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 **1**

2. Theorie **3**

2.1. Routinen 3

2.2. Mutexe 3

2.3. Channel 5

2.4. Probleme 7

2.4.1. Mutex 7

2.4.2. Channel 8

2.5. Detektor 10

2.5.1. Trace 11

2.5.2. Analyze 13

2.5.3. Select 21

3. Instrumentierung **23**

3.1. Trace 23

3.2. Select 26

3.3. Automatisierter Instrumenter 27

3.3.1. Instrumentierung der Dateien 28

3.3.2. Neue Main-Datei 32

3.3.3. Laufzeit 33

4. Analyzer **36**

4.1. Aufzeichung des Trace 36

4.2. Mutex 37

4.3. Channels 39

4.4. Output 43

5. Auswertung **44**

5.1. Standardprogramme 44

5.2. GoKer 46

6. Zusammenfassung **48**

A. Beschreibung der betrachteten Programme **49**

Literaturverzeichnis **61**

1. Einführung

(DRAFT: Einführung schreiben) Go ist eine seit 2007 von Google entwickelte Programmiersprache, welche sich in den letzten Jahren zu einer der beliebtesten Programmiersprachen entwickelt hat [1]. Ein Fokus der Sprache liegt dabei auf der Erstellung sicherer und effizienter nebenläufiger Programme. Go stellt verschiedene Funktionen für die Erstellung solcher nebenläufiger Programme zur Verfügung. Die wichtigsten sind dieser Funktionen sind dabei Go-Routinen, Mutexe und Channels.
Go-Routinen erlauben das nebenläufige Ausführen mehrerer Codeteile. Mutexe erlauben die Synchronisation von Routinen, sowie die Lösung des Problems des kritischen Ausschlusses. Channel können ebenfalls für die Synchronisation verwendet werden, können aber zusätzlich für eine Kommunikation, bzw. das Senden von Daten zwischen Routinen verwendet werden. Programmierer werden dabei angehalten, Channel anstelle von gemeinsamen Variablen zu verwenden, da bei diesen die Wahrscheinlichkeit für das Auftreten von Concurrency-Bugs geringer ist [2].
Solche Bugs (z.B. Deadlocks), welche durch die Nebenläufigkeit von Programmen sowie ihrer Synchronisationmechanismen entstehen sind in der Praxis aber dennoch sehr weit verbreitet [3].

Diese Arbeit beschäftigt sich mit der Entwicklung und Implementierung eines dynamischen Detektors “GoChan” für die automatische Erkennung von Problemen, welche durch die Verwendung von Mutexen und Channels in nebenläufigen Go-Programmen entstehen. Der größte Fokus wird dabei auf die Erkennung und Analyse von Deadlock-Situationen, sowie

das vorzeitige schließen von Channels gelegt, da diese zu einem Absturz eines Programms führen können. Der Detektor vereinigt dabei verschiedene Methoden aus iGoodLock [4] und UNDEAD [5] sowie Gopherlyzer-GoScout [6] und GFuzz [7] zur Erkennung und Analyse von durch Mutenxen und Channels erzeugten Bugs.
GoChan basiert dabei aus zwei Programmen, einem Insturmentierer und einem Analyzer, für welche jeweils eine Implementierung vorliegt. Für die Analyze wird das zu analysierende Programm mehrfach durchlaufen. Bei jedem Durchlauf zeichnet der Analyzer relevante Situationen auf und führt anschließend, basieren auf dem so erstellten Trace eine Analyse aus, um Probleme zu erkennen. Die Analyse wird mehrfach wiederholt, um verschiedene Ausführungspfade zu analysiere, welche durch die Verwendung verschiedener Pfade in Select-Operationen erzeugt werden. Um den Trace eines Programmdurchlaufs aufzeichnen zu können und bei Select-Operationen verschiedene Ausführungspfade zu erzwingen, muss der zu analysierende Programmcode verändert werden. Dies kann durch den Insturmentierer automatisiert erfolgen.

Der Aufbau dieser Arbeit ist wie folgt. Zuerst wird in Abschnitt 2 die theoretische Funktionsweise des Detektors beschrieben und dargelegt, welche Art von Situationen der Detektor erkennen soll. Abschnitt 3 geht auf die Funktionsweise und Implementierung des Instrumenters und Abschnitt 4 auf die Implementierung des Analyzers ein. In Abschnitt 5 wird der Detektor auf konstruierte und tatsächliche Programme angewandt, um zu überprüfen, wie gut er in der Lage ist, Probleme richtig einzusätzen. Für die tatsächlichen Programme werden dabei Programme aus GoBench [8] verwendet, welche eine Sammlung von Programmen mit Concurrency-Bugs aus mehreren großen open-source Programmen besitzt. Zuletzt werden die Erkenntnisse in Abschnitt 6 zusammengefasst. Die Implementierung der beschriebenen Programme befindet sich in `https://github.com/ErikKassubek/GoChan`.

2. Theorie

Im folgenden soll der theoretische Hintergrund bezüglich der Betrachteten Probleme sowie die theoretische Funktionsweise des Detektors betrachtet werden.

2.1. Routinen

Bei Go-Routinen handelt es sich um leichtgewichtige Threads, welche nebenläufig mit anderen Routinen auf dem selben Adressraum laufen [9]. Anders als Threads in vielen anderen Programmiersprachen werden sie nicht durch den OS-Kernel, sondern durch die Go-Runtime selber gesteuert. Diese besitzt einen eigenen Scheduler, welcher eine Technik namens m:n-Scheduling verwendet, um durch Multiplexing m Go-Routinen auf n OS-Threads auszuführen. Diese Herangehensweise hat den Vorteil, dass dadurch eine bessere Ausführungsgeschwindigkeit erreicht und ein geringerer Speicher benötigt wird. Dadurch ist es möglich, dass tausende oder sogar hunderttausende Routinen gleichzeitig auf einer Routine laufen. Sie erleichtert außerdem die Implementierung von Synchronisations- und Kommunikationsmechanismen.

2.2. Mutexe

Mutexe (auch Locks genannt) gehören zu den am weitesten verbreitenten Mechanismen zur Synchronisierung nebenläufiger Programme [5]. Sie bilden eine Lösung für das Problem des

kritischen Abschnitts. In solch einem kritischen Abschnitt darf sich immer nur maximal eine Routine gleichzeitig aufhalten. Solche Abschnitte werden nun mit einem Mutex umschlossen. Sie werden unter anderem verwendet um zu verhindern, dass mehrere Routinen gleichzeitig auf die selbe globale Datenstruktur zugreifen. Mutexe besitzen zwei Zustände, geöffnet und geschlossen, mit denen ihre Verhalten gesteuert wird. Sie besitzen dabei die folgenden Operationen:

- Lock: Die Lock-Operation wird vor dem Eintritt in einen kritischen Bereich aufgerufen. Sie versucht den Mutex zu schließen. Ist es momentan geöffnet, wird der Mutex geschlossen und der kritische Bereich ausgeführt. Ist der Mutex bereits geschlossen, dann muss die Routine, in welcher die Lock-Operation ausgeführt werden soll so lange warten, bis der Mutex wieder geöffnet wird und geschlossen werden kann.

- TryLock: Eine TryLock Operation versucht, wie die Lock-Operation, einen Mutex zu schließen. Anders als bei Lock wird die Routine allerdings nicht blockiert, wenn eine Schließung nicht direkt möglich ist. Es wird lediglich zurückgegeben, ob sie erfolgreich war oder nicht. Ein Programmierer kann in diesem Fall selber entscheiden, wie das Programm weiter ablaufen soll, ob also z.B. der kritische Bereich übersprungen werden soll.

- Unlock: Diese Operation öffnet einen geschlossenen Mutex. Der versuch ein geöffnetes Schloss zu öffnen führt zu einem Laufzeitfehler. Theoretisch verhalten sich Mutexe in Go wie binäre Semaphore, d.h. es ist möglich, dass eine Routine ein geschlossenes Lock freigibt, obwohl das Lock von einer anderen Routine geschlossen wurde. Da dies allerdings in der Praxis sehr leicht zu einem unvorhersehbaren Verhalten führen kann, ist es fasst immer üblich, dass ein Lock immer nur von derjenigen Routine freigegeben werden soll, von der es beansprucht wurde. Im weiteren wird daher angenommen, dass ein Mutex immer von der Routine geöffnet wird, von welchem es geschlossen wurde.

Mutexe können mit (Try)RLock Operationen zu RW-Mutex erweitert werden. Dabei kann der selbe Mutex von mehreren (Try)RLock-Operationen geschlossen werden, ohne dass

es zwischenzeitlich geöffnet werden muss. Es ist allerdings nicht möglich, dass ein Mutex gleichzeitig über eine RLock- und eine Lock-Operation geschlossen wird. RW-Mutexe 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.

2.3. Channel

Channel [9] 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 ein Receive-Statement auf dem Channel `c` ausführt. 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 müssen. Eine Routine 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.
Basierend auf dem Go-Memory-Model gilt, dass zum einen das Senden auf einem Channel vor dem dazugehörigen Receive auf dem Channel synchronisiert wird und dass das k-te Receive auf einem Channel mit Kapazität C vor der Beendigung des $k + C$-ten Send auf dem Channel synchronisiert wird [10]. In der Praxis verhält sich der Buffer eines Channels allerdings wie eine FIFO-Queue, das heißt, es wird bei einem Receive immer die ältest in dem Buffer vorhandene Nachricht ausgegeben. Im Folgenden wird daher immer von diesem Verhalten ausgegangen.

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 für Select

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.

Channel können vorzeitig durch eine `close` Operation geschlossen werden. In diesem

Fall kann auf diesem Channel nicht mehr gesendet werden. Versucht das Program auf einem geschlossenen Channel zu senden kommt es zu einem Laufzeitfehler und das Programm wird abgebrochen. Versucht das Programm auf einem geschlossenen Channel zu lesen, wir ein Default-Wert zurückgegeben, ohne dass die Routine blockiert.

2.4. Probleme

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. 2 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. 2.: 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.

Neben zyklischem Locking kann es auch durch doppeltes Locking von Mutexen zu Deadlocks kommen. Ein Beispiel dazu findet sich in Abb. 3. Der Mutex `x` soll hierbei mehrfach

```
1 func main() {
2   var x sync.Mutex
3
4   x.Lock()
5   x.Lock()
6 }
```

Abb. 3.: Beispielprogramm doppeltes Locking

geschlossen werden, ohne dass es zwischenzeitlich geöffnet wird. In diesem Fall blockiert die Routine endlos.

In Go terminiert ein Programm, wenn dessen Main-Routine terminiert, unabhängig davon, ob andere Routinen noch laufen. Dies bedeutet, dass es nur zu einem "echten" Deadlock, bei welchem das gesamte Programm blockiert, kommen kann, wenn die Main-Routine in der Deadlock-Situation involviert ist. Sind nur Nicht-Main-Routinen an dem Deadlock beteiligt, kann dass Programm dennoch terminieren. Man bezeichne Nicht-Main-Routinen, welche sich in einer blockenden Deadlock Situation befinden, als hängende Routinen. Auch wenn diese nicht zu einer nicht-terminierung des Programms führen können, handelt es sich bei ihnen dennoch um ungewolltes Verhalten.

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. Man betrachte das

Programm in Abb. 4. Es gibt in diesem zwei mögliche Ausführungspfade. Man betrachtet

```
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. 4.: Beispielprogramm mit hängendem Channel

zuerst den Fall, in dem 1 mit 3 synchronisiert. Da eine go-Routine automatisch abgebrochen wird, wenn die Main-Routine terminiert, entsteht hierbei kein “echter"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, 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.

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. 5. 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. 5.: 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. Hierbei kann es nur zu einem blockierenden Deadlock

kommen, wenn aus einem leeren Channel empfangen wird, ohne dass es irgendwann zu einer sendenden Operation kommt, oder wenn auf einem vollen Channel gesendet werden soll, ohne dass es zu einer Empfangenden Operation kommt.
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 Beseitigung der Probleme erleichtern.

Ein weiterer möglicher kritischer Fehler bei Channels liegt in dem Senden auf einem geschlossenen Channel. Da dies zu einem Abbruch des Programms führen kann. Dies sollte also unter allen Umständen vermieden werden. Das Empfangen auf einem geschlossen Channel ist hingegeben kein Problem.

2.5. Detektor

Der Detektor basiert auf zwei Schritten. Zuerst wird das Programm durchlaufen. Dabei wird ein Trace aufgezeichnet, welche die relevanten Informationen über den Programmablauf aufzeichnet. Dieser Trace wird im Anschluss analysiert, um Informationen über Probleme in dem Programm zu erhalten. Das ganze wird gegebenenfalls mehrfach wiederholt, um eine breitere Abdeckung des Programmcodes zu erreichen.

2.5.1. Trace

Um ein Program analysieren zu können, soll der Ablauf eines Programmdurchlaufs (Trace) aufgezeichnet werden. Der grundlegende Aufbau des Trace basiert auf [6]. Er wird aber um Informationen über Locks erweitert. Der Trace wird für jede Routine separat aufgezeichnet. Außerdem wird, anders als in [6] 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:

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

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!, n)`: 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. n zählt die wievielte erfolgreiche Send-Operation die zugrundeliegende Operation im Programmablauf war.

- `post(t, i, x?, t', n)`: 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. n zählt die wievielte erfolgreiche Receive-Operation die zugrundeliegende Operation im Programmablauf war.

- `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.

- `close(t, x)`: Mit diesem Eintrag wird das schließen eines Channels x in dem Trace aufgezeichnet.

- `lock(t, y, b, c)`: Der Beanspruchungsversuch eines Locks mit id y wurde gestartet. 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. Das Trace-Element wird vor der abgeschlossen Beanspruchung in den Trace eingefügt um sicher zu stellen, dass ein zyklisches Locking auch dann erkannt wird, wenn es zu einem tatsächlichen Deadlock führt.

```
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. 6.: Beispielprogramm für Tracer

- `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 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.

2.5.2. Analyze

Nach dem das Programm durchlaufen und der Trace aufgezeichnet wurde, wird dieser anschließend Analysiert. Dabei werden für die Erkennung von Problemen mit Mutexen und

Channels zwei verschiedene Ansätze verwendet.

Mutexe

Man betrachte zuerst die Erkennung von eines zyklischen Locking. Da solche Situationen nur in ganz besonderen Situation auftreten (in Abb. 2 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 [5]. 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 geschlossen worden ist, währen das Lock m_2 schließt [11]. 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. 7 In Abb. 8 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 werden können, wird in dem Kapitel zur Implementierung des Detektors (4.2) noch genauer betrachtet.

Channel

Man betrachte nun die Erkennung von Situationen, in denen eine Channel-Operation zwar begonnen, aber nie ausgeführt worden ist. 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

```
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. 7.: Beispielprogramm zyklisches Locking

tatsächlich auftretenden Channel hindeuten. Es sei allerdings auch hier dazu gesagt, dass eine solche hängende Operation nicht immer zu einem "echten" Deadlock führen muss. Man betrachte dazu das Beispiel in Abb. 9. Da auf dem Channel `x` nie gesendet wird, kommt es es in Zeile 3 zu 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.

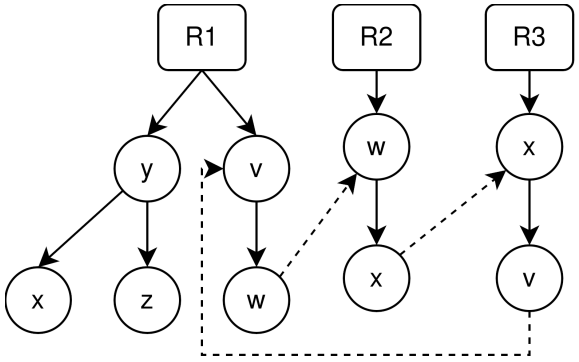


Abb. 8.: Graphische Darstellung des Lock-Graphen für das Beispielprogramm in Abb. 7. Durch die gestrichelten Pfeile wird der enthaltenen Zyklus angezeigt.

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

Abb. 9.: Hängender Channel ohne Deadlock

Es ist also möglich 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. 4 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. 10. Auf dem Channel `x` wird in 1 gesendet und kann

```
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. 10.: Beispielprogramm für unmögliche Synchronisation

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 10 aber auszuschließen, werden Vector-Clocks verwendet. Die grundlegende Idee basiert auf [6].
In einem ersten Durchlauf wird dabei der Trace mit Vector-Clock Informationen nach der Methode von Fidge [12] 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. 11. Man betrachte den Fall,

```
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. 11.: Beispielprogramm für die Betrachtung der Vector-Clocks

in dem 3 mit 4 und dann 1 mit 5 synchronisiert und 2 eine hängende Operation bildet. In diesem Fall erhält man folgenden Trace:

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

Aus diesem Trace lassen sich nun die Vector-Clocks für die einzelnen Channel-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. Für Close gilt, dass die Pre-Vectorclock gleich der Post-Vectorclock ist. Andere Elemente werden in den Vectorclock-Annotierte-Trace (VAT) nicht aufgenommen, da sie für die anschließende Analyse nicht

benötigt wird. Der VAT gibt sich in diesem Fall also als

$$
\begin{aligned}
&[[^{[2,0,0]}x?^{[3,0,2]}, {}^{[3,0,2]}\,x?^{[4,2,2]}]\\
&\;[^{[1,1,0]}x!^{[1,2,0]}, {}^{[1,2,0]}\,x?^{[max,max,max]}]\\
&\;[^{[2,0,1]}x!^{[2,0,2]}]]
\end{aligned}
$$

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 $s_{1,pre} \gtrless r_{pre}$. 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.

In gebufferten Channels ist nicht notwendig, dass Send und Receive gleichzeitig ausgeführt werden. Da der Trace somit bei einem Send ohne Receive, bzw. einem Receive ohne

gleichzeitigem Send ein gültiges Post-Event besitzt, wird dieses Problem nicht direkt erkannt. Situationen, bei denen eine gesendet Nachricht auf einem Channel nie ausgelesen wird lassen sich allerdings aus dem Trace direkt 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. 10 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. Wir betrachten den Buffer eines Channels als FIFO-queue. 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 Differenz zwischen der Anzahl der gesendeten Nachrichten in dem Send-Statement und der Anzahl der empfangenen Nachrichten in dem Receive-Statement kleiner als die maximale Kapazität des Channels ist.

Close

Versuch ein geschlossener Channel zu senden kommt es zu einem Laufzeitfehler, welcher in einem Programmabbruch führen kann. Solch eine Situation lässt sich dadurch erkennen, dass eine Close- und eine Send-Operation gleichzeitig ablaufen können, dass also die Pre-

oder Post-Vectorclock der Send-Operation unvergleichbar mit der Vectorclock der Close-Operation ist. Situationen, bei denen die Vectorclocks der Send-Operation streng später als die der Close-Operation sind führen immer dazu, das es zu einem einem Send auf einem geschlossenen Channel kommt. In diesem Fall kommt es also immer zu einem Laufzeitfehler des Programms, durch welchen das Programm abgebrochen wird.

2.5.3. 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 [13]. 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 [7] 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 mögli-

che 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 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 von Select wird auch für GoChan verwendet werden. Für jeden Durchlauf wird der Trace aufgezeichnet und dieser analysiert. Die Auswahl der Pfade wird dabei vereinfacht. Anders als in GFuzz basieren die betrachteten Ausführungsordnungen nicht auf schon versuchten Ordnungen, sondern es wird eine Menge von vollständig zufälligen Ordnungen betrachtet.

3. Instrumentierung

Um die Analyse eines Programms durchzuführen, wird der Programmcode so verändert, dass die Aufzeichnung der Traces, sowie die Ausführung der Analysen automatisiert abläuft. Um einem Programmcode nicht von Hand verändern zu müssen wurde ein Instrumenter implementiert, welcher dies automatisch durchführt.

3.1. Trace

Wie bereits 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. [14] oder [15] wird dabei der Tracer selbst implementiert und basiert nicht auf dem Go-Runtime-Tracer [16]. 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 Inkrementiert. 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)]$^{\text{nr}}$;
		[+]$^{\text{nr}}$; go { ts' := atomicInc(c); [wait(ts, nr)]$^{\text{id}}$; B};
ch <- i	$\Rightarrow$	ts := atomicInc(c); [pre(ts, ch.id, true)]$^{\text{id}}$; ch <- {i, id, ts};
		ts' := atomicInc(c); [post(ts', ch.id, true, id)]$^{\text{id}}$
<- ch	$\Rightarrow$	ts := atomicInc(c); [pre(ts, ch.id, false)]$^{\text{id}}$;
		{i, id_send, ts_send} := <-c; ts' := atomicInc(c);
		[post(ts', ch.id, false, id_send, ts_send)]$^{\text{id}}$; return i;
close(ch)	$\Rightarrow$	ts := atomicInc(c); close(ch); [close(ts, ch.id)]$^{\text{id}}$
select($e_i \rightsquigarrow B_i$)	$\Rightarrow$	ts := atomicInc(c); [pre(ts, e_1^*, ..., e_n^*, false)]$^{\text{id}}$;
		select($e_i \rightsquigarrow$ { ts' := atomicInc(c);
		[$\langle$post(ts, e_i.ch, false, e_i^m.id_send, e_i^m.ts_send) |
		post(ts, e_i.ch, true, id) $\rangle$]$^{\text{id}}$ B_i})
select($e_i \rightsquigarrow B_i$ | B_{def})	$\Rightarrow$	ts := atomicInc(c); [pre(ts, e_1^*, ..., e_n^*, false)]$^{\text{id}}$;
		select($e_i \rightsquigarrow$ { ts' := atomicInc(c);
		[$\langle$post(ts, e_i.ch, false, e_i^m.id_send, e_i^m.ts_send) |
		post(ts, e_i.ch, true, id) $\rangle$]$^{\text{id}}$ B_i} |
		ts' := atomicInc(c); [default(ts)]$^{\text{id}}$; B_{def})
mu.(Try)Lock()	$\Rightarrow$	ts := atomicInc(c); [lock(ts, mu.id, $\langle$-|t$\rangle$, $\langle$0\|1$\rangle$)]$^{\text{id}}$;
		mu.(Try)Lock();
mu.Unlock()	$\Rightarrow$	ts := atomicInc(c); mu.Unlock(); [unlock(ts, mu.id)]$^{\text{id}}$;
rmu.(Try)(R)Lock()	$\Rightarrow$	ts := atomicInc(c); rmu.(Try)(R)Lock();
		[lock(ts, rmu.id, $\langle$-|t|r|tr$\rangle$, $\langle$0\|1$\rangle$)]$^{\text{id}}$;
rmu.Unlock()	$\Rightarrow$	ts := atomicInc(c); rmu.Unlock(); [unlock(ts, rmu.id)]$^{\text{id}}$;

Für Receive, Send und Close auf Channels, allen Operation auf Mutexen sowie der Erzeugung von Strukturen als Ersatz für die eigentlichen Mutexe und Channels sind Funktionen implementiert, welche die Entsprechenden Operationen ersetzen und dabei sowohl die Aufzeichnung in dem Trace, als auch die eigentlichen Operationen durchführen. Hierbei sind besonders die Send- und Receive Operationen zu betrachten. Für den Trace muss dem Receive-Statement die Sender-Routine, sowie dessen momentane Zeitstempel bekannt

sein. Daher wird bei der eigentlichen Send-Operation nicht nur die eigentliche Information gesendet, sondern ein Objekt, welches die Information, sowie die Id der sendenden Routine und deren Zeitstempel beinhaltet.
Für die anderen Operationen sind Funktionen definiert, die in die entsprechenden Strukturen eingefügt werden, um so die Aufzeichnung in dem Trace zu gewehrleisten. Die eigentlichen Operationen werden in diesen Fällen aber weiterhin in dem eigentlichen Code vorgenommen.

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. 2.5.3. beschrieben. Die Implementierung basiert dabei zum größten Teil auf [7]. Abb. 12 zeigt ein Beispiel für die Instrumentierung eines Die Select-Operation wird durch eine Switch-Operation 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 [7] 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.

```
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. 12.: Beispiel für die Order-Enforcement-Instrumentierung eines Select-Statements vor (links) und nach der Implementierung (rechts). Ersetze ..... in dem Programm nach der Instrumentierung (rechts) durch das Programm vor der Instrumentierung (links). [7, gekürzt]

3.3. Automatisierter 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.

Der automatische Instrumenter besteht aus zwei Teilen. Zum einen werden alle in dem Programm vorhandenen ".go" Dateien instrumentiert. Zum anderen wird, basierend auf einem Template, eine neue Main-Datei erzeugt. Diese bestimmt die zu durchlaufenden Ordnungen der Select-Operationen und führt diese nacheinander aus.

3.3.1. Instrumentierung der Dateien

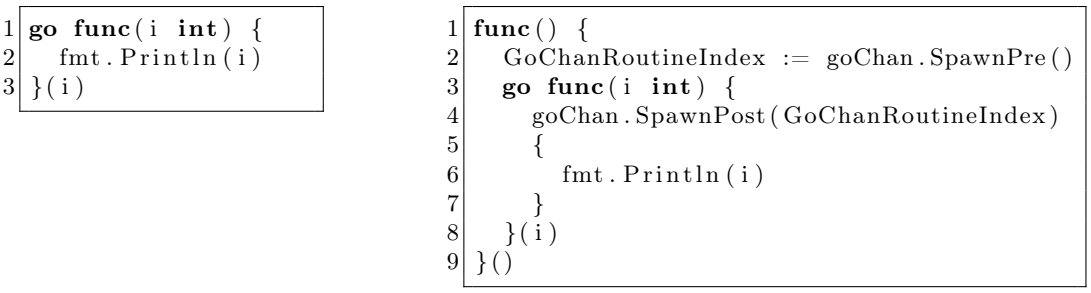
Bevor die einzelnen Dateien instrumentiert werden, wird der Ordner mit dem originalen Code durchlaufen, und seine Ordnerstruktur in den Output-Ordner kopiert. Anschließend werden alle Dateien durchlaufen. Handelt es sich nicht um ".go" Dateien werden sie einfach an die entsprechende Stelle im Output-Ordner kopiert. Handelt es sich um ".go" Dateien, dann wird der Programmcode so verändert, dass der erzeugte Code alle für den Analyzer notwendig Elemente enthält. Go besitzt in seiner Standart-Bibliothek ein Library zur Erzeugung und Bearbeitung des Abstract-Syntax-Trees einer Datei [17]. Zur Instrumentierung einer Datei wird dieser Baum mehrfach durchlaufen, dabei entsprechend angepasst und anschließend der veränderte AST als Code in der neuen Datei abgespeichert. Das mehrfache Durchlaufen ist notwendig, um Überschneidungen zwischen den einzelnen Teilen zu verhindern. Der Baum wird insgesamt drei mal durchlaufen.

1. Durchlauf: Im ersten Durchlauf wird der Import der GoChan-Bibliothek eingefügt und die Main-Funktion des Programms sowie Channels in Funktionsdeklarationen instrumentiert. Zuerst wird in jede Datei der Import der GoChan-Bibliothek eingeführt. In Go müssen alle importierten Bibliotheken verwendet werden. Andernfalls kommt es zu einem Compiling-Error. Um zu verhindern, dass dies bei einer Datei, die weder Mutex noch Channel-Operationen besitzt geschieht, wird nach der abgeschlossenen Instrumentierung das "goimport" tool aufgerufen, welches nicht verwendete import automatisch entfernt. Die Instrumentierung der Main-Funktion findet nur in der Main-Datei des Programms statt. Dabei wird der goChan-Analyzer initialisiert, sowie den Start der Analyze durch

eine defer-Statement vorbereitet. In der Initialisierung des Analyzers werden die Datenstrukturen zur Speicherung des Traces erzeugt. Außerdem wird eine zusätzliche Routine mit einem Timer gestartet. Läuft dieser Timer ab (default: 10s), nimmt der Analyzer an, dass es zu einem tatsächlichen Deadlock gekommen ist, und startet die Analyze, auch wenn das Programm noch nicht abgeschlossen ist. In diesem Fall wird das Programm nach der Analyze automatisch abgebrochen. Für Programme, die von selbst eine längere Laufzeit haben muss die Dauer des Timers entsprechen angepasst werden. Zusätzlich wird in die Datei mit der Main-Funktion eine globale Variable aufgenommen, welche die FetchOrder für die Select-Case des momentanen Durchlaufs speichert. Anschließend werden in allen Funktionen, welche Channels als Parameter oder Rückgabewerte haben, diese durch die entsprechenden Go-Chan-Objekte ersetzt.

2. Durchlauf Im zweiten Durchlauf werden die restlichen Channel Operationen instrumentiert. Hierbei werden alle Knoten des AST durchlaufen. Beinhaltet dieser eine Channel-Operation (z.B. Erzeugung einses Channels, Send, Receive usw.) wird sie durch die entsprechende Operation aus dem Go-Chan Analyzer ersetzt. Dabei muss beachtet werden, dass solche Operationen in den verschiedensten Operationen (z.B. in defer, range usw.) enthalten sein können, welche jeweils einzeln betrachtet und ersetzt werden müssen. Der Channel selber wird dabei durch ein Objekt ersetzt, welches den eigentlichen Channel, die Id des Channels, seine Kapazität sowie die Anzahl der bereits erfolgreich durchgeführten Send und Receive Operationen speichert. Die Definition dieses Struct wird, wie die Funktionen auf diesem Objekt durch GoChan definiert.

Auch die Erzeugung von neuen Go-Routinen wird hier instrumentiert. Ein Beispiel zur Instrumentierung der Erzeugung einer neuen Go-Routine findet sich in Abb. 13. `SpawnPre` erzeugt dabei das `signal` Element in dem Trace und gibt außerdem die Id der erzeugenden Routine zurück, welche in `SpawnPost` für die Erzeugung des `wait` Elements benötigt wird. Das umschließen des ganzen mit einer weiteren Funktion erleichtert die Implementierung des Instrumenters, ist aber nicht unbedingt notwendig.



```
1 go func(i int) {
2   fmt.Println(i)
3 }(i)
```

```
1 func() {
2   GoChanRoutineIndex := goChan.SpawnPre()
3   go func(i int) {
4     goChan.SpawnPost(GoChanRoutineIndex)
5     {
6       fmt.Println(i)
7     }
8   }(i)
9 }()
```

Abb. 13.: Instrumentierung der Erzeugung einer neuen Go-Routine. Links: vor der Instrumentierung, rechts: nach der Instrumentierung.

Zusätzlich werden auch die Select-Statements instrumentiert. Zum einen wird hier das Select-Statement wie in 2.5.3 beschrieben durch ein Switch-Statement mit mehreren Select-Statements ersetzt, um einen der Cases zu bevorzugen. Die Instrumentierung der Select-Statements für den Trace stellte dabei eine Komplikation da, da die Cases eines Select-Statements nur tatsächliche Channel-Operationen akzeptiert, diese also nicht direkt durch die Ersatzfunktionen des Analyzers ersetzt werden können. Die Lösung besteht darin, die Erzeugung der Events für den Tracer von dem eigentlichen Select-Case Statement zu trennen. Ein Beispiel dazu findet sich in Abb. 14. Zuerst wird die `goChan.PreSelect` Funktion aufgerufen, welche das Pre-Event in dem Trace erzeugt. Die Parameter dieser Funktion geben an, ob das Select-Statement einen Default-Case besitzt (in diesem Fall false), sowie für welche Channels es Cases gibt, und ob es sich bei diesen um Send- oder Receive-Statements handelt. Anders als in dem Original-Code werden in dem instrumentierten Code auf Channels nicht nur die Nachrichten selbst, sondern auch Informationen über die sendende Routine verschickt. Da in einem Select-Case die Channel-Operationen nicht direkt ersetzt werden können, müssen nun für alle Cases, welche durch eine Send-Operation ausgelöst werden diese Nachrichten erzeugt werden. In dem Beispiel git es einen Case, auf den dies Zutrifft. Durch `sel_RAjWwhTH := goChan.BuildMessage(1)` wird diese erzeugt. `RAjWwhTH` ist dabei lediglich eine zufällige Zeichenkette, um unterschiedliche Nachrichten voneinander trennen zu können. Anschließend beginnt das switch Statement. `goChanFetchOrder` ist dabei die globale Map, die für jedes Select-Statement den für diesen Durchlauf bevorzugte Case

```
1 select {
2   case a := <-x:
3     fmt.Println(a)
4   case y <- 1:
5     fmt.Println(1)
6 }
```

```
1  goChan.PreSelect(false, x.GetIdPre(true),
2                   y.GetIdPre(false))
3  sel_RAjWwhTH := goChan.BuildMessage(1)
4  switch goChanFetchOrder[1] {
5  case 0:
6    select {
7    case sel_VlBzgbaiCM := <-x.GetChan():
8      x.Post(true, sel_VlBzgbaiCM)
9      a := sel_VlBzgbaiCM.GetInfo()
10     fmt.Println(a)
11   case <-time.After(time.Second):
12     ....
13   }
14 case 1:
15   select {
16   case y.GetChan() <- sel_RAjWwhTH:
17     y.Post(false, sel_RAjWwhTH)
18     fmt.Println(1)
19   case <-time.After(time.Second):
20     ....
21   }
22 default:
23   ....
24 }
25
26
27
28 Mit .... als:
29 select {
30   case sel_VlBzgbaiCM := <-x.GetChan():
31     x.Post(true, sel_VlBzgbaiCM)
32     a := sel_VlBzgbaiCM.GetInfo()
33     fmt.Println(a)
34   case y.GetChan() <- sel_RAjWwhTH:
35     y.Post(false, sel_RAjWwhTH)
36     fmt.Println(1)
37   }
```

Abb. 14.: Instrumentierung eines Select-Statements. Links: vor der Instrumentierung, rechts: nach der Instrumentierung.

angibt. `1` ist dabei die Id des select-Cases. Dadurch, dass die Anzahl der möglichen Ids sehr groß gewählt ist und diese dann zufällig zugeteilt werden, ist die Wahrscheinlichkeit verschwindend gering, dass mehrere Select-Cases die selbe Id- erhalten.

Auf den Cases werden nun normale Channel-Operationen ausgeführt. Da aber nicht nur die eigentlichen Nachrichten versendet werden, müssen bei Receive-Statements mit anschließendem Assignement die eigentliche Nachricht aus dem Channel extrahiert werden. Außerdem

wird bei einem Send-Statement die Nachricht durch die vorher erzeugte ausgetauscht. Innerhalb des Cases wird die `goChan.Post` Funktion aufgerufen, um wenn der Case ausgewählt wurde das post-Event in dem Trace zu erzeugen.
Neben der eigentlichen Instrumentierung der Select-Statements werden die Select- Cases mit ihren Ids außerdem aufgezeichnet. Beinhaltet das Programm mindestens ein Select-Statement, so wird, nachdem der Baum vollständig durchlaufen wurde die Main-Funktion so verändert, dass es ein Command-Line-Argument `-order` annehmen kann. Dieses beinhaltet einen String, in dem die `FetchOrder` für den Programmdurchlauf an des Programm übergeben werden kann.

3. Durchlauf Im dritten Durchlauf werden nun noch die Mutex-Operationen durch, von dem GoChan-Analyzer zur Verfügung gestellt Funktionen ersetzt, welche das eigentliche Locking als auch die Aufzeichnung des Traces übernehmen. Da es hierbei keine speziellen Strukturen wie select o.ä. gibt, ist eine einfache direkte Ersetzung der Operationen möglich. Die Objekte, welche die Mutexe dabei ersetzen speichern dabei lediglich den eigentlichen Mutex, welcher das Locking tatsächlich ausführt, sowie die Id des Mutex. Für Mutex und RW-Mutex ist dabei jeweils ein eigenes Objekt definiert.

3.3.2. Neue Main-Datei

Der in 3.3 erzeugt Code ist nun prinzipiell Lauffähig. Allerdings wird er nur einmal durchlaufen. Aufgrund der Select-Statements soll das Programm nun aber mehrfach, mit verschiedenen Select-Orders durchlaufen werden. Dazu wird eine neue Main-Datei erzeugt, mit welcher das eigentliche Programm gestartet wird. Diese Datei wird über ein Template erzeugt, mit welchem, basierend auf den Aufzeichnungen der instrumentierten Select-Cases die neue Datei erzeugt wird. Zuerst erzeugen wir eine Map, die für jedes Select-Statement die Anzahl der Cases, also die Anzahl der Cases in dem Switch Statements speichert. Anschließend werden die `FetchOrder` für die eigentlichen Durchläufe erzeugt. Dazu wird für jedes Select-Statement eine gültige Case-Id zufällig ausgewählt. Der so erzeugte Ablauf wird

nun, wenn der selbe Ablauf nicht bereits erzeugt worden ist, gespeichert. Dabei wird gezählt, wie oft ein Ablauf erzeugt worden ist, welcher bereits zuvor erzeugt worden ist. Erreicht dieser Wert einen vorgegebenen maximalwert, dann nimmt das Programm an, dass genug verschiedene Abläufe erzeugt werden. Außerdem kann eine maximale Anzahl an Durchläufen festgelegt werden, um zu verhinder, dass die Dauer der Analyse zu lange wird, wenn sehr viele Select-Statements, bzw. sehr viele Select-Cases vorhanden sind. Anschließend wird das Instrumentierte Programm für jeden dieser Abläufe über einen `exec.Command` Befehl ausgeführt, wobei die Ordnung als Command-Line-Argument übergeben wird (also z.B. `exec.Command(./Examples/select/select -order="0,1")`, wobei hierbei für das Select mit Id 0 der Case 1 bevorzugt ausgewählt wird). Für das Programm wird nun der Trace aufgezeichnet und dieser anschließend analysiert (s. 4).

(TODO: Consoliedierng der Ergebnisse)

3.3.3. 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. 15)

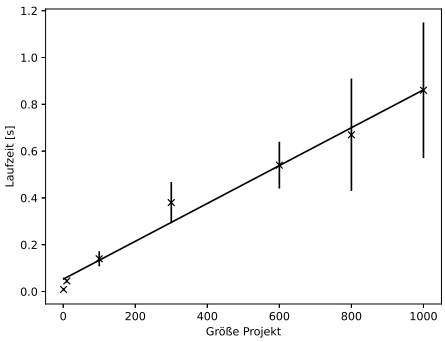


Abb. 15.: Laufzeit des Instrumenters

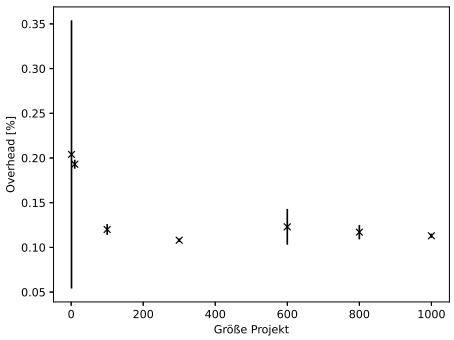


Abb. 16.: Prozentualer Overhead des Tracers ohne Analyse

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. 15 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 der Overhead des instrumentierten Codes im Vergleich zum originalen Code betrachtet werden. Hierbei wird nur die Laufzeit eines Durchlaufs des eigentlichen Programs (ohne Wiederholung aufgrund von Select), 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. 16 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 Verhältnis zu der Größe bzw. der Laufzeit des gesamten Programms ist, kann dieser Wert

abhängig von dem tatsächlichen Programm start schwanken. Dies wird unter anderem klar, wenn man den Overhead für ht-cat (9 ± 3 %) und go-dsp (60 ± 18 %) welche 51 ± 21 Prozentpunkte aufeinander liegen.

4. Analyzer

Das folgende Kapitel soll sich nun mit der Implementierung der Analyze des in Kap. 3.1 erstellten Trace befassen.

4.1. Aufzeichung des Trace

Der Trace wird wie in Kap. 3.1 beschriebenen erzeugt. Er wird dabei durch ein Slice von Slices gespeichert, wobei jeder Slice den Trace einer Routine speichert. Dazu wird für jedes Trace-Element ein Struct definiert, welches alle notwendigen Informationen speichert. Über ein Interface wird dafür gesorgt, dass diese Elemente alle in das Slice des Trace eingefügt werden kann. Dabei wird über einen Mutex dafür gesorgt, dass nicht zwei Routinen gleichzeitig ein Element in den Trace einfügen können
Jeder Routine wird ein fortlaufender Wert eines atomaren Zählers zugeordnet. Um die interne Id einer Routine zu erhalten, und auf die dem Trace zugeordnete Id zuzuordnen wird eine externe Bibliothek GoId [18] verwendet.
Auch die Ids für Channels und Mutexes werden durch laufende, atomare Zähler implementiert, welche in den Objekten für Channels und Mutexes gespeichert werden.

4.2. Mutex

Um potenzielle Deadlocks durch Mutexe erkennen zu können werden nun, wie in Kap. 2.4.1 beschrieben, Lock-Bäume verwendet. Diese 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 eine s.g. `dependendency` erzeugt und gespeichert. Diese beinhaltet das Lock `x` sowie eine Liste aller von der Routine momentan gehaltenen Locks, dem s.g. `holdingSet hs`. Dieses entspricht gerade `currentLocks`. Diese `dependendency` stellt also eine Menge von Kanten von in dem Lockgraphen der Routine da. 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 letzte 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 aus den Bäumen zusammengesetzten Graph ein Zyklus existiert. Nicht alle Zyklen bilden dabei gültige Zyklen. zum Beispiel 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. Gültige

Zyklen lassen sich durch dir folgenden Formeln charakterisieren.

$$\forall\ i, j \in \{1, ..., n\}\ \neg(hs_i \cap hs_j = \emptyset) \rightarrow (i = j) \tag{4.2.a}$$

$$\forall\ i \in \{1, ..., n-1\}\ mu_i \in hs_{i+1} \tag{4.2.b}$$

$$mu_n \in hs_1 \tag{4.2.c}$$

$$\forall i \in \{1, \ldots, n-1\}\ read(mu_i) \rightarrow (\forall mu \in hs_{i+1}(mu = mu_i) \rightarrow \neg read(mu)) \tag{4.2.d}$$

$$read(mu_n) \rightarrow (\forall\ mu \in hs_1 : (mu = mu_n) \rightarrow \neg read(mu)) \tag{4.2.e}$$

$$\begin{aligned}\forall\ i, j \in \{1, ..., n\}\ \neg(i = j) \rightarrow (\exists\ mu_1 \in hs_i\ \exists\ mu_2 \in hs_j((mu_1 = mu_2) \rightarrow \\ (read(mu_1) \wedge read(mu_2))))\end{aligned} \tag{4.2.f}$$

Dabei bezeichnet mi_i den Mutex hs_i das holdingSet der i-ten in dem Zyklus. Equation (4.2.a) stellt sicher, dass das selbe Lock nicht in dem HoldingSet von zwei verschiedenen Routinen auftauchen kann. Equation (4.2.b) und Equation (4.2.c) sorgen dafür, dass es sich bei der Kette tatsächlich um einen Zyklus handelt, dass also das Lock einer Dependency immer in dem HoldingSet der nächsten Dependency enthalten ist und das Lock der letzten Routine wiederum im HoldingSet der ersten Routine liegt, um den Zyklus zu schließen. Equation (4.2.d) bis Equation (4.2.f) beschäftigen sich mit dem Einfluss von RW-Locks auf die Gültigkeit von Zyklen. Auch wenn Equation (4.2.a) bid Equation (4.2.c) erfüllt sind, ist dies dennoch keine gültige Kette, wenn sowohl der Mutex mu_i als auch der Mutex mu in hs_{i+1}, für die $mu = mu_i$ gilt, beides Reader-Locks sind, also Locks welche durch eine RLock-Operation erzeugt worden sind. Dass solche Pfade ausgeschlossen werden wird durch Equation (4.2.d) und Equation (4.2.e) sichergestellt. Equation (4.2.f) beschäftigt sich mit Gate-Locks. Dabei handelt es sich um Situationen, bei denen mehrere Teile des Programmcodes, welche zu einem Deadlock führen könnten durch ein Lock umschlossen sind, in der Praxis also nicht gleichzeitig ausgeführt werden und somit einen Deadlock verhindern. Die Regel besagt nun, dass wenn es einen Mutex gibt, der in den HoldingSets zweier verschiedener Dependencies in dem Pfad vorkommt, so müssen beide diese Mutexe Reader-Locks sein. Sind sie es nicht, handelt es sich um Gate-Locks, und der entsprechende Pfad kann somit nicht zu einem Deadlock führen.

Für die Suche nach solchen Zyklen wird eine Depth-First-Search auf den gesammelten Dependencies ausgeführt. Dazu wird zuerst eine Dependency auf einen Stack gelegt. Der Stack entspricht immer dem momentan betrachteten Pfad. Anschließend werden schrittweise weitere Dependencies auf den Stack gelegt, wobei darauf geachtet wird, dass aus jeder Routine immer nur maximal eine Dependency auf dem Stack liegt. Bevor eine Dependency zu dem Stack hinzugefügt wird, wird überprüft ob der durch den Stack betrachtete Pfad einen gültigen Pfad bilden würde, ob also Equation (4.2.a), Equation (4.2.b), Equation (4.2.d) und Equation (4.2.f) immer noch gelten würden. Ist dies nicht der Fall, so wird die Dependency nicht auf den Stack gelegt. Werden die Regeln hingegen erfüllt, dann wird überprüft, um der Stack nun einen gültigen Zyklus enthält, also auch Equation (4.2.c) und Equation (4.2.e) gültig sind. In diesem Fall wurde ein potenzielles Deadlock gefunden, und dies ausgegeben. Andernfalls werden weiter Dependency auf dem Stack hinzugefügt. Dies wird wiederholt, bis es keine Dependency mehr gibt, die auf den Stack VaT gelegt werden könnte. In diesem Fall werden per Backtracking Dependencies von dem Stack entfernt, so dass andere Pfade ausprobiert werden können. Dies wird so lange durchgeführt. Bis alle gültigen Kombinationen durchprobiert worden sind.

4.3. Channels

Die Analyse des Programs zur Erkennung und Beschreibung von durch Channels ausgelösten Problemen läuft in mehreren Schritten ab. Zuerst werden die Vectorclock-Informationen der einzelnen Operationen bestimmt und mit diesen ein Vectorclock-Annotated-Trace (VaT) erzeugt. Anschließend wird nach hängenden Events (pre ohne post) und nicht-leere Channels, also gebufferte Channels, welche bei Terminierung des Programms noch Nachrichten in ihrem Buffer beinhalten gesucht. Werden solche Probleme entdeckt, bestimmt der Analyzer mögliche andere Kommunikationspartner für Send-Operationen auf Channels. Zum Schluss wird noch nach Situationen gesucht, bei denen es zu einem Send auf einen geschlossenen Channel kommen kann, da solche Situationen zu Laufzeitfehlern führen, welche den Abbruch

eines Programms zur Folge haben. Receives auf geschlossenen Channels werden nicht betrachtet, da diese lediglich einen Null-Wert zurückgeben und nicht blocken, somit also nicht zu Laufzeitfehlern führen.

Bestimmung der Vectorclocks Basierend auf dem aufgezeichneten Trace soll nun ein Vectorclock annotierter Trace (VaT) erzeugt werden. Dieser besteht aus einer Reihe von `vcn`'s, welche jeweils eine Send-, Receive- oder Close-Operation repräsentieren. Andere Operation, wie z.B. signal-wait werden zwar bei der Berechnung der Vectorclocks beachtet, allerdings nicht in den VaT aufgenommen, da sie für die weitere Analyse nicht benötigt werden. Ein `vcn` beinhaltet dabei die Channel-Id, die Routine, ob es sich um Send- oder Receive handelt (bei Close beliebig gesetzt), ein Counter für die Anzahl der bereits erfolgreich abgeschlossenen Send- bzw. Receive-Operationen bei der Ausführung des Send- oder Receive (bei Close -1), die Position der Operation im Programmcode sowie die Pre- und Post-Vectorclocks der Operation. Eine Close Operation wird dabei dadurch erkannt, dass die Pre- und Post-Vectorclocks übereinstimmen.

Bevor der eigentliche VaT erzeugt wird werden erst die Vectorclocks zu allen Zeitpunkten bestimmt. Dazu wird für jede Routine eine Vectorclock initialisiert. Anschließend werden die Elemente in der Reihenfolge durchlaufen, in der sie in den Trace eingefügt wurden, also aufsteigend sortiert nach dem Timestamp der Trace-Element. Für jeden Zeitpunkt wird nun eine Vectorclock berechnet, wobei für jeden Timestamp immer diejenige Vectorclock gespeichert wird, die der Vectorclock entspricht, auf welcher die entsprechende Operation ausgeführt wurde. Für die Berechnung der Vectorclocks werden signal-wait Paare wie das Senden einer Nachricht von signal nach wait betrachtet.
Für send (post) und Signal bzw. Receive (post) und Wait werden nun die Vectorclocks wie in Kap. **??** beschrieben aktualisiert. Für Send und Signal wird lediglich der eigene Timestamp in der eigenen Vectorclock um eins erhöht. Für die Aktualisierung bei einem Receive oder Wait wird die Vectorclock zur Zeit von Send oder Signal benötigt. Da diese in jedem Fall vor dem Receive oder Wait erzeugt worden sind, wurden sie bereits berechnet. Da für die

Receive-Elemente in dem Trace die Zeitstempel der Send-Operationen gespeichert sind, ist eine eindeutige Zuordnung der Send- und Receive-Statements möglich. Für die Signal- und Wait-Elemente ist jeweils die Id der neu erzeugten Routine gespeichert. Es ist also auch hier eine eindeutige Zuordnung möglich. Die Vectorclock der Send- bzw. Signal-Operation kann also immer eindeutig bestimmt werden und die Vectorclock somit wie in Kap. **??** beschrieben aktualisiert werden.
Für alle anderen Element, also alle Pre-Elemente, Close-Operationen und Mutex-Operationen werden die Vectorclocks lediglich kopiert.

Nach der Berechnung der Vectorclocks kann nun der VaT bestimmt werden. Dazu wird nun der Trace für die einzelnen Routinen durchlaufen. Bei jedem Pre- und PreSelect-Element wird eine `vcn` erzeugt. Dazu wird der restliche Trace der selben Routine durchlaufen um das zugehörige Post-Element zu finden. Wird diese gefunden wird das `vcn` erzeugt, wobei die Pre- und Post-Vectorclock über den Zeitstempel der Pre- und Post-Elemente aus der Liste der Vectorclocks übernommen wird. Wird kein Post-Element gefunden, handelt es sich also um ein hängendes Event, wird die Pre-Vectorclock auf die Vectorclock des Pre-Events und alle Elemente der Post-Vectorclock auf `maxInt`, als den maximal möglichen Wert gesetzt.
Für Close-Elemente gibt es keine nur ein Element in dem Trace. Aus diesem Grund besitzt das Element nur eine Vectorclock. Pre- und Post-Vectorclock werden dabei auf die gleiche Vectorclock gesetzt.

Suche nach hängenden Events Die Suche nach hängenden Events wird auf dem ursprünglichen Trace ausgeführt. Dazu wird der Trace jeder einzelnen Routine durchlaufen. Für Pre- und PreSelect-Event wird dabei überprüft, ob ein entsprechendes Post-Event existiert. Ein Element gilt dabei als entsprechendes Post-Event, wenn es sich nach dem Pre- oder PreSelect-Event befindet und dabei auf dem selben Channel und der selben Operation (Send oder Receive) arbeitet. Wird eine solche Situation gefunden, wird eine Nachricht, welche die Situation beschreibt zwischengespeichert und nach dem Durchlaufen aller Routinen die Liste der gefundenen Situationen zurückgegeben.

Suche nach nicht-leeren Channels Um nicht-leere Channels, also gebufferte Channels, bei denen nicht alle gesendeten Nachrichten gelesen wurden, zu finden, wird der Vectorclock annotierte Trace durchlaufen und für jeden Channel in dem Programm gezählt, wie viele erfolgreiche Send- und Receive-Operationen auf dem Channel erfolgreich ausgeführt wurden. Eine Operation gilt in dem VaT als erfolgreich ausgeführt, wenn das erste Element der Vector-Clock nicht "maxInt" entspricht. Gibt es eine Differenz zwischen der Anzahl der gesendeten und der empfangen Nachrichten, dann lässt sich daraus schließen, dass sich bei der Terminierung des Programms noch ungelesene Nachrichten in dem Channel befanden. Für jeden dieser Fälle wird eine entsprechende Nachricht zurückgegeben.

Suche nach potenziellen Kommunikationspartnern Zur Suche nach alternativen Kommunikationspartner werden Kombinationen von zwei Elementen in dem VaT betrachtet. Dabei werden all diejenigen Elemente verglichen, bei welchen beide Operationen auf dem selben Channel ausgeführt werden und eines der Elemente ein Send- und das andere eine Receive-Operation ist. Zwei Operationen werden als potenzielle Kommunikationspartner angesehen, wenn der Channel ungebuffert ist und entweder die Pre- oder die Vectorclocks der beiden Operationen unvergleichbar sind, oder wenn der Channel gebuffert ist Anzahl der Send-Operationen der Anzahl der Receive-Operationen auf dem Channel entspricht. Für den gebufferten Channel müssen Send- und Receive nicht gleichzeitig ausgeführt werden. Aus diesem Grund müssen die Vectorclocks nicht unvergleichbar sein. Allerdings sind gebufferte Channel als FIFO-Queue implementiert. Die n-te Receive-Operation erhält also die Nachricht der n-ten Send-Operation.

Senden auf geschlossenen Channel Für die Suche nach Situationen, die dazu führen können, dass auf einem geschlossenen Channel gesendet wird, werden die Vectorclock aller in dem Trace vorhandenen Close-Operationen mit den Vectorclocks aller Send-Operationen auf dem selben Channel verglichen. Ist mindestens eine der beiden Vectorclocks unvergleichbar, dann nimmt das Programm an, das eine Send-Operation auf einen geschlossenen Channel möglich ist, und gibt eine entsprechende Warnung zurück.

4.4. Output

Da das Programm gegebenenfalls mehrfach analysiert wird ist es möglich, dass die selben Probleme mehrfach erkannt werden. Es ist daher nicht sinnvoll die Informationen direkt nach jedem Durchlauf auszugeben. Daher werden die Fehler gesammelt, und dabei jeweils notiert, bei welchen Durchlaufen die Fehler aufgetreten sind. Nach Abschluss aller Durchläufe werden die gesammelten Fehler vollständig ausgegeben.

5. Auswertung

Im folgenden soll betrachtet werden, wie gut der beschriebene Detektor in der Lage ist, Situationen wie in Kap. 1 beschrieben zu erkennen. Dabei werden sowohl künstlich konstruierte Situationen, als auch tatsächliche Programme betrachtet. Für die Betrachtung tatsächlicher Programme werden Programme aus GoBench bzw. Goker [8] verwendet. Dabei handelt es sich um eine Sammlung von Programmteilen mit Concurrency-Bugs aus 9 großen open-source Anwendungen wie z.B. Kubernetes und Moby. Beschreibungen der Probleme, sowie die Ergebnisse des Detektors befinden in Anhang A.

5.1. Standardprogramme

Für die Analyse wurden insgesammt 35 Standartprobleme betrachtet. Dabei wurde überprüft, ob der Detektor in der Lage ist, das in dem Programm erhaltene Problem richtig zu erkennen, bzw. zu erkenne wenn die vorliegende Situation nicht zu einem Problem führen kann. Eine tabellarische Beschreibung der betrachteten Situationen, sowie der Ergebnisse des Detektors ist in Tab. 2 in Anhang A aufgeführt. Abb. 17 - 20 zeigen die Ergebnisse der Auswertung.

Von den 43 Programmen konnten 38 korrekt kategorisiert werden. Dabei bestehen 18 Probleme aus Problemen mit Mutexen, 18 aus Problemen mit Channel und 7 mit einem Mix aus Mutexen und Channel. Abbildungen refChap:Eval-Sec:Stand-Fig:Total bis 20 geben an, welcher Anteil der Betrachteten Standartprobleme korrekt erkannt wurde.

Für Programme, bei denen der Fehler auf die Verwendung von Mutexen basiert, konnten 17 der 18 Probleme richtig kategorisiert werden. Dies entspricht ca. 94.4%.
Bei den Programmen, bei welchen es durch Channel zu Problemen kommen kann, konnten 16 der 18 Programme richtig kategorisiert werden, was einem Anteil von ca. 88.9% entspricht.
Bei Programmen, welche sowohl Mutexe als auch Channels verwenden liegt die Erfolgsquote mit 5 aus 7 (71.4%) am niedrigstem. Da der Detektor zwei verschiedene Methoden verwendet, um Probleme mit Mutexen und Probleme mit Channels zu erkenne, aber keine direkte Methode für die Erkennung von Problemen besitzt, welche durch eine Kombination der beiden entstehen, werden solche Situationen nicht direkt erkannt. Sie werden nur dann erkannt, wenn sie dazu führt, dass einer der beiden Mechanismen sie erkennen kann. Es ist daher nicht verwunderlich, dass hierbei eine höhere Fehlerquote betrachtet werden kann, als wenn man die beiden Situationen einzeln betrachtet.
Insgesamt hat der Detektor für die betrachteten Programme eine Trefferwahrscheinlichkeit von 88.4%.
Es sei noch dazu gesagt, dass die betrachteten Programme immer so implementiert worden sind, dass die entsprechenden Situationen auch in dem Durchlauf auftreten. Es ist allerdings auch möglich, dass Situationen bei den Durchläufen nicht durchlaufen werden, z.B. wenn sie sich ein einem Konditionellen Block (If) befinden, bei welchem die Bedingung während keinem der Durchläufe wahr wird. Da die entsprechenden Operationen somit nicht aufgezeichnet werden können, ist es demnach logischerweise auch nicht möglich, dass der Detektor die entsprechenden Situationen erkennt.

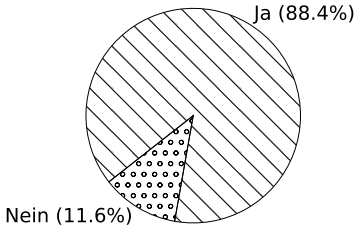


Abb. 17.: Verteilung der Ergebnisse für Standardprogramme

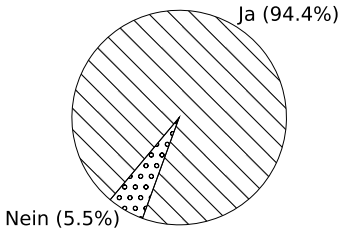


Abb. 18.: Verteilung der Ergebnisse für Standardprogramme mit Mutexen

Abb. 19.: Verteilung der Ergebnisse für Standardprogramme mit Channel

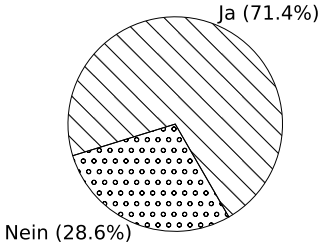


Abb. 20.: Verteilung der Ergebnisse für Standardprogramme mit Mutexen und Channel

5.2. GoKer

Für die Analyse wurden insgesamt 35 Programme betrachtet. Die betrachteten Programme und deren Ergebnisse befinden sich in Tab. 3 in Anhang A. Von den Programmen betrafen 15 die Verwendung von Resource wie Mutexen (14 korrekt erkannt), 11 die Verwendung von Kommunikationen und 9 eine Kombination aus Mutexen und Channel (7 korrekt erkannt). Die Erfolgsraten des Detektors für die Programme aus GoKer (Abb. 21 bis 24) stimmen dabei in etwa mit denen der Standardprogramme überein. Für die Analyse wurden dabei nur solche Programme ausgewählt, welche basieren auf ihrer Beschreibung für den Detektor theoretisch erkennbare Situation enthielt. Situationen, welche sich auf andere Concurrency-Bugs, z.B. Race-Conditions bezogen wurden nicht betrachtet. Die Betrachtung der Programme aus Goker hat einen Nachteil der hier verwendeten Methode, bzw. der Implementierung deutlich

gemacht. Der Instrumenter ist nur in der Lage den vorliegenden Code zu instrumentieren. Es kann aber vorkommen, dass in einem Programm externe Funktionen verwendet werden, welche Mutexe oder Channel als Parameter oder Rückgabewerte besitzen. Da bei der Instrumentierung Mutexe und Channel durch eingens implementierte Objekte ersetzt werden, externe Funktionen aber nicht entsprechend Instrumentiert werden können kommt es hierbei zu Compiler-Fehlern. Die entsprechenden Programme sind daher nicht Lauffähig und können somit auch nicht analysiert werden. Programme aus GoKer, bei denen dies der Fall war wurden für die Analyse nicht betrachtet.

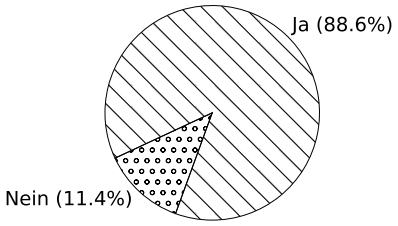


Abb. 21.: Verteilung der Ergebnisse für Programme aus GoKer

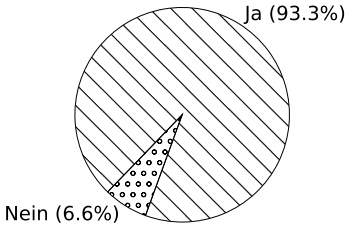


Abb. 22.: Verteilung der Ergebnisse für Programme aus GoKer mit Mutexen

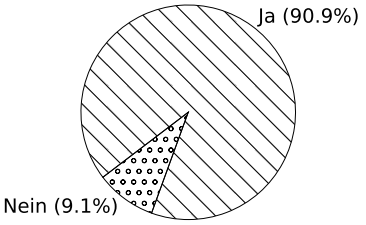


Abb. 23.: Verteilung der Ergebnisse für Programme aus GoKer mit Channel

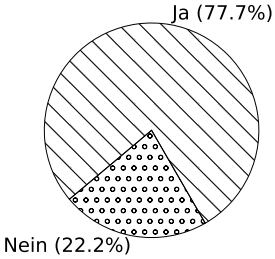


Abb. 24.: Verteilung der Ergebnisse für Programme aus GoKer mit Mutexen und Channel

6. Zusammenfassung

Ziel dieser Arbeit war es einen Detektor für von Mutexen und Channels erzeugte Concurrency-Bugs zu entwickeln und zu implementieren.
Der implementierte Detektor vereinigt und erweitert dabei verschiedene Methoden zur Erkennung solcher Probleme. Der Detektor erzeugt dynamisch einen Trace eines vorliegenden Programms, welcher im Anschluss analysiert werden kann. Für Deadlocks durch Mutexe werden dabei unter anderem Lock-Bäume verwendet. Für Channel-Operationen wird der Trace mit Vector-Clocks erweitert, durch welche Schlussfolgerungen auf mögliche Kommunikationspartner oder das potenzielle Senden auf geschlossene Channels erkannt werden. Bei der Anwendung auf konstruierte Probleme und tatsächliche Programme ist der so entwickelte Detektor in der Lage etwa 88.5% aller betrachteten Situationen richtig zu kategorisieren.

A. Beschreibung der betrachteten Programme

Beschreibung	**Ok?**
Potenzielles Deadlock durch zyklisches Locking von zwei Mutexen	Ja
Potenzielles Deadlock durch zyklisches Locking von drei Locks	Ja
Locking von zwei Locks, welche keinen Deadlock bilden, da Locking nicht zyklisch ist	Ja
Zyklisches Locking welches durch Gate-Locks nicht zu einem Deadlock führen kann	Ja
Potenzielles Deadlock, welches durch Verschachtlung mehrerer Routinen (fork/join) verschleiert wird	Nein
Deadlock durch doppeltes Locken	Ja
Tatsächliches Deadlock durch zyklisches Locking von Locks in zwei Routinen	Ja
Tatsächliches Deadlock durch zyklisches Locking von Locks in drei Routinen	Ja
Doppeltes Locking mit TryLock (TryLock $\rightarrow$ Lock)	Ja
Kein doppeltes Locking mit TryLock (Lock $\rightarrow$ TryLock)	Ja
Deadlock durch zyklisches Locking mit TryLock	Ja
Zyklisches Locking, welches durch TryLock nicht zu einem Deadlock führen kann	Ja
Potenzielles Deadlock mit RW-Mutexe in zwei Routinen	Ja
Kein potenzielles Deadlock mit RW-Mutexe in zwei Routinen	Ja
Kein potenzielles Deadlock, wegen Lock von RW-Locks als Gate-Locks	Ja

Potenzielles Deadlock, da R-Lock von Deadlock nicht als Gate-Lock funktioniert	Ja
Doppeltes Locking von RW-Locks, welches zu Deadlock führt (Lock→Lock, RLock→Lock, Lock→Rlock)	Ja
Doppeltes Locking von RW-Locks, welches nicht zu einem Deadlock führt(RLock→Rlock)	Ja
Deadlock oder hängende Routine durch Receive auf ungepuffertem Channel ohne Send	Ja
Deadlock oder hängende Routine durch Receive auf gepuffertem Channel ohne Send	Ja
Deadlock oder hängende Routiune durch 2-faches Receive mit 1-fachem Send $[R_0$: $\{\leftarrow x_1^0\}$, R_1: $\{x_1^0 \leftarrow 1\}$, R_2: $\{\leftarrow x_1^0\}]$	Ja
Deadlock oder hängende Routiune durch 2-faches Send mit 1-fachem Receive $[R_0$: $\{x_1^0 \leftarrow 1\}$, R_1: $\{x_1^0 \leftarrow 1\}$, R_2: $\{\leftarrow x_1^0\}]$	Ja
Deadlock oder hängende Routine durch Send auf ungepuffertem Channel ohne Receive	Ja
Kein Deadlock aber ungelesene Nachricht in Channel durch Send in Main-Routine auf gepuffertem Channel ohne Receive	Ja
Deadlock durch zweifaches Send auf gebufferten Channel in Kapazität 1 in Main-Routine ohne Receive	Ja
Ungelesene Nachricht bei $[R_0$: $\{c_1^1 \leftarrow 1; c_1^1 \leftarrow 1; c_1^1 \leftarrow 1\}$, R_1: $\{\leftarrow c_1^1\}$, R_2: $\{\leftarrow c_1^1\}]$ sowie Erkennung der potenziellen Kommunikationspartner	Ja
Keine Probleme bei $[R_0$: $\{c_1^1 \leftarrow 1; c_1^1 \leftarrow 1\}$, R_1: $\{\leftarrow c_1^1\}$, R_2: $\{\leftarrow c_1^1\}]$	Ja
Kein Kommunikationspartner wenn Receive in Fork bei ungebuffertem Channel $[R_0$: $\{c_1^0 \leftarrow 1$; fork $R_1\}$, R_1: $\{\leftarrow c_1^0\}]$	Ja
Mögliche Kommunikationspartner wenn Receive in Fork bei gebuffertem Channel $[R_0$: $\{c_1^1 \leftarrow 1$; fork $R_1\}$, R_1: $\{\leftarrow c_1^1\}]$	Nein

Deadlock bei Wahl eines bestimmten Select-Case [R_0: {$\leftarrow c_3^0$}, R_1: {$c_1^0 \leftarrow 1$}, R_2: {$c_2^0 \leftarrow 1$}, R_3: {select { case $\leftarrow c_1^0 \Rightarrow \{c_3^0 \leftarrow 1\}$, case $\leftarrow c_2^0 \Rightarrow \{\leftarrow c_3^0\}\}\}$]	Ja
Deadlock bei Wahl eines bestimmten Select-Case [R_0: {$\leftarrow c_3^0$}, R_1: {$c_1^0 \leftarrow 1$} R_2: {select { case $\leftarrow c_1^0 \Rightarrow \{c_3^0 \leftarrow 1\}$; default $\Rightarrow \{\leftarrow c_3^0\}\}\}$]	Ja
Tatsächliches Send auf geschlossenen Channel	Ja
Potenzielles aber nicht tatsächliches Send auf geschlossenen Channel Channel	Ja
Kein Problem, wenn Channel erst nach letztem Send geschlossen werden kann [R_0: {$c_1^0 \leftarrow 1$; close(c_1^1)}, R_1: {$\leftarrow c_1^1$}]	Ja
Kein Problem, wenn Channel erst nach letztem Send geschlossen werden kann [R_0: {$\leftarrow c_1^0$; close(c_1^0)}, R_1: {$c_1^1 \leftarrow 1$}]	Nein
Korrekte Kommunikationspartner bei [R_0: {$c_1^1 \leftarrow 1$; $c_1^1 \leftarrow 1$; $c_1^1 \leftarrow 1$}, R_1: {$\leftarrow x$; $\leftarrow x$}] (letztes Send hat keinen Kommunikationspartner)	Ja
Deadlock, da gleichzeitiges Send und Receive durch Mutex Lock verhindert wird [R_0: {m_1.Lock; $\leftarrow c_1^0$; m_1.Unlock}, R_1: {m_1.Lock; $c_1^0 \leftarrow 1$; m_1.Unlock}]	Ja
Kein Problem, da gleichzeitiges Send und Receive durch RWMutex R-Lock nicht verhindert wird [R_0: {m_1^r.RLock; $\leftarrow c_1^0$; m_1^r.RUnlock}, R_1: {m_1^r.RLock; $c_1^0 \leftarrow 1$; m_1^r.RUnlock}]	Ja
Kein potenzielles zyklisches Locking da Operationen durch Channel-Operation getrennt sind [R_0: {$\leftarrow c_1^0$; m_1.Lock; m_2.Lock; m_2.Unock; m_1.Unlock;}, R_1: {m_2.Lock; m_1.Lock; m_1.Unock; m_2.Unlock; $c_1^0 \leftarrow 1$}]	Nein
Tatsächlicher Deadlock, da Send durch Lock nicht erreicht werden kann [R_0: {m_1.Lock; $c_1^0 \leftarrow 1$; m_1.Unlock}, R_1: {m_1.Lock; $\leftarrow c_1^0$; m_1.Unlock}]	Ja
Potenzieller aber nicht tatsächlicher Deadlock (R_1 vor R_2), da Send durch Lock nicht erreicht werden könnte (R_2 vor R_1) [R_0: {m_1.Lock; $c_1^0 \leftarrow 1$; m_1.Unlock}, R_1: {m_1.Lock; $\leftarrow c_1^0$; m_1.Unlock}]	Nein

Potenzielles zyklisches Locking bei Wahl eines Select-Cases [R_0: {m_1.Lock; m_2.Lock; m_2.Unlock; m_1.Unlock}, R_1: {select {case $\leftarrow c_1^0 \Rightarrow$ {m_2.Lock; m_1.Lock; m_1.Unlock; m_2.Unlock}, default $\Rightarrow$ {}}; R_2: {c_1^0 <- 1}}]	Ja
Potenzielles zyklisches Locking bei Wahl eines DefaultSelect-Cases [R_0: {m_1.Lock; m_2.Lock; m_2.Unlock; m_1.Unlock}, R_1: {select {case $\leftarrow c_1^0 \Rightarrow$ {}, default $\Rightarrow$ {m_2.Lock; m_1.Lock; m_1.Unlock; m_2.Unlock}}; R_2: {c_1^0 <- 1}}]	Ja

Tab. 2.: Beschreibung der für die Auswertung betrachteten Standardprogramme sowie Information (Ok?) ob die Situation korrekt erkannt wurde. Ja bedeutet, dass die Situation korrekt erkannt wurde und Nein, das das Problem nicht richtig kategorisiert wurde. In den Fällen in denen Teile des Programmcodes angegeben sind sei R_0 die Main-Routine. c_i^j sei ein Channel mit Kapazität j, m_i ein Mutex und m_i^r ein RWMutex.

Programm	**Type**	**SubType**	**SubsubType**	**Ok?**
Cockroach 584	Blocking	Resource Deadlock	Doppeltes Locking	Ja
moby 17176	Blocking	Resource Deadlock	Doppeltes Locking	Ja
moby 36114	Blocking	Resource Deadlock	Doppeltes Locking	Ja
etcd 5509	Blocking	Resource Deadlock	Doppeltes Locking	Ja
etcd 6708	Blocking	Resource Deadlock	Doppeltes Locking	Ja
moby 7559	Blocking	Resource Deadlock	Doppeltes Locking	Ja
serving 4829	Blocking	Resource Deadlock	Doppeltes Locking	Ja
Cockroach 9935	Blocking	Resource Deadlock	Doppeltes Locking	Ja
moby 4951	Blocking	Resource Deadlock	Zyklisches Locking	Ja
Cockroach 7504	Blocking	Resource Deadlock	Zyklisches Locking	Ja
Cockroach 10214	Blocking	Resource Deadlock	Zyklisches Locking	Ja
hugo 3251	Blocking	Resource Deadlock	Zyklisches Locking	Ja
kubernetes 13135	Blocking	Resource Deadlock	Zyklisches Locking	Ja
Cockroach 6181	Blocking	Resource Deadlock	RWR Deadlock	Ja
kubernetes 58107	Blocking	Resource Deadlock	RWR Deadlock	Nein
Cockroach 24808	Blocking	Communication Deadlock	Channel	Ja
Cockroach 25456	Blocking	Communication Deadlock	Channel	Ja
Cockroach 35073	Blocking	Communication Deadlock	Channel	Ja
Cockroach 35931	Blocking	Communication Deadlock	Channel	Ja
etcd 6857	Blocking	Communication Deadlock	Channel	Ja
kubernetes 38669	Blocking	Communication Deadlock	Channel	Ja
kubernetes 5316	Blocking	Communication Deadlock	Channel	Nein
kubernetes 70277	Blocking	Communication Deadlock	Channel	Ja
moby 4395	Blocking	Communication Deadlock	Channel	Ja
moby 29733	Blocking	Communication Deadlock	Konditionelle Variable	Ja
moby 30408	Blocking	Communication Deadlock	Konditionelle Variable	Ja
etcd 6873	Blocking	Mixed Deadlock	Channel&Mutex	Ja

etcd 7492	Blocking	Mixed Deadlock	Channel&Mutex	Nein
etcd 7902	Blocking	Mixed Deadlock	Channel&Mutex	Ja
istio 16224	Blocking	Mixed Deadlock	Channel&Mutex	Ja
kubernetes 10182	Blocking	Mixed Deadlock	Channel&Mutex	Ja
kubernetes 1321	Blocking	Mixed Deadlock	Channel&Mutex	Nein
kubernetes 26980	Blocking	Mixed Deadlock	Channel&Mutex	Ja
kubernetes 6632	Blocking	Mixed Deadlock	Channel&Mutex	Ja
serving 2137	Blocking	Mixed Deadlock	Channel&Mutex	Ja
moby 4951	Blocking	Resource Deadlock	AB-BA Deadlock	Ja
moby 7559	Blocking	Resource Deadlock	Double Locking	Ja
serving 5865	NonBlocking	Go-Specific	Misuse channel	Ja
serving 2137	Blocking	Mixed Deadlock	Channel&Lock	Ja
serving 4829	Blocking	Resource Deadlock	Double Locking	Ja

Tab. 3.: Für Auswertung betrachteten Programme aus Gobench [8] sowie Information (Ok?) ob die Situation korrekt erkannt wurde. Ja bedeutet, dass die Situation korrekt erkannt wurde und Nein, das das Problem nicht richtig kategorisiert wurde.

Abbildungsverzeichnis

1. Beispielprogramm für Select 6
2. Beispielprogramm zyklisches Locking 7
3. Beispielprogramm doppeltes Locking 8
4. Beispielprogramm mit hängendem Channel 9
5. Beispielprogramm für nicht gelesenen Nachricht in gepuffertem Channel . . 9
6. Beispielprogramm für Tracer 13
7. Beispielprogramm zyklisches Locking 15
8. Graphische Darstellung des Lock-Graphen für das Beispielprogramm in Abb. 7. Durch die gestrichelten Pfeile wird der enthaltenen Zyklus angezeigt. 16
9. Hängender Channel ohne Deadlock 16
10. Beispielprogramm für unmögliche Synchronisation 16
11. Beispielprogramm für die Betrachtung der Vector-Clocks 18
12. Beispiel für die Order-Enforcement-Instrumentierung eines Select-Statements vor (links) und nach der Implementierung (rechts). Ersetze ..... in dem Programm nach der Instrumentierung (rechts) durch das Programm vor der Instrumentierung (links). [7, gekürzt] 27
13. Instrumentierung der Erzeugung einer neuen Go-Routine. Links: vor der Instrumentierung, rechts: nach der Instrumentierung. 30
14. Instrumentierung eines Select-Statements. Links: vor der Instrumentierung, rechts: nach der Instrumentierung. 31
15. Laufzeit des Instrumenters 33

16. Prozentualer Overhead des Tracers ohne Analyse 33

17. Verteilung der Ergebnisse für Standardprogramme 46
18. Verteilung der Ergebnisse für Standardprogramme mit Mutexen 46
19. Verteilung der Ergebnisse für Standardprogramme mit Channel 46
20. Verteilung der Ergebnisse für Standardprogramme mit Mutexen und Channel 46
21. Verteilung der Ergebnisse für Programme aus GoKer 47
22. Verteilung der Ergebnisse für Programme aus GoKer mit Mutexen 47
23. Verteilung der Ergebnisse für Programme aus GoKer mit Channel 47
24. Verteilung der Ergebnisse für Programme aus GoKer mit Mutexen und Channel 47

Tabellenverzeichnis

1. Laufzeit des Instrumenters für ausgewählte Programme 34

2. Beschreibung der für die Auswertung betrachteten Standardprogramme sowie Information (Ok?) ob die Situation korrekt erkannt wurde. Ja bedeutet, dass die Situation korrekt erkannt wurde und Nein, das das Problem nicht richtig kategorisiert wurde. In den Fällen in denen Teile des Programmcodes angegeben sind sei R_0 die Main-Routine. c_i^j sei ein Channel mit Kapazität j, m_i ein Mutex und m_i^r ein RWMutex. 52

3. Für Auswertung betrachteten Programme aus Gobench [8] sowie Information (Ok?) ob die Situation korrekt erkannt wurde. Ja bedeutet, dass die Situation korrekt erkannt wurde und Nein, das das Problem nicht richtig kategorisiert wurde. 54

ToDo Counters

(TODO: Remove ToDo Counter List)

To Dos: 3; 1, 2, 3

Parts to extend: 0;

Draft parts: 1; 1

Literaturverzeichnis

[1] StackOverflow, "2020 developer survey," 2020. `https://insights.stackoverflow.com/survey/2020`.

[2] A. Gerrand, "Share memory by communicating," 2010. `https://go.dev/blog/codelab-share`.

[3] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," *Proc. FSE 2014*, pp. 155–165, 11 2014.

[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] 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.

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

[7] 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.

[8] 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)*, pp. 187–199, 2021. `https://github.com/timmyyuan/gobench`.

[9] The Go Team, "Effective go," 2022. `https://go.dev/doc/effective_go`.

[10] The Go Team, "The go memory model," 2022. `https://go.dev/ref/mem#chan`.

[11] 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.

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

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

[14] 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.

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

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

[17] The Go Team, "Go-documentation - ast." `https://pkg.go.dev/go/ast`.

[18] P. Mattis, "goid." https://github.com/petermattis/goid.