Bachelorarbeit

Dynamic Analysis of Message-Passing Go Programs

Erik Daniel Kassubek

Gutachter: Prof. Dr. Thiemann

Betreuer: Prof. Dr. Thiemann

Prof. Dr. Sulzmann

Albert-Ludwigs-Universität Freiburg

Technische Fakultät

Institut für Informatik

Lehrstuhl für Programmiersprachen

14. Februar 2023

Bearbeitungszeit

14. 11. 2022 – 14. 02. 2023

Gutachter

Prof. Dr. Thiemann

Betreuer

Prof. Dr. Thiemann

Prof. Dr. Sulzmann

ERKLÄRUNG

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum	Unterschrift

Zusammenfassung

(TODO: Zusammenfassung (Abstract) schreiben)

Inhaltsverzeichnis

1 Einführung

(TODO: Einführung schreiben)

2 Hintergrund

(TODO: Einführung)

2.1 Mutexe

Mutexe (auch Locks genannt) gehören zu den am weitesten Verbreitenten Mechanismen zur Synchronisierung nebenläufiger Programme [1]. Sie bilden eine Lösung für das Problem des kritischen Abschnitts. In solch einem kritischen Abschnitt darf sich immer nur maximal eine Routine gleichzeitig aufhalten. Sie werden vor allem verwendet um zu verhinder, dass mehrere Routinen gleichzeitig auf die selbe globale Datenstruktur zugreifen. Solche Abschnitte werden nun mit einem Lock umschlossen. Diese besitzen die folgenden Operationen:

- Lock: Die Lock-Operation wird vor dem Eintritt in einen kritischen Bereich aufgerufen. Sie versucht den Mutex zu beanspruchen. Wird es im moment von keiner Routine gehalten, wird der Mutex geschlossen und der kritische Bereich ausgeführt. Wird der Mutex bereits von einer anderen Routine gehalten, muss die Routine, in welcher die Lock-Operation ausgeführt worden ist so lange warten, bis der Mutex wieder freigegeben wird und beansprucht werden kann.

- TryLock: Eine TryLock Operation versucht wie die Lock-Operation einen Mutex zu beanspruchen. Anders als bei Lock wird die Routine allerdings nicht blockiert, wenn eine Beanspruchung nicht direkt möglich ist. Es wird lediglich zurückgegeben, ob die erfolgreich war oder nicht. Ein Programmierer kann in diesem Fall selber entscheiden,

wie das Programm weiter ablaufen soll, ob z.B. der kritische Bereich übersprungen werden soll.

- Unlock: Diese Operation gibt einen gehaltenen Mutex wieder frei. Er k kann nur von der Routine freigegeben werden, welche das Lock momentan hält. Der Versuch ein Lock freizugeben, welches nicht gehalten wird führt zu einem Laufzeitfehler.

Mutexe können mit (Try)RLock Operationen zu RW-Mutex erweitert werden. Dabei kann der selbe Mutex von mehreren Routinen gleichzeitig über (Try)RLock-Operationen beansprucht bzw. gehalten werden. Es ist allerdings nicht, dass ein Mutex gleichzeitig über eine RLock- und eine Lock-Operation gehalten werden. Diese können z.B. verwendet werden, wenn mehrere Routinen gleichzeitig lesend auf eine Datenstruktur zugreifen dürfen, allerdings nur, wenn gerade keine Routine schreibend auf die selbe Datenstruktur zugreift.

2.2 Channel

Chap:Back-Sec:Chann Channel [2] ermöglichen es Routinen untereinander zu kommunizieren. Ein Channel `c` kann Daten `d` senden (`c <- d`) und empfangen (`<- c`). Das genaue Verhalten der Channel hängt dabei davon ab, ob es sich um einen gebufferten oder ungebufferten Channel handelt.
Bei einem ungebufferten Channel müssen Send- und Receive-Operation gleichzeitig ablaufen. Möchte eine Routine `R` Daten auf einem Channel `c` senden, ist aber keine Routine bereit die Daten von `c` zu empfangen, dann muss `R` so lange warten, bis eine andere Routine zu einem Receive-Statement des Channels `c` kommt. Das selbe gilt auch, wenn eine Routine an ein Receive-Statement eines Channels kommt auf welchem momentan nicht gesendet wird. Bei einem gebufferten Channel der Größe n können bis zu n Nachrichten zwischengespeichert werden. Dies bedeutet, dass Send und Receive nicht mehr gleichzeitig ablaufen können. Eine Operation muss hierbei nur dann vor einer Operation warten, wenn eine Send-Operation auf einem vollen Channel, oder eine Receive-Operation auf einem leeren Channel ausgeführt wird. In allen anderen Fällen kann die Operation die Nachricht in den Buffer schreiben, bzw.

eine Nachricht aus dem Buffer lesen. Der Buffer bildet dabei eine FIFO-Queue, dass heißt, es wird bei einem Receive immer die ältest in dem Buffer vorhandene Nachricht ausgegeben. Sowohl bei gebufferten als auch bei ungebufferten Channels kann es zu Situationen kommen, in denen mehrere Routinen gleichzeitig auf dem selben Channel auf eine Nachricht warten. Wird nun auf diesem Channel gesendet wird eine der Routinen pseudo-zufällig ausgewählt, um die Nachricht zu empfangen.

Go macht es mit der Select-Operation möglich, auf die erste von mehreren erfolgreichen Channel-Operationen gleichzeitig zu warten. Ein Beispiel für solch ein Select befindet sich in Abb. 1. In diesem Beispiel besitzt die Select-Operation 3 Cases und einen Default-Case. Die

```
1  x := make(chan int)
2  y := make(chan int)
3  z := make(chan int)
4
5  select {
6    case <- x:
7      fmt.Println("x")
8    case a := <- y:
9      fmt.Println(a)
10   case  z <- 5:
11     fmt.Println(5)
12   default:
13     fmt.Println("default")
14 }
```

Abb. 1: Beispielprogramm zyklisches Locking

Cases bestehen aus verschiedenen Channel-Operationen (Receive auf Channel, Receive auf Channel mit direkter Variablendeklaration und Send auf Channel). Das Select-Statement probiert nun die Cases in einer zufälligen Reihenfolge aus. Findet es einen Case, in dem die Operation ausgeführt werden kann, wird die Operation, sowie der darunter stehende Block (Print-Statement) ausgeführt. Der Default-Case wird ausgeführt, wenn keiner der anderen Cases ausgeführt werden kann. Er ist allerdings nicht notwendig. Besitzt das Select-Statement keinen Default-Case, dann blockiert die Routine so lange, bis einer der Cases ausgeführt werden kann.

2.3 Trace

(TODO: Warum trace?) Um ein Program analysieren zu können, soll der Ablauf eines Programmdurchlaufs (Trace) aufgezeichnet werden. Der Aufbau des Trace basiert auf [3]. Er wird aber um Informationen über Locks erweitert. Der Trace wird für jede Routine separat aufgezeichnet. Außerdem wird, anders als in [3] ein globaler Program-Counter für alle Routinen und nicht ein separater Counter für jede Routine verwendet. Dies ermöglicht es bessere Rückschlüsse über den genauen Ablauf des Programms zu ziehen. Die Syntax des Traces in EBNF gibt sich folgendermaßen:

$$
\begin{array}{lcll}
T & = & '' [\,'', \{\, U \,\}, '']\,''; & \text{Trace} \\
U & = & '' [\,'', \{\, t \,\}, '']\,''; & \text{lokaler Trace} \\
t & = & signal(t,i) \mid wait(t,i) \mid pre(t,as) \mid post(t,i,x!) & \text{Event} \\
 & & \mid post(t,i,x?,t') \mid post(t,default) & \\
 & & \mid lock(t,y,b,c) \mid unlock(t,y); & \\
a & = & x,\ (''\,!\,'' \mid ''\,?\,''); & \\
as & = & a \mid (\{\, a \,\}, [\,''\, default\, '']); & \\
b & = & ''-'' \mid ''t'' \mid ''r'' \mid ''tr'' & \\
c & = & ''0'' \mid ''1'' &
\end{array}
$$

wobei i die Id einer Routine, t einen globalen Zeitstempel, x die Id eines Channels und y die Id eines Locks darstellt. Die Events haben dabei folgende Bedeutung:

- `signal(t, i)`: In der momentanen Routine wurde eine Fork-Operation ausgeführt, d.h. eine neue Routine mit Id i wurde erzeugt.
- `wait(t, i)`: Die momentane Routine mit Id i wurde soeben erzeugt. Dies ist in allen Routinen außer der Main-Routine das erste Event in ihrem lokalen Trace.
- `pre(t, as)`: Die Routine ist an einer Send- oder Receive-Operation eines Channels oder an einem Select-Statement angekommen, dieses wurde aber noch nicht ausgeführt.

Das Argument as gibt dabei die Richtung und den Channel an. Ist $as = x!$, dann befindet sich der Trace vor einer Send-Operation, bei $as = x?$ vor einer Receive-Operation. Bei einem Select-Statement ist as eine Liste aller Channels für die es einen Case in dem Statement gibt. Besitzt das Statement einen Default-Case, wird dieser ebenfalls in diese List aufgenommen.

- `post(t, i, x!)`: Dieses Event wird in dem Trace gespeichert, nachdem eine Send-Operation auf x erfolgreich abgeschlossen wurde. i gibt dabei die Id der sendenden Routine an.

- `post(t, i, x?, t')`: Dieses Event wird in dem Trace gespeichert, nachdem eine Receive-Operation des Channels x erfolgreich abgeschlossen wurde. i gibt dabei die Id der sendenden Routine an. t' gibt den Zeitstempel an, welcher bei dem Pre-Event der sendenden Routine galt. Durch die Speicherung der Id und des Zeitstempels der sendenden Routine bei einer Receive-Operation lassen sich die Send- und Receive-Operationen eindeutig zueinander zuordnen.

- `post(t, default)`: Wird in einem Select-Statement der Default-Case ausgeführt, wird dies in dem Trace der entsprechenden Routine durch $post(t, default)$ gespeichert.

- `lock(t, y, b, c)`: Der Beanspruchungsversuch eines Locks mit id y wurde beendet. b gibt dabei die Art der Beanspruchung an. Bei $b = r$ war es eine R-Lock Operation, bei $b = t$ eine Try-Lock Operation und bei $b = tr$ ein Try-R-Lock Operation. Bei einer normalen Lock-Operation ist $b = -$. Bei einer Try-Lock Operation kann es passieren, dass die Operation beendet wird, ohne das das Lock gehalten wird. In diesem Fall wird c auf 0, und sonst auf 1 gesetzt.

- `unlock(t, y)`: Das Lock mit id y wurde zum Zeitpunkt t wieder freigegeben.

Man betrachte als Beispiel das folgende Programm in Go: Dieser ergibt den folgenden

```
1  func main() {
2      x := make(chan int)
3      y := make(chan int)
4      a := make(chan int)
5
6      var v sync.RWMutex
7
8      go func() {
9          v.Lock()
10         x <- 1
11         v.Unlock()
12     }()
13     go func() { y <- 1; <-x }()
14     go func() { <-y }()
15
16     select {
17     case <-a:
18     default:
19     }
20 }
```

Abb. 2: Beispielprogramm für Tracer

Trace:

$$[[signal(1,2),\ signal(2,3),\ signal(3,4),\ pre(4,a?,default),\ post(5,default)]$$

$$[wait(8,2),\ lock(9,1,-,1),\ pre(10,x!),\ post(16,2,x!,1),\ unlock(17,1)]$$

$$[wait(11,3),\ pre(12,y!),\ post(13,3,y!,1),\ pre(14,x?),\ post(15,2,x?,10,1)]$$

$$[wait(6,4),\ pre(7,y?),\ post(18,3,y?,12,1)]]$$

Aus diesem lässt sich der relevante Ablauf des Programms rekonstruieren. **(EXTEND: Tracer)**

2.4 Probleme und Problemerkennung

2.4.1 Mutex

Durch die Verwendung von Mutexen in nebenläufigen Programmen kann es zu sogenannten Deadlocks kommen. Blockieren sich dabei mehrere Routinen gegenseitig bezeichnen wir eine solche Situation als zyklisches Locking. Abb. 3 zeigt ein Beispiel, in welchem es zu zyklischem Locking kommen kann. Routine 0 und Routine 1 können dabei gleichzeitig

```
1  func main() {
2    var x sync.Mutex
3    var y sync.Mutex
4
5    go func() {
6      // Routine 1
7      x.Lock()
8      y.Lock()
9      y.Unlock()
10     x.Unlock()
11   }()
12
13   // Routine 0
14   y.Lock()
15   x.Lock()
16   x.Unlock()
17   y.Unlock()
18 }
```

Abb. 3: Beispielprogramm zyklisches Locking

ausgeführt werden. Man betrachte den Fall, in dem Zeile 7 und 14 gleichzeitig ausgeführt werden, also Lock `y` von Routine 0 und Lock `x` von Routine 1 gehalten wird. In diesem Fall kann in keiner der Routinen die nächste Zeile ausgeführt werden, da das jeweilige Locks, welches beansprucht werden soll bereits durch die andere Routine gehalten wird. Da sich diese Situation auch nicht von alleine auflösen kann, blockiert dass Programm, befindet sich also in einem zyklischen Deadlock.

Da solche Situationen nur in ganz besonderen Situation auftreten (in dem obigen Beispiel müssen Zeilen 7 und 14 genau gleichzeitig ausgeführt werden, ohne dass Zeile 8 oder

15 ausgeführt werden), muss ein Detektor, welcher vor solchen Situationen warnen soll, nicht nur tatsächliche Deadlocks, sondern vor allem potenzielle, also nicht tatsächlich aufgetretene Deadlocks erkennen. Die Erkennung der potenziellen Deadlocks basiert hierbei auf iGoodLock [4] und UNDEAD [1]. Dabei wird für jede Routine ein Lock-Baum aufgebaut. Jeder in der Routine vorkommende (RW-)Mutex m_i wird durch einen Knoten k_i in dem Baum repräsentiert. In diesem Baum gibt es nun eine gerichtete Kante von k_1 nach k_2, wenn m_1 das von der Routine momentan gehaltene Lock ist, welches zuletzt von der Routine beansprucht worden ist, währen das Lock m_2 beansprucht wird [5]. Ist es nun möglich, Knoten aus verschiedenen Bäumen, welche den selbe Mutex repräsentieren so zu verbinden, dass der entstehende Graph einen Zyklus enthält, bedeutet dies, dass in dem Programm zyklisches Locking möglich ist. Man betrachte dazu das Programm in Abb. 4 In Abb. 5 sind die entsprechenden Lock-Bäume, sowie der enthaltene Zyklus graphisch dargestellt. Es sei allerdings festzuhalten, dass besonders bei der Verwendung von RW-Locks nicht jeder Zyklus auch direkt zu einem potenziellen Deadlock führen kann. Wie solche false-positives verhinder werde können wird in dem Kapitel zur Implementierung des Detektors (4.1) noch genauer betrachtet.

2.4.2 Channel

Wie Mutexe können auch Channels zu Deadlocks führen. Diese treten auf, wenn ein Channel auf eine Send- oder Receive-Operation wartet, ohne dass während der Laufzeit des Programms ein Kommunikationspartner für die Operation gefunden wird. Anders als bei Mutexen, bei denen immer eine Lock- und eine Unlock-Operation fest zusammen gehören können Channel-Operationen je nach dem wie dass Programm genau anläuft verschieden Kommunikationspartner haben. Die Wahl der Kommunikationspaar kann außerdem, vor allem in Select-Statements quasi non-deterministisch ablaufen, was eine Voraussage von Potenziellen Deadlocks deutlich verkompliziert. Aus diesem Grund beschränken wir uns hierbei darauf tatsächlich auftretende Probleme zu erkennen, sie zu analysieren und dann, soweit möglich Hinweise über die Probleme zu liefern, welche die Erkennung und manuelle

```
1  func cyclicLockingExample {
2    var v Mutex
3    var w Mutex
4    var x Mutex
5    var y Mutex
6    var z Mutex
7
8    go func ( ) { // R1
9      v . Lock ( )
10     w . Lock ( )
11     w . Unlock ( )
12     v . Unlock ( )
13     y . Lock ( )
14     z . Lock ( )
15     z . Unlock ( )
16     x . Lock ( )
17     x . Unlock ( )
18     y . Unlock ( )
19   }()
20
21   go func ( ) { // R2
22     w . Lock ( )
23     x . Lock ( )
24     x . Unlock ( )
25     w . Unlock ( )
26   }()
27
28   go func ( ) { // R3
29     x . Lock ( )
30     v . Lock ( )
31     v . Unlock ( )
32     x . Unlock ( )
33   }
```

Abb. 4: Beispielprogramm zyklisches Locking

Beseitigung der Probleme erleichtern.

Ungepufferte Channel

Man betrachte das Programm in Abb. 6. Es gibt in diesem zwei mögliche Ausführungspfade. Man betrachtet zuerst den Fall, in dem 1 mit 3 synchronisiert. Da eine go-Routine automatisch abgebrochen wird, wenn die Main-Routine terminiert, entsteht hierbei kein Deadlock. Anders ist es, wenn 1 mit 2 synchronisiert. In diesem Fall wird die Main-Routine blockiert, ohne dass es eine Möglichkeit gibt, dass sie sich wieder befreit. Es kann also,

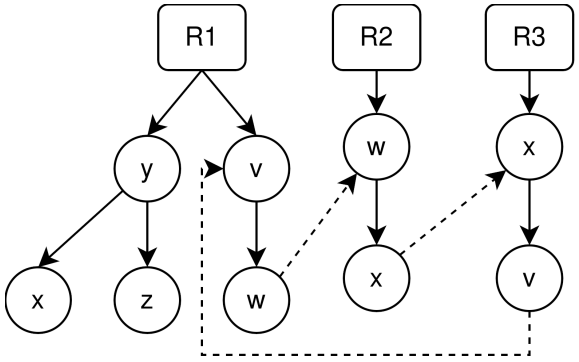

Abb. 5: Graphische Darstellung des Lock-Graphen für das Beispielprogramm in Abb. 4. Durch die gestrichelten Pfeile wird der enthaltenen Zyklus angezeigt.

```
1 func main() {
2   x := make(chan int, 0)
3
4   go func() { x <- 1}()   // 1
5   go func() { <- x }()    // 2
6   <- x                    // 3
7 }
```

Abb. 6: Beispielprogramm mit hängendem Channel

abhängig davon, ob 1 oder 2 die Nachricht erhält zu einem Deadlock kommen. Was allerdings beide Fälle gemeinsam haben ist, dass sie eine Channel-Operation besitzen, welche zwar gestartet, allerdings nie ausgeführt wird. In diesen Fällen gibt es in dem Trace eine Channel-Information, welche ein Pre- aber kein Post-Event besitzt. Solch eine Situation bezeichnen wir als hängenden Channel. Solche hängenden Channel können auf einen potenziellen oder tatsächlich auftretenden Channel hindeuten. Es sei allerdings dazu gesagt, dass eine solche hängende Operation nicht immer zu einem Deadlock führen muss. Man betrachte dazu das Beispiel in Abb. 7. Da auf dem Channel `x` nie gesendet wird, kommt es es in Zeile 3 zu

```
1 func main() {
2   x := make(chan int, 0)
3   go func() { <- x }()
4 }
```

Abb. 7: Hängender Channel ohne Deadlock

einer hängenden Channel-Operation. Da dabei aber die Main-Routine nicht blockiert wird,

kommt es nicht zu einem Deadlock und die Go-Routine terminiert, sobald die Main-Routine terminiert. Solch eine Routine bezeichnen wir als leakende Routine. Sie führt hierbei nicht zu einem Deadlock, ist aber in der Regel dennoch eine ungewollte Situation. Es ist also sinnvoll, auch solche Situationen zu erkennen.

Wir sind nun also in der Lage solche Situationen zu erkennen. Um solche Situationen verhindern zu können, ist es sinnvoll, die möglichen Kommunikationspartner für diese Operation zu finden um dem Nutzer bei der Suche und Beseitigung solcher Situationen zu helfen. Für Abb. 6 soll also angegeben werden können, dass die Send-Operation in 1 sowohl mit 2 als auch mit 3 synchronisieren kann. Dabei sei allerdings zu beachten, dass nur weil eine Send- und eine Receive-Operation auf dem selben Channel und in unterschiedlichen Routine geschehen, nicht in jedem Fall eine Kommunikation zwischen diesen möglich ist. Man betrachte dazu das Beispiel in Abb. 8. Auf dem Channel `x` wird in 1 gesendet und

```
1 func main() {
2   x := make(chan int)
3   y := make(chan int)
4   go func() { x <- 1; y <- 1 }   // 1
5   go func() { <- y; <- x }       // 2
6   <- x                           // 3
7 }
```

Abb. 8: Beispielprogramm für unmögliche Synchronisation

kann in 2 und 3 empfangen werden. Da es zwei Receive, aber nur eine Send-Operation gibt, kommt es zu einem hängenden Channel. Betrachtet man nur Channel `x` könnte man davon ausgehen, dass 1 nach 2 senden kann, was zu einem Deadlock führen würde. Dies ist aber nicht möglich. Da der Channel `y` in 1 nach `x` sendet, in 2 allerdings von `x` empfangen muss, ist eine Synchronisierung auf `x` von 1 nach 2 nicht möglich und die beiden Operationen bilden demnach keine möglichen Kommunikationspartner.

Um mögliche Kommunikationspartner zu erkennen, nicht mögliche Kommunikationspartner wie in Abb 8 aber auszuschließen, werden Vector-Clocks verwendet. Die grundlegende Idee basiert auf [3].

In einem ersten Durchlauf wird dabei der Trace mit Vector-Clock Informationen nach der Methode von Fidge [6] erweitert. Für jede Routine wird eine Vector-Clock gespeichert, welche für jede Routine einen Wert enthält. Zu Begin werden all diese Werte auf 0 gesetzt. Bei jedem Post-Event, sowohl für Send als auch Receive und für Signal und Wait Elemente wird der Wert der eigenen Routine in der lokalen Vector-Clock um eins erhöht. Bei einem Post-Receive und einem Wait Element wird die Vectorclock vc' betrachtet, welche in der sendenden Routine zum Zeitpunkt des Post-Send- bzw. Signal-Elements vorlag. Da ein Send- bzw. Signal-Event immer vor dem Receive- bzw. Wait-Element erzeugt wird, wurde die entsprechende Vectorclock in jedem Fall bereits bestimmt. Die Zuordnung der Trace-Elemente ist möglich, da der globale Counter bei einem Send an den Empfangenden Channel mitgesendet wird, und in dem entsprechenden Post-Receive- bzw. Wait-Trace-Element gespeichert wird. Bei einem Select-Statement wird nur derjenige Fall betrachtet, der auch tatsächlich ausgeführt wurde. Diese Vector-Clock `vc` wird nun mit der lokalen Vector-Clock vc der empfangenden Routine, bzw. der Wait-Routine `q` verrechnet und ersetzt diese. Dabei gilt

```
if vc[q] <= vc'[q] {
  vc[q] = 1 + vc'[q]
}
for i := 0; i < n; i++ {
  vc[i] = max(vc[i], vc'[i])
}
```

wobei n die Anzahl der Routinen ist.

Für alle andern Elemente, z.B. Pre usw. wird einfach die lokale Vector-Clock der Routine übernommen, ohne diese zu veränden. Da nun die Vector-Clocks zu jedem Zeitpunk bestimmt wurde, kann jedem Send- und Receive-Trace-Element eine Pre- und eine Post-Vector-Clock zugeordnet werden. Dabei handelt es sich um die Vector-Clocks, die bei Erzeugung des Pre- bzw. Post-Events in der Routine, in der die Operation ausgeführ wurde vorlag. Für hängende Operationen, bei denen kein Post-Element existiert, werden alle Werte der Post-Vector-Clock auf max(Int32) gesetzt. Man betrachte das Beispiel in Abb. 9. Man betrachte den Fall, in dem 3 mit 4 und dann 1 mit 5 synchronisiert und 2 eine hängende Operation bildet. In

```
1  func main() {
2    x := make(chan int)
3    go func() {
4      x <- 1                    // 1
5      <- x                      // 2
6    }()
7    go func() { x <- 1 }()      // 3
8    <- x                      // 4
9    <- x                      // 5
10 }
```

Abb. 9: Beispielprogramm für die Betrachtung der Vector-Clocks

diesem Fall erhält man folgenden Trace:

$$[[signal(1,2), signale(2,3), pre(1,x?), post(7,1,x?,6), pre(8,x?), post(13,1,x?,11)]$$
$$[wait(9,2), pre(10,x!), post(11,1,x!), pre(12,1?)]$$
$$[wait(4,3), pre(5,1!), post(6,2,1!),]]$$

Aus diesem Trace lassen sich nun die Vector-Clocks für die einzelnen Operationen berechnen. Diese werden im Folgenden in der Form ${}^{vc}a^{vc'}$ mit der Pre-Vector-Clock vc und der Post-Vector-Clock vc' und der Channel-Operation a. Sie geben sich also als

$$[[{}^{[2,0,0]}x?^{[3,0,2]}, {}^{[3,0,2]}x?^{[4,2,2]}]$$
$$[{}^{[1,1,0]}x!^{[1,2,0]}, {}^{[1,2,0]}x?^{[max,max,max]}]$$
$$[{}^{[2,0,1]}x!^{[2,0,2]}]]$$

Man bezeichne zwei Vector-Clocks vc und vc' als vergleichbar, wenn $\forall i : vc[i] \leq vc'[i]$ oder $\forall i : vc[i] \geq vc'[i]$. Man schreibe in diesem Fall $vc \leq vc$ bzw. $vc \geq vc$ bzw. allgemein $vc \gtreqless vc'$. Andernfalls bezeichnen man vc und vc' als unvergleichbar $vc \gtrless vc'$. Sind zwei Vector-Clocks unvergleichbar, sind sie unabhängig und können somit gleichzeitig auftreten. Man erkennt also zwei Operationen, welche eine mögliche Kommunikation durchführen können daran, dass sie auf dem selben Channel definiert sind, eine Send- und eine Receive-Operation definieren und dass entweder ihre Pre- oder Vector-Clock (oder beide) unvergleichbar sind.

Um alternative Kommunikationspartner für hängende Kanäle zu finden, werden also die Pre- und Post-Vector-Clocks der Channels mit dem selben Channel verglichen.
Man betrachte die hängende Receive-Operation in dem obigen Beispiel. Es gibt in dem Programm 2 Send-Operationen, welche als Kommunikationspartner für die hängende Operation r in frage kommen. Vergleicht man die Vector-Clocks der Send-Operationen mit denen der hängenden Operation wird aber klar, dass nur eine der beiden Operationen tatsächlich möglich ist. Für die Send-Operation in der 2. Routine (der selben wie die hängende Operation) s_1 gilt $s_{1,pre} \leq r_{pre}$ und $s_{1,post} \leq r_{pre}$. Für die andere Send-Operation s_2 in Routine 3 hingegen gilt $s_{1,pre} \not\gtrless r_{pre}$. Sie kann also gleichzeitig mit der hängenden Operation ausgeführt werden und bildet somit einen potenziellen Kommunikation. Hierbei wird auch klar, warum es nicht nur ausreicht einen Trace aufzuzeichnen, wenn eine Operation ausgeführt wird. Da für die Post-Vector-Clocks gilt, dass $s_{2,post} \geq r_{post}$, kann allein aus der Post-Vector-Clock nicht auf eine mögliche Kommunikation geschlossen werden. **(TODO: Besser beschreiben)**

Gebufferte Channel

Anders als in ungebufferten Kanälen müssen in gepufferten Kanälen Send und Receive einer Nachricht nicht gleichzeitig ablaufen. Hierbei kann es zu Situationen kommen, in dem eine Nachricht zwar erfolgreich gesendet, aber nie ausgelesen wird. Man betrachte dazu das Beispiel in Abb. 10. Es besitzt 2 Send- aber nur eine Receive-Operation. Wäre der Channel

```
1 func main() {
2   x := make(chan int, 2)
3   x <- 1          // 1
4   go func() {
5     x <- 1        // 2
6   }
7   <- x
8 }
```

Abb. 10: Beispielprogramm für nicht gelesenen Nachricht in gepuffertem Channel

nicht gepuffert, und hätte 1 mit 3 synchronisiert, würde 2 blockieren und es würde zu einem Deadlock, bzw. dadurch dass 2 nicht in der Main-Routine liegt zu einer hängenden Operation

kommen. Dadurch das der Channel allerdings gepuffert ist können sowohl 1 als auch 2 senden ohne zu blockieren. Da der Trace somit ein gültiges Post-Event dieser Operation besitzt, wird dieses Problem nicht durch die in Abschnitt 2.4.2 beschriebenen Methode erkannt. Solche Situation lassen sich allerdings aus dem Trace einfach erkennen. Dazu wird der mit Vector-Clocks annotierte Trace durchlaufen. Auf diesem wird nun für jeden Channel gezählt, wie oft erfolgreich gesendet bzw. erfolgreich empfangen wurde (Anzahl der entsprechenden Elemente, bei denen die Post-Vector-Clock nicht max(Int) ist). Ist die Anzahl der erfolgreichen Send größer als die Anzahl der erfolgreichen Receive, bedeutet diese, dass nach Abschluss des Programms auf dem entsprechenden Channel noch nicht gelesene Nachrichten vorhanden sind.

In dem Beispiel in Abb. 2.4.2 wird eine weitere Problematik mit gepufferten Channels deutlich. Betrachtet man nur die Vector-Clock Informationen, dann scheint es, als wären 2 und 3 potenzielle Kommunikationspartner. Dies ist allerdings nicht der Fall. In Go sind gepufferte Channels als FIFO Queue implementiert [7]. Das bedeutet, dass bei einem Receive immer diejenige Nachricht ausgegeben wird, welche bereits am längsten in dem Puffer des Channels gehalten wird. Da 1 in dem Beispiel immer vor 2 ausgeführt wird und demnach in 3 immer die Nachricht aus 1 empfangen wird, bilden 2 und 3 kein mögliches Kommunikationspaar und sollte somit auch nicht als solches ausgegeben werden. Um solche Situationen zu erkennen, wird in dem Trace für jedes Post-Event gespeichert, die wievielte Send- bzw. Receive-Operation auf diesem Channel bereits erfolgreich ausgeführt worden sind. Bei der Suche nach möglichen Kommunikationspartnern wird nun für gepufferte Channels überprüft, ob die Anzahl der gesendeten Nachrichten in dem Send-Statement mit der Anzahl der empfangenen Nachrichten in dem Receive-Statement übereinstimmt. Da die Send- und Receive-Operation nicht gleichzeitig ausgeführt werden müssen, ist es nicht notwendig die Vector-Clocks zu betrachten.

2.4.3 Select

In Go können Select-Statements dazu verwendet werden, abhängig davon, auf welchen Channels Nachrichten gesendet werden unterschiedliche Programmteile auszuführen. Dies erschwert die Analyse des Programs, da der tatsächliche Programmablauf dabei praktisch nicht-deterministisch werden kann [8]. Da der aufgezeichnete Trace immer nur einen Programmablauf widerspiegelt, führt dies zu Problemen, da nicht betrachtete Ausführungspfade zu Deadlocks oder anderen Problemen führen können. Eine Möglichkeit besteht darin, das Programm mehrfach auszuführen, und dabei immer unterschiedliche Select-Cases zu erzwingen. Dies ist der Ansatz, welcher für den GFuzz-Detektor [9] gewählt wurde. Dazu werden die Select-Statements in dem Programmcode so verändert, dass aus den vorhandenen Cases eines gezielt ausgewählt werden kann. Durch die Switch-Operation lässt sich einer der Fälle in der ursprünglichen Select-Operation auswählen, welche bevorzugt ausgeführt werden soll. Nur wenn dieser innerhalb einer voreingestellten Zeit nicht ausgeführt wird, wird die ursprüngliche Select-Operation vollständig ausgeführt um zu verhindern, dass es durch diese Instrumentierung zu einem Deadlock kommt, welche ohne sie nicht aufgetreten wäre. Der Ablauf eines Programms bezüglich seiner Select-Statements kann nun als Liste von Tupeln $[(s_0, c_0, e_0), \ldots, (s_n, c_n, e_n)]$ dargestellt werden. Dabei bezeichnet $s_i (0 \leq i \leq n)$ die ID einer Select-Operation, c_i die Anzahl der Cases in dieser Operation und e_n den Index des Ausgeführten Case. Das Programm wird nun mehrfach durchlaufen, wobei die Ordnung zufällig verändert wird. Dazu wird nach jedem Durchlauf der Index e_i jedes Tupels auf einen zufälligen, aber gültigen Wert gesetzt. Da die Anzahl der möglichen Ausführungspfade durch die Select-Statements gegebenenfalls gegen unendlich gehen kann, ist es nicht möglich jede mögliche Kombination von Select-Cases auszuführen. Aus diesem Grund sammelt GFuzz während der Ausführung einer Ordnung Informationen über diese, um die Qualität einer

Ordnung abzuschätzen. Aus diesen wird über

$$score = \sum_{j} \log_2 CountChOpPair_j + 10 \cdot \#CreateCh \\ + 10 \cdot \#CloseCh + 10 \cdot \sum_{j} MaxChBufFull_j \tag{1}$$

eine Wertung für die durchlaufenden Ordnung bestimmt, über welche die Anzahl der Mutationen bestimmt werden. Hierbei bezeichnet $CountChOpPair_j$ die Anzahl der neuen Ausführungen von Channel-Operationen pro Channel j, $CreateCh$ die Anzahl der neu erzeugten Channels, $CloseCh$ die Anzahl der neu geschlossenen Channels und $MaxChBufFull_j$ die neue maximale Anzahl an gepufferten Nachrichten in jedem Channel j. Neu bedeutet hierbei, dass die entsprechende Operation noch nicht, auch nicht in vorherigen Durchläufen, aufgetreten ist. Die Anzahl der Mutationen einer Ordnung gibt sich nun als $\#nuwMut = \lceil 5 \cdot {}^{score}/_{maxScode} \rceil$, wobei $score$ den Score der Ordnung und $maxScore$ den bisher maximal erhaltenen Score bezeichnet. Diese $\#newMut$ neuen Mutationen werden nun basierend auf der alten erzeugt, in eine Queue eingefügt und anschließend durchlaufen. GFuzz beschränkt sich bei der Erkennung von Bugs auf die Erkennung von blockenden Bugs.

Die Betrachtung von verschiedenen Pfaden soll nun auch für GoChan **(TODO: gegebenenfall umbenennen)** verwendet werden. Für jeden Durchlauf wird der Trace aufgezeichnet und dieser basierend auf 2.4 analysiert. Die Auswahl der Pfade wird dabei vereinfacht. Anders als in GFuzz basieren Ausführungsordnungen nicht auf schon versuchten Ordnungen, sondern es wird eine Menge von vollständig zufälligen Ordnungen betrachtet. Die Anzahl der Durchläufe basiert dabei auf der Anzahl der möglichen Pfade. Dies wird später (Kap. 3.2) noch genauer erläutert.

(TODO: Close nochmal schauen)

3 Instrumentierung

3.1 Trace

Wie bereits in dem vorherigen Kapitel beschrieben soll, um das Programm analysieren zu können ein Trace aufgezeichnet werden.
Anders als in vielen anderen Programmen, welche den Trace von Go-Programmen analysieren, wie z.B. [10] oder [11] wird dabei der Tracer selbst implementiert und basiert nicht auf dem Go-Runtime-Tracer [12]. Dies ermöglicht es, den Tracer genau auf die benötigten Informationen zuzuschneiden und so einen geringeren negativen Einfluss auf die Laufzeit des Programms zu erreichen.

Um diesen Trace zu erzeugen, werden die Standartoperation auf Go durch Elemente des Tracers ersetzt. Die Funktionsweisen dieser Ersetzungen sind im folgenden angegeben. Dabei werden nur solche Ersetzungen angegeben, welche direkt für die Erzeugung des Traces notwendig sind. Zusätzlich werden noch weitere Ersetzungen durchgeführt, wie z.B. die Ersetzung der Erzeugung von Mutexen und Channel von den Standardvarianten zu den Varianten des Tracers. Hierbei wird auch die Größe jedes Channels gespeichert. Dies werden in der Übersicht zur Vereinfachung nicht betrachte. Auch werden in der Übersicht nur die Elemente betrachtet, die für die Durchführung der Operation und dem Aufbau des Traces benötigt werden. Hilfselemente, wie z.B. Mutexe, welche verhindern, dass mehrere Routinen gleichzeitig auf die selbe Datenstruktur, z.B. die Liste der Listen, welche die Traces für die einzelnen Routinen speichern, zugreifen, werden nicht mit angegeben. Dabei sei c ein Zähler,

nR ein Zähler für die Anzahl der Routinen, nM ein Zähler für die Anzahl der Mutexe und nC ein Zähler für die Anzahl der Channels. nM und nC werden bei der Erzeugung eines neuen Mutex bzw. eines neuen Channels atomarisch Incrementiert. Den erzeugten Elementen wird er neue Wert als id zugeordnet. All diese Zähler seien global und zu Beginn als 0 initialisiert. Außerdem bezeichnet mu einen Mutex, rmu einen RW-Mutex, ch einen Channel und B bzw. B_i mit $i \in \mathbb{N}$ den Körper einer Operation. Zusätzlich sei id die Id der Routine, in der eine Operation ausgeführt wird, $[signal(t, i)]^{id}$ bedeute, dass der das entsprechende Element (hier als Beispiel $signal(t, i)$), in den Trace der Routine mit id id eingeführt wird und $[+]^i$ bedeute, das in die Liste der Traces ein neuer, leerer Trace eingefügt wird, welcher für die Speicherung des Traces der Routine i verwendet wird. $\langle a|b\rangle$ bedeutet, dass ein Wert je nach Situation auf a oder b gesetzt wird. Welcher Wert dabei verwendet wird, ist aus der obigen Beschreibung der Trace-Elemente erkennbar. e_1 bis e_n bezeichnet die Selektoren in einem Select statement. e_i^* bezeichnet dabei einen Identifier für einen Selektor, der sowohl die Id des beteiligten Channels beinhaltet, als auch die Information, ob es sich um ein Send oder Receive handelt und e_i^m die Message, die in einem Case empfangen wurde.

go B	$\Rightarrow$	nr := atomicInc(nR); ts := atomicInc(c); [signal(ts, nr)]^{nr};
		[+]^{nr}; go { ts' := atomicInc(c); [wait(ts, nr)]^{id}; B};
ch <- i	$\Rightarrow$	ts := atomicInc(c); [pre(ts, ch.id, true)]^{id}; ch <- {i, id, ts};
		ts' := atomicInc(c); [post(ts', ch.id, true, id)]^{id}
<- ch	$\Rightarrow$	ts := atomicInc(c); [pre(ts, ch.id, false)]^{id};
		{i, id_send, ts_send} := <-c; ts' := atomicInc(c);
		[post(ts', ch.id, false, id_send, ts_send)]^{id}; return i;
select($e_i \rightsquigarrow B_i$)	$\Rightarrow$	ts := atomicInc(c); [pre(ts, e_1^*, ..., e_n^*, false)]^{id};
		select($e_i \rightsquigarrow$ { ts' := atomicInc(c);
		[⟨post(ts, e_i.ch, false, e_i^m.id_send, e_i^m.ts_send) |
		post(ts, e_i.ch, true, id) ⟩]^{id} B_i})
select($e_i \rightsquigarrow B_i$ | B_{def})	$\Rightarrow$	ts := atomicInc(c); [pre(ts, e_1^*, ..., e_n^*, false)]^{id};
		select($e_i \rightsquigarrow$ { ts' := atomicInc(c);
		[⟨post(ts, e_i.ch, false, e_i^m.id_send, e_i^m.ts_send) |
		post(ts, e_i.ch, true, id) ⟩]^{id} B_i} |
		ts' := atomicInc(c); [default(ts)]^{id}; B_{def})
mu.(Try)Lock()	$\Rightarrow$	ts := atomicInc(c); mu.(Try)Lock();
		[lock(ts, mu.id, ⟨-|t⟩, ⟨0|1⟩)]^{id};
mu.Unlock()	$\Rightarrow$	ts := atomicInc(c); mu.Unlock(); [unlock(ts, mu.id)]^{id};
rmu.(Try)(R)Lock()	$\Rightarrow$	ts := atomicInc(c); rmu.(Try)(R)Lock();
		[lock(ts, rmu.id, ⟨-|t|r|tr⟩, ⟨0|1⟩)]^{id};
rmu.Unlock()	$\Rightarrow$	ts := atomicInc(c); rmu.Unlock(); [unlock(ts, rmu.id)]^{id};

(TODO: Mehr)

3.2 Select

Neben den Ersetzungen der einzelnen Operationen, werden auch die Select-Statements verändert, um einen der Fälle bevorzugt auswählen zu können, wie in Kap. 2.4.3. beschrieben. Die Implementierung basiert dabei zum größten Teil auf [9]. Abb. 11 zeigt ein Beispiel

für die Instrumentierung eines Die Select-Operation wird durch eine Switch-Operation

```
select {
  case <- Fire(time.Second):
    Log("Timeout!")
  case e := <- ch:
    Log("Unexpected!")
  case e := <- errCh:
    Log("Error!")
}
```

```
switch FetchOrder(...) {
  case 0:
    select {
      case <- Fire(time.Second):
        Log("Timeout!")
      case <- time.After(T):
        .....
    }
  case 1:
    select {
      case e := <- ch:
        Log("Unexpected")
      case <- time.After(T):
        .....
    }
  case 2:
    select {
      case e := <- errCh:
        Log("Error")
      case <- time.After(T):
        .....
    }
  default:
    .....
}
```

Abb. 11: Beispiel für die Order-Enforcement-Instrumentierung eines Select-Statements durch GFuzz for (links) und nach der Implementierung (rechts). Ersetze ..... in dem Programm nach der Instrumentierung (rechts) durch das Programm vor der Instrumentierung (links). [9, gekürzt]

auf der `Fetch-Order` ersetzt. Die Fetch-Order speichert für jede Select-Operation den Index des bevorzugten Case. Für eine gewisse, festgelegte Zeit T wird somit nur auf die in der `Fetch-Order` spezifizierten Operation gewartet. Wird diese in der vorgegebenen Zeit nicht ausgeführt, geht das Programm wieder in die Ausführung der ursprünglichen Select-Operation um zu verhindern, dass es zu einem Deadlock kommt, welcher in dem originalen-Code nicht vorgekommen wäre. Das selbe gilt auch, wenn für die Select-Operation fälschlicherweise kein gültiges Case ausgewählt worden ist. Anders als in [9] beschrieben, wird für ein Select in dem selben Durchlauf immer der selbe Channel priorisiert, auch wenn die Operation mehrfach durchlaufen wird. Um für jede Ausführung einen eigenen Case zu priorisiere, müsste, da die Ordnung bereits vor dem Durchlauf festgelegt werden soll, der

Ablauf des Programms bereits bekannt sein. Dies ist allerdings nur möglich, wenn die Wahl der Cases der Select-Operationen keine Einfluss auf die Ausführung anderer Select-Cases hat. Dass dadurch nicht alle möglichen Abläufe durchlaufen werden können, muss dabei in kauf genommen werden.

3.3 Automatisierter Instrumenter

(DRAFT: Instrumenter) Um den Trace zu erzeugen, müssen verschiedene Operationen durch Funktionen des Tracers ersetzt bzw. erweitert werden. Bei einem größeren Programmcode ist eine händische Instrumentierung nicht machbar. Da sich der Tracer auch negativ auf die Laufzeit des Programms auswirken kann, ist es in vielen Situationen nicht erwünscht, ihn in den eigentlichen Release-Code einzubauen, sondern eher in eine eigenständige Implementierung, welche nur für den Tracer verwendet werden. Um dies zu automatisieren wurde ein zusätzliches Programm implementiert, welches in der Lage ist, den Tracer in normalen Go-Code einzufügen.

Die Implementierung arbeitet aus dem Abstract Syntax Tree des Programms. Bei dem Durchlaufen dieses Baums werden die entsprechenden Mutex-, Channel- und Select-Operationen in dem Programm erkannt, und durch ihre entsprechenden Tracer-Äquivalente ersetzt. Neben dem Ersetzen der verschiedenen Operationen werden außerdem einige Funktionen hinzugefügt. Zu Begin der Main-Funktion des Programms wird der Tracer initialisiert. Zusätzlich wird eine zusätzliche Go-Routine gestartet, in welcher ein Timer läuft. Ist dieser abgelaufen, wird die Analyse gestartet, auch wenn das Programm noch nicht vollständig durchlaufen ist. Dies führt dazu, dass auch Programme, in welchen ein Deadlock aufgetreten ist, analysiert werden können. Endet das Programm in der vorgegebenen Zeit, wird der Analyzer nach der Beendigung des Programms gestartet.

(EXTEND: Instrumenter)

3.3.1 Laufzeit

Instrumenter Zuerst soll die Laufzeit des Instrumenters betrachtet werden. Es ist erwartbar, dass sich die Laufzeit linear in der Anzahl der Ersetzungen in dem AST, also der Anzahl der Mutex- und Channel-Operationen verhält. Dies bestätigt sich auch durch die Messung der Laufzeit des Programms (vgl. Abb. 12)

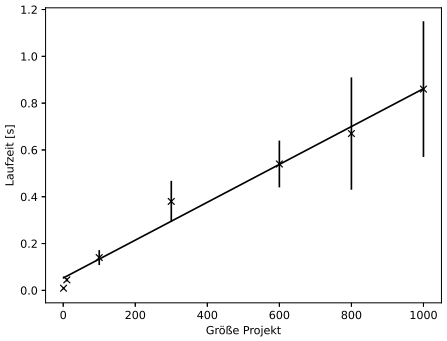

Abb. 12: Laufzeit des Instrumenters

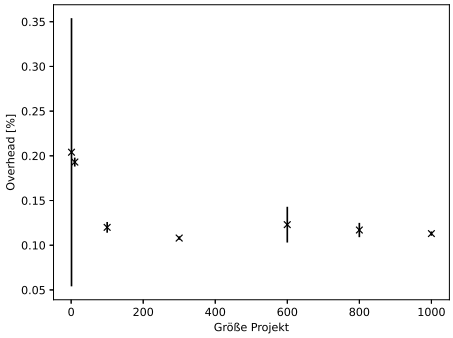

Abb. 13: Prozentualer Overhead des Tracers ohne Analyse

Der abgebildete Graph zeigt die Laufzeit des Programms in s abhängig von der Größe des Programms. Das Programm besteht dabei aus einem Testprogramm, welches alle möglichen Situationen mit Channels und Mutexen abbildet. Die Vergrößerung des Programmes wurde dadurch erreicht, dass die Datei mit dem Programmcode mehrfach in dem Projekt vorkam. Ein Projekt mit Größe n besteht vor der Instrumentierung also aus n Dateien, mit insgesamt $65n$ Zeilen von Code und $52n$ Ersetzungen in dem AST. Die tatsächliche Laufzeit des Instrumenters auf einen Programm hängt schlussendlich natürlich von der tatsächlichen Größe des Projekt und der Verteilung der Mutex- und Channel-Operationen in dem Code ab.

Zusätzlich wurde die Messung auch mit drei tatsächlichen Programmen durchgeführt. Die

Projekt	LOC	Nr. Dateien	Nr. Ersetzungen	Zeit [s]
ht-cat	733	7	233	0.013 ± 0.006
go-dsp	2229	18	600	0.029 ± 0.009
goker	9783	103	4928	0.09 ± 0.03

Tab. 1: Laufzeit des Instrumenters für ausgewählte Programme

dort gemessenen Werte befinden sich in Tabelle 1. Gerade in Abhängigkeit von der Anzahl der Ersetzungen, stimmen die hier gemessenen Werte mit denen in Abb. 12 gut überein, während es bei den anderen Parametern größere Abweichungen gibt. Dies bestätigt dass der dominante Faktor für die Laufzeit des Programms die Anzahl der Ersetzungen in dem AST ist, und die Laufzeit linear von dieser abhängt.

Tracer Folgend soll nun auch die Laufzeit des Tracers betrachtet werden. Hierbei wird nur die Laufzeit des eigentlichen Tracers, nicht aber der anschließenden Analyse betrachtet. Um den Overhead in Abhängigkeit von der Größe des Projektes messen zu können, wird das selbe Testprogramm betrachtet, welches bereits in der Messung für den Instrumenter verwendet wurde. Abb. 13 zeigt den gemessenen Overhead. Der durchschnittliche Overhead über alle gemessenen Werte liegt dabei bei 14 ± 2 %. Da der Overhead aber linear davon abhängt, wie groß der Anteil der Mutex- und Channel-Operationen im Verhältniss zu der Größe bzw. der Laufzeit des gesammten Programms ist, kann dieser Wert abhängig von dem tatsächlichen Programm start schwanken. Dies wird unteranderem klar, wenn man den Overhead für ht-cat (9 ± 3 %) und go-dsp (60 ± 18 %) welche 51 ± 21 Prozentpunkte außeinander liegen. **(EXTEND: Laufzeit)**

4 Implementierung

Das folgende Kapitel soll sich nun mit der Analyze des in Kap **??** erstellten Trace befassen. Sie befasst sich dabei mit der Erkennung unerwünschter Situationen durch die vorkommenden Mutexe und Routinen. **(EXTEND: Mehr zur Einführung)**

4.1 Mutex

(TODO: Für mutex: regeln, gate-locks)

Eine genauere Erklärung der Implemenierung des Locks findet sich in [13]. **(TODO: ist des erlabut, oder soll ich es nochmal komplett beschreiben)**

Die Lock-Bäume werden basierend auf dem aufgezeichneten Trace aufgebaut. Dazu werden die Traces der einzelnen Routinen nacheinander durchlaufen. Für jede Routine erzeugen wir eine Liste `currentLocks` aller Locks, die momentan von der Routine gehalten werden. Die einzelnen Elemente des Trace einer Routine werden nun durchlaufen. Handelt es sich dabei um ein Lock Event eines Locks `x`, wird für jedes Lock `l` in `currentLocks` eine Kante von `l` nach `x` in den Lock-Graphen eingefügt. Anschließend wird `x` in `currentLocks` eingefügt. Ist das handelt es sich bei dem Element um ein unlock Event auf dem Lock `x`, dann wird das letzt vorkommen von `x` auf `currentLocks` entfernt.
Nachdem der Trace einer Routine durchlaufen wurde, wird überprüft ob sich noch Elemente in `currentLocks` befinden. Ist dies der Fall, handelt es sich um Locks, welche zum Zeitpunkt

der Terminierung des Programms noch nicht wieder freigegeben worden sind. Dies deutet darauf hin, dass die entsprechende Routine nicht beendet wurde, z.B. weil das Programm bzw. die Main-Routine beendet wurden. Dies kann einfach durch die entsprechende Logik des Programms zustande gekommen sein, es kann aber auch auf einen tatsächlich auftretenden Deadlock, z.B. durch doppeltes Locking des selben Locks in einer Routine, ohne dass es zwischenzeitlich wieder freigegeben wurde. In diesem Fall wird eine warnung ausgegeben.
Ein potenzieller Deadlock gibt sich nun, wenn in diesem Graph ein Kreis existert. Dabei muss darauf geachtet werden, dass nicht alle Kanten durch die selbe Routine erzeugt wurden, und dass in zwei, in dem Kreis hintereinander folgende Kanten der gemeinsame Knoten nicht beides mal durch eine R-Lock Operation durch Kanten verbunden wurde. Die Erkennung solcher Zyklen geschieht nun durch eine Tiefensuche auf dem erzeugen Baum. Wird ein solcher Zyklus erkannt, wird ebenfalls eine Warnung ausgegeben.

4.2 Channels

(TODO: cont, implementierung beschreiben)

5 Auswertung

6 Zusammenfassung

(TODO: Zusammenfassung schreiben)

7 Acknowledgments

(TODO: Acknowledgments schreiben)

Abbildungsverzeichnis

Tabellenverzeichnis

ToDo Counters

(TODO: Remove ToDo Counter List)

To Dos: 14; 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14

Parts to extend: 4; 1, 2, 3, 4

Draft parts: 1; 1

Literaturverzeichnis

[1] J. Zhou, S. Silvestro, H. Liu, Y. Cai, and 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), pp. 729–740, IEEE Computer Society, 11 2017.

[2] The Go Team, "Effective go - channels," 2022. `https://go.dev/doc/effective_go#channels`.

[3] M. Sulzmann and K. Stadtmüller, "Two-phase dynamic analysis of message-passing go programs based on vector clocks," *CoRR*, vol. abs/1807.03585, 2018.

[4] P. Joshi, C.-S. Park, K. Sen, and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," *SIGPLAN Not.*, vol. 44, p. 110–120, jun 2009.

[5] R. Agarwal, L. Wang, and S. D. Stoller, "Detecting potential deadlocks with static analysis and run-time monitoring," in *Hardware and Software, Verification and Testing* (S. Ur, E. Bin, and Y. Wolfsthal, eds.), (Berlin, Heidelberg), pp. 191–207, Springer Berlin Heidelberg, 2006.

[6] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," 1988.

[7] The Go Authors, "Go-standart library: src/runtime/chan.go," 2014. `https://go.dev/src/runtime/chan.go`.

[8] The Go Team, "The go programming language specification - select," 2022. https://go.dev/ref/spec#Select_statements.

[9] Z. Liu, S. Xia, Y. Liang, L. Song, and 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*, ASPLOS '22, (New York, NY, USA), p. 888–902, Association for Computing Machinery, 2022.

[10] S. Taheri and G. Gopalakrishnan, "Goat: Automated concurrency analysis and debugging tool for go," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 138–150, 2021.

[11] I. Danyliuk, "Visualizing concurrency in go," 2016. https://divan.dev/posts/go_concurrency_visualize/.

[12] The Go Team, "Go documentation: trace," 2022. https://pkg.go.dev/cmd/trace.

[13] E. Kassubek, "Dynamic deadlock detection in go." https://github.com/ErikKassubek/BachelorProjektGoDeadlockDetection/blob/main/Bericht/Bericht.pdf, 22 08.