1988.
- [10] The Go Team, Go, <https://go.dev/>, 2022.
- [11] R. Agarwal, L. Wang und S. D. Stoller, „Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring,“ in *Hardware and Software, Verification and Testing*, S. Ur, E. Bin und Y. Wolfsthal, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, S. 191–207, ISBN: 978-3-540-32605-2.
- [12] T. Yuan, G. Li, J. Lu, C. Liu, L. Li und J. Xue, „GoBench: A Benchmark Suite of Real-World Go Concurrency Bugs,“ in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, <https://github.com/timmyyuan/gobench>, 2021, S. 187–199. DOI: 10.1109/CGO51591.2021.9370317.
- [13] A. Gerrand, *Share Memory By Communicating*, <https://go.dev/blog/codelab-share>, 2010.
- [14] B. Ray, D. Posnett, V. Filkov und P. Devanbu, „A large scale study of programming languages and code quality in github,“ *Proc. FSE 2014*, S. 155–165, Nov. 2014. DOI: 10.1145/2635868.2635922.