Bachelorprojekt

im Studiengang
Bachelor of Science in Informatik

Dynamic Deadlock Detection in Go

Erik Daniel Kassubek

15.07.2022

Institut für Informatik
Technische Fakultät
Albert-Ludwigs-Universität Freiburg

Betreuer
Prof. Dr. Thiemann, Albert-Ludwigs-Universität Freiburg
Prof. Dr. Sulzmann, Hochschule Karlsruhe

Abstract

Dieses Projekt betrachtet Detektoren zur Detektion von Ressourcen-Deadlocks in Go. Dazu wird ein bereits implementierten Detektor Go-Deadlock betrachtet und anschließend ein eigener Detektor entwickelt. Der Bericht beschreibt die Funktionsweise und Implementierung beider Detektoren und vergleicht sie im Bezug auf ihre Fähigkeit Deadlocks zu erkennen, bzw. False-Positives zu vermeiden. Dies geschieht sowohl durch die Betrachtung von Standartsituationen als auch durch Anwendung auf tatsächlich verwendete Programme. Außerdem wird der Laufzeit-Overhead der beiden Detektoren betrachtet.

Inhaltsverzeichnis

1 Einführung

Deadlocks sind häufig die Ursache, wenn ein Programm nicht mehr reagiert[1]. Deadlocks entstehen, wenn sich mehrere Routinen in einem nebenläufigen Programm zyklisch blockieren. Locks, welche zu den häufigsten elementaren Methoden zur Synchronisierung in nebenläufigen Programmen gehöre, können leicht zu solchen Situationen führen [2]. Dabei warten mehrere Routinen zyklisch auf die Freigabe eines oder mehrerer Locks, ohne dass die Möglichkeit besteht, dass eines der Locks freigegeben wird. Eine weiter Situation, bei denen es zu Deadlocks kommen kann besteht darin, dass eine Routine ein Lock beansprucht, während es das selbe Lock bereits hält.
Neben solchen Ressourcen-Deadlock gibt es noch weitere Möglichkeiten Deadlocks zu erzeugen, z.B. Kommunikations-Deadlocks, die im folgenden aber nicht betrachtet werden sollen.
Da Deadlocks häufig nur in sehr bestimmten Fällen auftreten, kann es passieren, dass solche Situationen während der Entwicklung eines Programms nicht bemerkt werden. Dies führt dazu, dass in einer Stichprobe in vier open-source Anwendungen (MySql, Apache, Mozilla und OpenOffice) etwa 30% aller Concurrency-Probleme auf Deadlocks zurückzuführen waren [3]. Ein Programm, bzw. eine Implementierung von Locks, die solche Situationen erkennen kann, kann daher sehr hilfreich sein. Auf solche Programme soll im folgenden näher eingegangen werden. Dabei werden dynamische Programme betrachtet, also Programm, die während der Laufzeit des eigentlichen Programs nach Deadlocks suchen. Diese speichern Abhängigkeiten zwischen Lock-Operationen in Lock-Graphen oder Lock-Bäumen, und versuchen aus diesen auf Deadlocks zu schließen.

Nach einem Überblick über die Theorie wird ein vorhandener Detektor [4] betrachtet und seine Funktionsweise analysiert. Anschließend wird ein eigener Detektor für Deadlocks, teilweise basierend auf dem UNDEAD Detector [2] für die Programmiersprache Go entwickelt und implementiert. Die Implementierung des Detectors findet sich in [5]. Der Detektor implementiert sowohl normale Locks, als auch RW-Locks mit Lock, TryLock und Unlock Operationen. Im Anschluss werden die beiden Detektoren analysiert und verglichen.

2 Theoretischer Hintergrund

2.1 Locks, Mutex

Im folgenden werden die Begriffe Lock und Mutex synonym verwendet.
Es gibt Situation in nebenläufigen Programmen, in denen sich verschiedene Routienen nicht gleichzeitig in bestimmten Bereichen aufhalten dürfen.
Locks gehören zu den am weitesten verbreiteten Mechanismen um solche kritische Bereiche in einem Program zu schützen [2]. Möchte ein Routine R_1 nun in einen Bereich eintreten, der durch ein Lock, welches bereits von einer anderen Routine R_0 gehalten wird, geschützt ist, muss sie so lange vor dem Lock warten, bis dieses von R_0 wieder frei gegeben wird.

2.1.1 Operationen

Auf Locks sind in der Regel die drei folgenden Operationen definiert.

- Lock: Eine Routine R versucht das Lock zu beanspruchen. Wird das Lock von keiner Routine gehalten, wird das Lock geschlossen, so dass andere Routinen es nicht schließen können, bis es von R wieder frei gegeben wird. Ist es nicht möglich das Lock zu beanspruchen, da es bereits von einer anderen Routine gehalten wird, muss R so lange von der Operation warten, bis eine Beanspruchung erfolgreich ist.
- TryLock: TryLock unterscheidet sich von Lock dadurch, das die Routine, wenn es nicht möglich ist, das Lock zu beanspruchen nicht vor der Operation wartet, sondern weiter ausgeführt wird, ohne das Lock geschlossen zu haben. Die Operation gibt außerdem zurück, ob die Beanspruchung erfolgreich war.
- Unlock: Unlock gibt ein Lock wieder frei. Es kann nur von der Routine frei gegeben werden, die es geschlossen hat.

2.1.2 RW-Locks

Neben den allgemeinen Locks gibt es noch Reader-Writer Locks. Diese haben im Vergleich zu den allgemeinen Locks zwei Lock und TryLock Operation, R-Lock und W-Lock (analog für Try-Lock). W-Lock, im folgenden einfach Lock genant funktioniert identisch zu Lock in den allgemeinen Locks. Bei R-Locks hingegen ist es, anders als bei Lock, dass das Lock gleichzeitig von verschiedenen Routine gehalten wird (oder mehrfach von der selben Routine), solange alle diese Beanspruchungen durch R-Lock erfolgt sind. Es ist nicht möglich ein Lock gleichzeitig durch R-Lock und Lock zu schließen.

2.2 Deadlock

Ein Deadlock ist ein Zustand in einem nebenläufigen Programm, also einem Programm, in denen mehrere Routinen gleichzeitig ausgeführt werden, indem alle laufenden Routinen zyklisch auf die Freigabe von Ressourcen warten.
Im großen und ganzen lassen sich Deadlocks in zwei Gruppen einteilen: Ressourcen-Deadlocks und Kommunikations-Deadlocks [2]. Im folgenden sollen jedoch nur durch Locks erzeugte Ressourcen-Deadlocks betrachtet werden.
Ein Deadlock kann nun entstehen, wenn all Treads vor einem solchen Lock warten müssen, wobei die Locks immer von einem anderen Tread gehalten werden. Im folgende bezeichnet $acq_i(l)$,

dass das Lock l von der Routine R_i beansprucht und $rel_i(l)$, dass l von R_i wieder freigegeben wird. Man betrachte nun das folgende Beispiel [6] mit den Routinen R_1 und R_2:

	R_1	R_2
1.	$acq_1(y)$	
2.	$acq_1(x)$	
3.	$rel_1(x)$	
4.	$rel_1(y)$	
5.		$acq_2(x)$
6.		$acq_2(y)$
7.		$rel_2(y)$
8.		$rel_2(x)$

Da R_1 und R_2 gleichzeitig ablaufen ist folgender Ablauf möglich:

	R_1	R_2
1.	$acq_1(y)$	
5.		$acq_2(x)$
2.	$B - acq_1(x)$	
6.		$B - acq_2(y)$

Dabei impliziert $B - acq_i(l)$, dass $acq_i(l)$ nicht ausgeführt werden konnte, bzw. dass die Routine R_i vor dem Lock halten muss, da das Lock bereits von einer anderen Routine beansprucht wird. In diesem Beispiel wartet nun R_1 darauf, dass das Lock x geöffnet wird und R_2 wartet darauf, dass Lock y geöffnet wird. Da nun alle Routinen warten müssen, bis ein Lock freigegeben wird, allerdings keine der Routinen weiter laufen kann um ein Lock zu öffnen, kommt es zum Stillstand. Dieser Zustand wird als zyklicher Deadlock bezeichnet.
Eine andere Situation, bei der ein Deadlock entstehen kann tritt auf, wenn eine Routine das selbe Lock mehrfach beansprucht, ohne es zwischendurch wieder frei zu geben. Dies ist sogar möglich, wenn in dem Programm zu diesem Zieitpukt nur eine Routine aktiv ist. Ein Beispiel dafür gibt folgendes Programm:

	R_1
1.	$acq_1(x)$
2.	$acq_1(x)$
3.	$rel_1 x$

Die Routine muss dabei vor der zweiten Beanspruchung warten, ohne dass es es die Möglichkeit gibt, dass die Routine irgendwann weiter laufen wird. Solch ein Deadlock wird als doppeltes Locking bezeichnet.

2.3 Deadlocks-Detection

Deadlocks in Programmen sind Fehler, die oft den vollständige Abbruch eines Programmes zu Folge haben, wenn keine zusätzliche Logik zur Erkennung und Auflösung von Deadlocks implementiert ist. Aus diesem Grund möchte man bereits bei der Implementierung eines Programmes verhindern, dass ein solcher Deadlock auftreten kann. Unglücklicherweise kann es ohne zusätzliche Hilfsmittel schwierig sein, einen solchen Deadlock zu erkennen, da das Auftreten eines Deadlock von dem genauen zeitlichen Ablauf der verschiedenen Routinen abhängt.
Um dennoch Deadlocks erkennen zu können, können Lock-Graphen oder Lock-Bäume verwendet werden, die im folgenden betrachtet werden sollen.

2.3.1 Lock-Graphen

Ein Lock-Graph ist ein gerichteter Graph $G = (L, E)$. Dabei ist L die Menge aller Locks. $E \subseteq L \times L$ ist definiert als $(l_1, l_2) \in E$ genau dann wenn es eine Routine R gibt, auf welche $acq(l_2)$ ausgeführt wird, während sie bereits das Lock l_1 hält [7]. Mathematisch ausgedrückt gilt also

$$(l_1, l_2) \in E \Leftrightarrow \exists t_1, t_3 \ \nexists t_2 \ ((t_1 < t_2 < t_3) \wedge aqr(l_1)[t_1] \wedge rel(l_1)[t_2] \wedge aqr(l_2)[t_3])$$

wobei $aqr(l_1)[t_i]$ bedeutet, dass $aqr(l_1)$ zum Zeitpunkt t_i ausgeführt wird und equivalent für $rel(l_1)[t_i]$.
Ein Deadlock kann nun auftreten wenn es innerhalb dieses Graphen einen Kreis gibt. Um zu verhindern, dass ein False-Positive dadurch ausgelöst wird, dass alle Kanten in einem solchen Kreis aus der selben Routine kommen (wodurch kein Deadlock entstehen kann), können die Kanten noch zusätzlich mit einem Label versehen werden, welches die Routine identifiziert, durch welche die Kante in den Graphen eingefügt wurde. Bei dem Testen nach Zyklen muss nun beachtet werden, dass nicht alle Kanten in dem Kreis das selbe Label haben [7]. Es sei zusätzlich noch gesagt, dass ein Zyklus in einem Lockgraphen nicht immer auf ein potentielles Deadlock hinweist. Ein solcher Fall, bei dem die Detektion von Zyklen zu einer fälschlichen Detektion führt sind sogenannt Gate-Locks. Diese treten z.B. in folgender Situation auf:

	R_1	R_2
1.	$acq_1(z)$	
2.	$acq_1(y)$	
3.	$acq_1(x)$	
4.	$rel_1(x)$	
5.	$rel_1(y)$	
6.	$rel_1(z)$	
7.		$acq_2(z)$
8.		$acq_2(x)$
9.		$acq_2(y)$
10.		$rel_2(y)$
11.		$rel_2(x)$
12.		$rel_2(z)$

In diesem Fall bilden die Locks x und y in dem Lockgraphen einen Zyklus. Allerdings verhindert das Lock z, dass in diesem Fall ein tatsächliches Deadlock auftreten kann, da sich immer nur eine der beiden Routinen in dem Bereich mit den Locks x und y aufhalten kann.

2.3.2 Lock-Bäume

Anders als bei Lock-Graphen, speichert bei Lock-Bäumen jede Routine seine eigenen Abhängigkeiten. Dies bedeutet, dass der Lock-Baum b der Routine R genau dann die Kante $x \rightarrow y$ besitzt, wenn in Routine R das Lock y beansprucht wird, während in das Lock x in R bereits gehalten wird.
Im folgenden, wird ein Lock-Baum als eine Menge von Dependencies betrachtet. Eine solche Dependency besteht aus einem Lock mu und einer Menge von Locks hs, von denen mu anhängt, für die es also eine Kante $x \rightarrow mu$ mit $x \in hs$ gibt.
Ein potentielles Deadlock liegt nun vor, wenn es in der Menge aller Dependencies eine gültige, zyklische Kette gibt.

Ein Pfad mit n Elementen ist eine gültige Kette, wenn die folgenden Eigenschaften gelten:

$$\forall i, j \in \{1, ..., n\} : \neg(i = j) \rightarrow \neg(dep_i = dep_j) \tag{2.3.2.a}$$
$$\forall i \in \{1, ..., n\} : mu_i \in hs_{i+1} \tag{2.3.2.b}$$
$$\forall i \in \{1, ..., n\} : read(mu_i) \rightarrow (\forall\ mu \in hs_{i+1} : (mu = mu_i) \rightarrow \neg read(mu)) \tag{2.3.2.c}$$
$$\forall i, j \in \{1, ..., n\} : \neg(i = j) \rightarrow (\exists\ l1 \in hs_i\ \exists\ l2 \in hs_j((l1 = l2) \rightarrow (read(l1) \wedge read(l2)))) \tag{2.3.2.d}$$

Dabei bezeichnet mu_i den Mutex und hs_i das holdingSet der i-ten Dependency in der Kette. $read(mu)$ bedeutet, dass das Locking von mu, welches in der Dependency abgebildet ist durch ein R-Lock zustande gekommen ist.
Formel 2.3.2.a stellt sicher, dass die selbe Dependency nicht mehr als ein mal in der Kette auftauchen kann. Formel 2.3.2.b besagt, dass die Dependencies tatsächlich eine Kette bilden müssen, dass also der Mutex einer Dependency immer in dem HoldingSet der nächsten Dependency in dem Pfad enthalten sein muss. Auch wenn dies wahr ist, 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. Dass solche Pfade ausgeschlossen werden wird durch Formel 2.3.2.c sichergestellt. Die letzte Formel 2.3.2.d beschäftigt sich mit Gate-Locks. Sie besagt, dass wenn es einen Mutex gibt, der in den HoldingSets zweier verschiedener Dependencies in dem Pfad vor kommt, 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. Wenn diese vier Formeln erfüllt sind, handelt es sich um eine gültige Kette.
Ein potentielles Deadlock ergibt sich nun, wenn diese Kette einen Kreis bildet, wenn als

$$mu_n \in hs_1 \tag{4.3.e}$$
$$read(mu_n) \rightarrow (\forall\ mu \in hs_1 : (mu = mu_n) \rightarrow \neg read(mu)) \tag{4.3.f}$$

gilt.

Diese Bäume haben mehrere Vorteile gegenüber Lock-Graphen. Da jede Routine seine eigene Datenstruktur hat, wird verhindert, dass es durch gleichzeitigen Zugriff verschiedener Routinen auf die selbe Datenstruktur zu Problemen kommet. Zudem muss die Zugehörigkeit der Abhängigkeiten zu verschiedenen Routinen nicht explizit gespeichert werden. Ein weiterer Vorteil besteht darin, dass Gate-Locks bei Lock-Bäumen, anders als bei Lock-Graphen nicht zu False-Positives führen können.

2.3.3 Doppeltes Locking

Um doppeltes Locking zu erkennen ist es ausreichend vor der Lock-Operation zu überprüfen, ob das Lock bereits von der selben Routine gehalten wird, ohne dass es sich bei beiden Operationen um Reader-Locks handelt.

3 Review bestehender Tools

3.1 go-deadlock

Im folgenden soll eine Software zur Erkennung von Deadlocks analysiert werden. Dazu wird die Software "sasha-s/go-deadlock" [4], veröffentlicht auf Github, betrachtet.
Dieses verwendet Drop-In Replacements für die, in go standardmäßig implementierten, sync.Mutex Locks. Diese führen sowohl das eigentliche Locking aus und können, beim Durchlaufen des Programms Situationen erkennen, die zu einem Deadlock führen können. Dabei werden sowohl ein allgemeines Lock als auch Readers–writer-locks implementiert. Dies wirkt sich allerdings nur auf die Anwendung des eigentlichen Lockings aus, nicht aber auf die Detection von Deadlocks. Aus diesem Grund wird hierauf im folgenden nicht weiter eingegangen, und die Methoden für die Detection von Deadlocks bezieht sich sowohl auf die allgemeinen als auch die Readers–writer Locks.
Für die Erkennung werden drei verschiedene Fälle betrachtet:

- Rekursives Locking
- Zyklisches Locking
- Timeout

Diese sollen im folgenden genauer betrachtet werden.

3.1.1 Rekursives Locking

Um rekursives Locking zu erkennen speichert das Programm ein Dictionary *cur* mit allen momentan gehaltenen Locks. Die Werte zu den jeweiligen Keys speichern sowohl, von welcher Go-Routine das Lock momentan gehalten wird, als auch in welcher Datei und Zeile der Befehl, der das Locking dieses Locks zu folge hatte, zu finden ist. Wird ein Lock wieder freigegeben, so wird der entsprechende Eintrag aus *cur* entfernt. Wird nun ein Lock neu beansprucht, überprüft das Programm, ob dieses Lock mit der Routine, die das Lock beanspruchen möchte bereits in *curl* auftaucht. Ist dies der Fall, dann nimmt das Programm an, dass es sich hierbei um ein mögliches Deadlock durch rekursives Locking handelt und führt entsprechende Schritte aus, um den Nutzer zu warnen. Wird ein Lock wieder frei gegeben, wird der entsprechende Eintrag aus *cur* entfernt.

3.1.2 Zyklisches Locking

Mit dieser Methode werden mögliche Deadlocks gefunden, die dadurch entstehen, dass alle Threads zyklisch auf die Freigabe eines Locks warten, welches von einem anderen Thread gehalten wird.
Der Detektor arbeitet dabei mit einem Lock-Graphen.
Für die Detektion verwendet das Programm zwei Dictionaries. Das erste ist *cur*, welches bereits in 3.1.1 betrachten wurde.
Das andere Dictionary *order* definiert mit seinen Keys die Kanten des Lock-Graphen. Die Keys bestehen dabei aus einer Struktur $beforeAfter$, die Referenzen zu den beiden Locks speicherte, welche von der Kante im Graphen verbunden werden. Wird ein neues Lock p beansprucht, so wird für jedes Lock b, welches sich momentan in *cur* befindet ein neuer Eintrag $beforeAfter\{b, p\}$ in *order* hinzugefügt. Die Werte, die für jeden Key in *order* gespeichert werden, entsprechen den Information, die auch in *cur* für die beiden Locks gespeichert wird. Allerdings wird auf die

Speicherung der ID der erzeugenden Go-Routine verzichtet, da sie nicht benötigt wird. Information aus *order* werden nur entfernt, wenn *order* eine festgelegte maximale Größe überschreitet. Die Überprüfung, ob ein Lock p zu einem Deadlock führen kann, finden bereits statt, bevor das Lock in *cur* und *order* eingetragen wird. Dazu wird für jedes Lock b in *cur* überprüft, ob *order* einen Key $beforeAfter\{p, b\}$ besitzt, der Graph also die beiden Locks in umgekehrter Reihenfolge enthält. Existiert solch ein Key, und wurde b von einer anderen Routine als p in *cur* eingefügt, bedeutet dies einen Loop aus zwei Kanten im Lockgraphen und somit einen möglichen Deadlock.
Dies bedeutet aber auch, dass das Programm nicht in der Lage ist, ein Kreis in einem Lock-Graphen zu finden, wenn dieser aus drei oder mehr Kanten besteht. Soche Situationen können aber dennoch zu Deadlocks führen. Ein Beispiel dafür ist die folgende Funktion:

```
1  func threeEdgeLoop() {
2      var x Mutex
3      var y Mutex
4      var z Mutex
5      ch := make(chan bool, 3)
6
7      go func() {
8          // first routine
9          x.Lock()
10         y.Lock()
11         y.Unlock()
12         x.Unlock()
13         ch <- true
14     }()
15
16     go func() {
17         // second routine
18         y.Lock()
19         z.Lock()
20         z.Unlock()
21         y.Unlock()
22         ch <- true
23     }()
24
25     go func() {
26         // third routine
27         z.Lock()
28         x.Lock()
29         x.Unlock()
30         z.Unlock()
31         ch <- true
32     }()
33
34     <-ch
35     <-ch
36     <-ch
37 }
```

Führen die drei Routinen jeweils ihre erste Zeile gleichzeitig aus, muss jede Routine vor ihrer zweiten Zeile warten und es kommt zu einem Deadlock. Da diese Konstellation in einem Lockgraphen aber zu einem Kreis mit einer Länge von drei Kanten führen würde, kann das Programm den möglichen Deadlock nicht erkennen.

3.1.3 Timeout

Neben diesen beiden Methoden, die vorausschauend nach möglichen Deadlocks Ausschau halten, versucht das Programm auch mit Timeouts um zu überprüfen ob sich das Programm bereits in einem Deadlock befindet. Möchte eine Routine ein Lock beanspruchen wird vorher eine go routine mit einem Counter gestartet. Sollte die Inanspruchnahme des Locks innerhalb der vorgegebenen Zeit (default: 30s) gelingen, wird die go routine beendet. Sollte es nicht gelingen, nimmt das Programm nach der festgelegten Zeit an, dass es zu einem Deadlock gekommen ist und gibt eine entsprechende Nachricht aus.
Diese Methode kann durchaus nützlich sein, um über Deadlocks informiert zu werden. Allerdings führt sie sehr leicht zu false-positives, wenn die Abarbeitung anderer Routinen und damit die Freigabe des Locks länger Dauer, als die festgelegte Timeout Zeit. Im folgenden Beispiel wird dies deutlich:

```
1  func falsePositive() {
2      var x deadlock.Mutex
3      finished := make(chan bool)
4
5      go func() {
6          // first go routine
7          x.Lock()
8          time.Sleep(40 * time.Second)
9          x.Unlock()
10     }()
11
12     go func() {
13         // second go routine
14         time.Sleep(2 * time.Second)
15         x.Lock()
16         x.Unlock()
17         finished <- true
18     }()
19
20     <-finished
21 }
```

Der Chanel finished wird lediglich verwendet um zu verhindern, dass das Programm beendet wird, bevor die gol Routinen durchlaufen wurden. Er ist für die Deadlock Analyse also irrelevant. Das Programm startet zwei go-Routinen, die beide das selbe Lock x verwenden. Durch den time.Sleep(2 * time.Second) command wird sichergestellt, dass die erste go Routine zuerst auf das Lock zugreift. Dass Lock in Routine 2 muss also warten, bis es in Routine 1 wieder freigegeben wird. Dies geschieht in etwa $38s$ nachdem die zweite Routine mit dem warten auf die Freigabe von x beginnt. Da dies länger ist als die standardmäßig festgelegte maximale Wartezeit von $30s$ nimmt das Programm an, es sei in einen Deadlock gekommen, obwohl kein Solcher vorliegt, und auch kein Deadlock möglich ist.

3.1.4 False Negatives / Positives

Bei False Negatives oder False Positives handelt es sich um Fälle, bei denen es zu einem Deadlock kommen kann, ohne dass dieser Detektiert wird, bzw. Fälle die nicht zu einem Deadlock führen können, aber dennoch als Deadlock angezeigt werden. Neben den in den vorherigen Abschnitten gezeigten Fällen, kann es auch in weiteren Fällen zu solchem Verhalten kommen.
Neben den oben bereits genannten Fällen, kann dies noch in weiteren Situationen auftreten. Ein Fall, bei dem ein Deadlock auftreten könnte, welcher von dem Programm aber nicht erkannt wird (False Negative), entsteht bei verschachtelten Routinen. Dabei erzeugt eine go-Routine eine weitere.

```
1  func nestedRoutines() {
2      x := deadlock.NewLock()
3      y := deadlock.NewLock()
4      ch := make(chan bool)
5      ch2 := make(chan bool)
6
7      go func() {
8          x.Lock()
9          go func() {
10             y.Lock()
11             y.Unlock()
12             ch <- true
13         }()
14         <-ch
15         x.Unlock()
16         ch2 <- true
17     }()
18     go func() {
19         y.Lock()
20         x.Lock()
21         x.Unlock()
22         y.Unlock()
23         ch2 <- true
24     }()
25
26     <-ch2
27     <-ch2
28 }
```

Diese Funktion ist bezüglich ihres Ablaufs identisch zu der Funktion circularLocking in Kapitel 3.1.2. Dennoch ist es dem Programm aufgrund der Verschachtelten go-Routine nicht möglich, den möglichen Deadlock zu erkennen.
Ein Fall, bei dem es zu False Positives kommen kann, wird durch Gate-Locks ausgelöst. Wie bereits beschrieben ist ein auf Lock-Graphen basierender Detektor, und damit solch ein Detektor nicht in der Lage zu erkennen, wenn ein potentielles Deadlock durch Gate-Locks unmöglich gemacht wird.

3.2 Go Runtime Deadlock Detection

Nebenbei sei noch erwähnt, dass Go einen eigenen Detektor zur Erkennung von Deadlocks besitzt. Allerdings kann dieser, im Unterschied zu dem in 3.1 betrachteten Detector nur tatsächlich auftretende Deadlocks erkennen. Eine Erkennung ob in dem Code ein Deadlock möglich ist findet hierbei nicht statt. Um zu erkennen, ob ein Deadlock vorliegt zählt go die nicht blockierten Go-Routinen. Fällt dieser Wert auf 0, nimmt Go an, dass es zu einem Deadlock gekommen ist und bricht das Programm mit einer Fehlermeldung ab [8].

4 Implementierung des Deadlock-Detektors „Deadlock-Go “

Im folgenden soll die Funktionsweise und Implementierung von Deadlock-Go betrachtet werden. Dieser wurde entwickelt um Ressourcen-Deadlocks in Go-Programmen zu erkennen und den Nutzer vor solchen zu warnen. Der Code kann in [5] gefunden werden.
Der Detektor basiert zum Teil auf dem UNDEAD-Algorithmus [2]. Zusätzlich wurde er um RW-Locks sowie eine Detektion von doppeltem Locking erweitert, welche in UNDEAD nicht betrachtet werden.
Der Detektor implementiert dabei Lock-Bäume, mit welchen zyklisches Locking erkannt werden kann. Neben den Informationen, die für die Erkennung solcher Zyklen benötigt werden, werden außerdem Information darüber, wo in dem Programm- Code die Initialisierung sowie die (Try-)(R-)Lock Operationen aufgerufen werden gespeichert. Diese werden dem Nutzer bereitgestellt, wenn ein potentielles Deadlock erkannt wurde, um das Auffinden und Korrigieren des entsprechenden Codes zu erleichtern.
Das folgende Program ist ein Beispiel zur Verwendung des Detektors:

```
1  import "github.com/ErikKassubek/Deadlock-Go"
2
3  func main() {
4      defer deadlock.FindPotentialDeadlocks()
5      x := deadlock.NewLock()
6      y := deadlock.NewLock()
7
8      // make sure, that program does not terminates
9      // before all routines have terminated
10     ch := make(chan bool, 2)
11
12     go func() {
13         x.Lock()
14         y.Lock()
15         y.Unlock()
16         x.Unlock()
17         ch <- true
18     }()
19
20     go func() {
21         y.Lock()
22         x.Lock()
23         x.Unlock()
24         y.Unlock()
25         ch <- true
26     }()
27     <-ch
28     <-ch
29 }
```

Es erzeugt den folgenden Output:

```
POTENTIAL DEADLOCK

Initialization of locks involved in potential deadlock:

/home/ * * * /undead_test.go 5
/home/ * * * /undead_test.go 6

Calls of locks involved in potential deadlock:

Calls for locks created at /home/ * * * /undead_test.go 5
/home/ * * * /undead_test.go 22
/home/ * * * /undead_test.go 13

Calls for locks created at /home/ * * * /undead_test.go 6
/home/ * * * /undead_test.go 21
/home/ * * * /undead_test.go 14
```

In diesem, sowie in allen folgenden Beispielen, wurden die Pfade durch * * * gekürzt. In der tatsächlichen Ausgabe wird der vollständige Pfad angegeben.
Der genaue Output hängt von dem tatsächlichen Ablaufs ab, der aufgrund der Nebenläufigkeit bei verschiedenen Durchläufen unterschiedlich sein kann.

Das restliche Kapitel ist in die folgenden Abschnitte eingeteilt:

- Aufbau der Datenstrukturen
- (R-)Lock, (R-)TryLock und (R-)Unlock
- Periodische Detection
- Abschließende Detektion
- Meldung gefundener Deadlocks
- Optionen

4.1 Aufbau der Datenstrukturen

Im folgenden sollen die in dem Detektor verwendeten Datenstrukturen betrachtet werden. Wie diese genau verwendet werde, wird in späteren Abschnitten noch genauer betrachtet.

4.1.1 (RW-)Mutex

Die Strukturen für die (RW-)Mutexe sind so implementiert, dass sie als Drop-In-Replacements für die klassischen sync.(RW)Mutex verwendet werden. Dabei gibt es eine Struktur für Mutexe und eine für RW-Mutexe, die über ein zusätzliches Interface MutexInt zusammengefasst werden können.
Die Mutex Strukturen beinhalten alle Informationen, die für diese benötigt werden. Dabei handelt es sich um die folgenden Informationen:

- mu (*sync.(RW)Mutex) ist das eigentliche Lock, welches für das tatsächliche Locking verwendet wird. Für ein Mutex ist dies ein *sync.Mutex, für ein RW-Mutex ein *sync.RWMutex.
- context ([]callerInfo) ist die Liste der callerInfo des Mutex. CallerInfo sind die Informationen darüber wo in dem Programm-Code die Initialisierung und die (Try-)(R-)Lock Operationen stattgefunden haben. Dazu werden die Datei und Zeilennummer gespeichert. Je nachdem,

wie das Program über die Optionen konfiguriert ist, kann auch ein vollständiger Call-Stack für die entsprechenden Operationen gespeichert werden.
- in (bool) ist ein Marker welcher speichert, ob der Mutex richtig initialisiert wurde. Es ist nicht mögliche ein Mutex einfach durch $varxMutex$ oder $varxRWMutex$ zu erzeugen und sofort zu verwenden, sondern die Mutex Variable muss über $x := NewLock()$ bzw. $x := NewRWLock()$ initialisiert werden. Wird eine Lock-Operation o.ä. auf einem (RW-)Mutex ausgeführt, ohne dass dieses entsprechend initialisiert wurde, wird das Programm mit einem Fehler abgebrochen.
- numberLocked (int) speichert, wie oft das Lock im moment gleichzeitig beansprucht ist. Ist das Lock frei, ist dieser Wert 0. Bei einem Mutex kann der Wert maximal 1 werden, wenn das Lock gerade von einer Routine gehalten wird. Das selbe ist wahr für RW-Mutexe, wenn diese über ein Writer-Lock gehalten werden. Werden diese allerdings von Reader-Locks gehalten, kann das selbe Lock von mehreren Locks gehalten werden.
- isLockedRoutineIndex (*map[int]int) speichert die Indices der Routinen, von welchen das Lock momentan gehalten wird, sowie wie oft es gehalten wird. Dies wird für die Erkennung von doppeltem Locking verwendet.
- memoryPosition (uintptr) speichert die Speicheradresse des Mutex Objekts bei seiner Erzeugung.

4.1.2 Dependencies

Dependencies werden verwendet, um die Abhängigkeiten der verschiedenen Mutexe untereinander zu speichern. Sie entsprechen dabei einer Menge an Kanten in einem Lock Graphen. Sie enthalten die folgenden Informationen:
- mu (mutexInt) ist das Lock, für welches gespeichert werden soll, von welchen anderen Locks mu abhängt, welche Locks also in der selben Routine bereits gehalten wurden, als mu erfolgreich beansprucht wurde.
- holdingSet ([]mutexInt) ist die Liste der Locks, von denen mu abhängt.
- holdingCount (int) ist die Anzahl der Locks, von denen mu abhängt, also die Anzahl der Elemente in holdingSet.

4.1.3 Routine

Für jede Routine wird automatisch eine Struktur angelegt, die alle für sie relevanten Informationen speichert. Diese Strukturen werden in einem globalen Slice (Liste) routines gespeichert. Bei den Informationen, die von diesen Routinen gehalten werden handelt es sich vor allem um eine Liste der Mutexe, die momentan von der Routine gehalten werden, sowie die Inforationen, die für den Aufbau des Lock-Baums benötigt werden. Es handelt sich dabei um die folgenden Informationen:
- index (int) ist der Index der Routine in routines
- holdingCount (int) speichert die Anzahl der im Moment gehaltenen Locks
- holdingSet ([]MutexInt) ist eine Liste, welche Referenzen zu den momentan von der Routine gehaltenen Locks.
- dependencyMap (map[uintptr]*[]*dependency) ist ein Dictionary, welches verwendet wird, um zu verhindern, dass wenn die selbe Situation im Code mehrfach auftritt (z.B. durch Schleifen) die entsprechenden Dependencies mehrfach in dem Lock-Baum gespeichert werden.
- dependencies ([]*dependencies) speichert die aufgetretenen Dependencies. Dies stellt also die Implementierung des Lock-Baum da.
- curDep (*dependency) ist die letzte in den Lock-Baum (dependencies) eingefügte dependency.
- depCount (int) gibt die Anzahl der in dem Lock-Baum gespeicherten Dependencies an.

- collectSingleLevelLocks speichert Informationen über single-level Locks, also Locks, welche von keinem andere Lock abhängen. Auch dies ist wie dependencyMap dazu da, um zu verhindern verhinder, dass Informationen, die bereits bekannt sind mehrfach gespeichert werden.

4.2 (R)-Lock, (R)-TryLock, (R)-Unlock

Bei jeder Operation, welche auf den Locks ausgeführt wird, werden die Datenstrukturen, wie sie in 4.1 beschrieben wurden verändert oder neu erzeugt, um auf diesen eine Detektion von Deadlocks zu ermöglichen.
Diese Vorgänge sollen im folgenden beschrieben werden.

4.2.1 (R)-Lock

Das Locking von RW-Mutexes unterscheidet sich von dem Locking der Mutexe nur dadurch, das isRead in dem Mutex entsprechend gesetzt wird, und das das Locking des eigentlichen sync.(RW)Locks entsprechend angepasst wird. Die Sammlung der Informationen, welche für die Detektion der Deadlocks benötigt wird ist die selbe.

Zuerst wird überprüft, ob das Lock sowie der Detektor bereits initialisiert wurde. Ist der Mutex nicht über $NewLock()$ initialisiert worden, wird das Programm mit einer Fehlermeldung abgebrochen.
Anschließend wird das eigentliche Locking des sync.Mutex über ein defer statement vorbereitet. Dabei wird darauf geachtet, ob es sich um ein RWMutex oder ein Mutex handelt, und wenn es sich um din RWMutex handelt, ob es eine R-Lock operation ist.
Sind die periodische und die abschließende Detektion in den Optionen deaktiviert, wird ist die Lock Operation an diesem Punkt beendet. Ist mindestens eine von ihnen aktiviert, wird nun der Lock-Baum der entsprechenden Routine aktualisiert.
Zuerst wird überprüft, ob für für die entsprechende Routine bereits ein Lock-Baum existiert. Ist dies nicht der Fall wird ein leere Baum erzeugt und mit dieser Routine verknüpft.
Solange die Detektion von doppeltem Locking nicht deaktiviert ist, wird nun überprüft, ob das Beanspruchen dieses Locks zu einem Deadlock führen würde. Dazu wird zuallererst überprüft, ob der Mutex momentan bereits von einer Routine gehalten wird. Ist dies nicht der Fall kann es nicht zu doppeltem Locking kommen. Wird es bereits gehalten, muss es dennoch nicht zu einem doppelten Locking kommen. Dies ist der Fall, wenn die Routine, die das Mutex hält, und die die es momentan beansprucht nicht die selben sind, oder wenn sie die selben sind wenn beide R-Locks sind. Ist nichts davon der Fall, nimmt das Programm an, es sei ein Deadlock gefunden worden. In diesem Fall wird eine Beschreibung des Deadlocks ausgegeben (vgl. 4.5). Anschließend wird die abschließende Detektion gestartet und das Programm anschließend abgebrochen.
Im Anschluss werden nun die entsprechenden Datenstrukturen aktualisiert. Dies geschieht allerdings nur, wenn momentan mindestens eine Routine läuft, da Deadlocks, abgesehen von doppeltem Locking, nur bei der gleichzeitigen Ausführung mehrerer Routinen auftreten. Man betrachte zu erst den Fall des Single-Level-Locks. Dabei handelt es sich um einen Mutex, der zu einem Zeitpunkt beansprucht wird an dem die selbe Routine keine anderen Locks hält. Da sich somit keine Dependencies bilden, muss der Lock-Baum der Routine nicht verändert werden. In diesem Fall wird lediglich die Information über den Aufruf des Lockings in “context” gespeichert, solange der Aufruf der exakt selben Beanspruchung (selbe Datei und selbe Zeilennummer) noch nicht gespeichert wurde. Hält die Routine allerdings bereits ein oder mehrere Lock, wird die entsprechende Dependency in den Lock-Baum eingefügt, solange sie in dieser noch nicht existiert. Diese Überprüfung wird mit Hilfe von “dependencyMap” ausgeführt. Existiert sie noch nicht, wird sie in den Lock-Baum und in dependencyMap eingefügt. Für diese Dependency entspricht “mu”

gerade dem zu lockenden Mutex und “holdingSet” dem momentanen “holdingSet” der Routine, also der Liste aller Mutexe, die von der Routine im Moment gehalten werden.
Sowohl bei Single-Level-Locks als auch bei Locks, welche zu Dependencies führen wird das Lock im Anschluss in das “holdingSet” der Routine eingefügt.

4.2.2 (R)-TryLock

Bei einer Try-Lock Operation wird ein Lock nur beansprucht, wenn es im Moment der Operation beansprucht werden kann, das Lock also nicht bereits von einer Routine gehalten wird. Aus diesem Grund kann die Beanspruchung des Locks nicht direkt zu einem Deadlock führen. Es ist also nicht notwendig nach doppeltem Locking zu suchen, oder den Lock-Baum zu aktualisieren, wenn das Lock beansprucht wird. Ein solches Locking kann nur zu einem Deadlock führen, wenn es bereits durch die (R)-TryLock-Operation gehalten wird und von einer anderen Operation ebenfalls beansprucht werden soll. Es wird also zuerst versucht das Lock zu beanspruchen. Wenn die Beanspruchung erfolgreich war, wird angepasst wie oft das Mutex gelockt ist, das Mutex in das “holdingSet” der Routine eingefügt und anschließend zurück gegeben, ob die (R)-TryLock operation erfolgreich war

4.2.3 (R)-Unlock

Zuerst wird überprüft, ob der Mutex überhaupt gelockt ist. Ist dies nicht der Fall wird das Programm mit einer Fehlermeldung abgebrochen. Andernfalls wird die Anzahl der Lockungen des Locks angepasst, dass Mutex aus dem “holdingSet” entfernt und das Lock wieder frei gegeben.

4.3 Periodische Detektion

Ist sie nicht deaktiviert, so wird die periodische Detektion in regelmäßigen Abständen (default: 2s) gestartet um nach lokalen, tatsächlich auftretenden Deadlocks zu suchen. Lokal bedeutet dabei, dass sich nur ein Teil der Routinen in einem Deadlock befindet. Sollte es zu einem totalen Deadlock kommen, bei dem alle Routinen blockiert werden, wird das Programm automatisch von der Go Runtime Deadlock Detection beendet. In diesem Fall ist keine weiter abschließende Detektion von Deadlocks möglich.

Für die periodische Detektion wird von jeder Routine nur “curDep” betrachtet, also diejenige Routine, welche als letztes in den Lock-Baum eingefügt wurde. Die Detektion wird nur ausgeführt, wenn sich diese Menge seit der letzten periodischen Detektion verändert hat und momentan mindestens zwei Routinen im Moment ein Lock halten. Im diesem Fall wird versucht Zyklen in der Menge der “curDep” zu finden.
Dazu wird eine Depth-First-Search auf diesen Dependencies ausgeführt. Dazu wird zuerst die “curDep” einer der Routinen auf einen Stack gelegt. Der Stack entspricht immer dem momentan betrachteten Pfad. Anschließend wird für die “curDep” jeder Routine, die noch nicht auf dem Stack liegt und noch nicht zuvor bereits betrachtet wurde überprüft, ob das hinzufügen der Dependency zu einer gültigen Kette führt, ob also die Formeln 2.3.2.a - 2.3.2.d immer noch gelten. Ist dies der Fall, so wird überprüft, ob die Kette einen Kreis bildet, also ob die Formeln 4.3.e und 4.3.f ebenfalls gelten. Ist der Pfad mit der neuen Dependency eine gültige Kette aber kein Kreis, so wird die Dependency auf den Stack gelegt und das ganze rekursiv wiederholt. Ist die Kette ein Kreis, so nimmt dass Programm fürs erst an, es sei ein lokaler Deadlock erkannt worden. In diesem Fall wird überprüft, ob sich die HoldingSets der Routinen, von denen sich eine Dependency in der Kette befindet seit dem Begin der momentanen periodische Detektion verändert hat. Ist dies der Fall, so geht der Detektor vorerst davon aus, dass es sich um einen falschen Alarm handelt. Andernfalls wird dem Nutzer mitgeteilt, dass ein Deadlock gefunden

wurde, es wird die abschließende Detektion gestartet und das Programm anschließend abgebrochen. Gibt es keine Dependency die, wenn sie auf den Stack gelegt wird zu einer gültigen Kette führt, so wird die oberste Dependency von dem Stack entfernt, so dass andere mögliche Pfade betrachtet werden können.
Wenn es keinen Pfad gibt, der eine gültige, zyklisches Kette bildet, so geht der Detektor davon aus, dass sich das Program nicht in einem lokalen Deadlock befindet.

4.4 Abschließende Detektion

Die abschließende Detektion wird am Ende des Programs durchgeführt um potentielle Deadlock zu finden, auch wenn diese in dem Durchlauf nicht tatsächlich aufgetreten sind. Sie muss vom Nutzer manuell in seinem Code gestartet werden. Sie wird in diesem Fall nur ausgeführt, wenn sie nicht deaktiviert ist, die Anzahl der Routinen, die in dem Program vorkamen sowie die Anzahl der einzigartigen Dependencies mindestens zwei ist.
Im Großen und Ganzen verläuft die Detektion identisch zu der periodischen Detektion (vgl. 4.3). Allerdings werden nun nicht nur die zuletzt in die Lock-Bäume aufgenommenen Dependencies sondern alle in den Bäumen vorkommenden Dependencies betrachtet. Dabei wird darauf geachtet, dass von jeder Routine maximal eine Dependency in der Kette vorkommen kann. Außerdem wird die Detektion nicht beim ersten Auftreten eines potentiellen Deadlocks beendet, sondern erst, wenn alle möglichen Pfade betrachtet wurde.

4.5 Meldung gefundener Deadlocks

Wird ein tatsächlicher oder potentieller Deadlock gefunden, wird dem Nutzer dies durch eine eine Nachricht über den standard error file descriptor (Stderr) mitgeteilt.

4.5.1 Doppeltes Locking

Tritt ein Fall von doppeltem Locking auf, so wird dem Nutzer das dabei involvierte Lock, sowie seine Aufrufe mitgeteilt. Im folgenden ist ein Beispiel für solch eine Ausgabe gegeben:

```
DEADLOCK (DOUBLE LOCKING)

Initialization of lock involved in deadlock:

/home/ * * * /undead_test.go 238

Calls of lock involved in deadlock:

/home/ * * * /undead_test.go 239
/home/ * * * /undead_test.go 240
```

Für Doppeltes Locking ist es nicht möglich, den Stacktrace für die Lock-Vorgänge anzuzeigen. könnte man des noch einfügen

4.5.2 Deadlocks

Bei einem tatsächlichen oder potentiellen Deadlock werden die in dem Zyklus, welcher den Deadlock verursacht vorkommenden Locks, sowie die Positionen ihrer Lock-Operationen angegeben.

Dazu werden die callerInfo der Mutexe in den Dependencies betrachtet, die in dem Stack, welcher einen Deadlock beschreibt vorkommen.
Dies führt z.B. zu folgender Ausgabe:

```
POTENTIAL DEADLOCK

Initialization of locks involved in potential deadlock:

/home/***/undead_test.go 40
/home/***/undead_test.go 39

Calls of locks involved in potential deadlock:

Calls for lock created at: /home/***/undead_test.go:40
/home/***/undead_test.go 48
/home/***/undead_test.go 60

Calls for lock created at: /home/***/undead_test.go:39
/home/***/undead_test.go 47
/home/***/undead_test.go 61
```

Es ist möglich, sich statt nur der Datei und Zeilennummer auch einen Call-Stack anzeigen zu lassen:

```
POTENTIAL DEADLOCK

Initialization of locks involved in potential deadlock:

/home/***/deadlockGo.go 24
/home/***/deadlockGo.go 25

CallStacks of Locks involved in potential deadlock:

CallStacks for lock created at: /home/***/deadlockGo.go:24

goroutine 21 [running]:
DeadlockExamples/selfWritten.DeadlockGoPotentialDeadlock.func2()
    /home/***/deadlockGo.go:43 +0x59
created by DeadlockExamples/selfWritten.DeadlockGoPotentialDeadlock
    /home/***/deadlockGo.go:41 +0x14a

goroutine 20 [running]:
DeadlockExamples/selfWritten.DeadlockGoPotentialDeadlock.func1()
    /home/***/deadlockGo.go:33 +0x7b
created by DeadlockExamples/selfWritten.DeadlockGoPotentialDeadlock
    /home/***/deadlockGo.go:29 +0xdb

CallStacks for lock created at: /home/***/deadlockGo.go:25

goroutine 21 [running]:
DeadlockExamples/selfWritten.DeadlockGoPotentialDeadlock.func2()
    /home/***/deadlockGo.go:42 +0x45
```

```
created by DeadlockExamples/selfWritten.DeadlockGoPotentialDeadlock
    /home/ * * * /deadlockGo.go:41 +0x14a

goroutine 20 [running]:
DeadlockExamples/selfWritten.DeadlockGoPotentialDeadlock.func1()
    /home/ * * * /deadlockGo.go:34 +0x8f
created by DeadlockExamples/selfWritten.DeadlockGoPotentialDeadlock
    /home/ * * * /deadlockGo.go:29 +0xdb
```

4.6 Optionen

Die Funktionsweise des Detektors kann über verschiedenen Optionen gesteuert werden, die vor der der erste Lock-Operation gesetzt werden müssen. Dies beinhaltet die folgenden Möglichkeiten:

- Aktivierung oder Deaktivierung des gesamten Detektors (Default: aktiviert)
- Aktivierung oder Deaktivierung der periodische Detektion (Default: aktiviert)
- Aktivierung oder Deaktivierung der abschließenden Detektion (Default: aktiviert)
- Festlegung der Zeit zwischen periodischen Detektionen (Default: 2s)
- Aktivierung oder Deaktivierung der Sammlung von Call-Stacks (Default: deaktiviert)
- Aktivierung oder Deaktivierung der Sammlung von Informationen über Single-Level-Locks (Default: aktiviert)
- Aktivierung oder Deaktivierung der Detektion von doppeltem Locking (Default: aktiviert)
- Festlegung der maximalen Anzahl von Dependencies pro Routine (Default: 4096)
- Festlegung der maximalen Anzahl von Mutexe von denen ein Mutex abhängen kann (Default: 128)
- Festlegung der maximalen Anzahl von Routinen (Default: 1024)
- Festlegung der maximalen Länge eines Call-Stacks in Bytes (Default 2048)

5 Analyse von Deadlock-Go und Vergleich mit go-deadlock (sasha-s)

Im Folgenden soll der in 4 beschriebene und in [5] implementierte Detektor „Deadlock-Go “analysiert und mit dem in 3.1 betrachteten Detektor „go-deadlock “verglichen werden. Um Verwechslungen zwischen den beiden relativ ähnlichen Namen zu vermeiden, wird dieser im folgenden nach dem Besitzer des Git-Hub Repositories als sasha-s bezeichnet.
Für den Vergleich werden verschiedene Beispiel-Programme betrachtet, die in [9] implementiert sind. Diese bestehen zum einen aus verschiedenen Standartsituationen, zum anderen auf Programmen, die aus tatsächlichen Programmen übernommen wurden. Diese werden aus [10] ausgewählt.

5.1 Funktionale Analyse

Man betrachte die Programme zuerst funktional, d.h. man überprüft, ob sie zu den gewünschten Ergebnissen führen.
Men betrachte zuerst die Standartsituationen. In der folgenden Tabelle 5.1 sind die betrachteten Situation beschrieben. Dazu wird für jede Situation eine kurze Beschreibung angegeben und dann überprüft, ob die beiden Detektoren in der Lage sind diese korrekt zu klassifizieren, d.h. ein Warnung über einen Deadlock auszugeben wenn ein solcher vorliegt, und keine Warnung zu geben, wenn kein Deadlock auftreten kann.

Ein potenzielles Deadlock, welches durch das zyklische Locking von zwei Deadlocks auftritt kann durch beide Tools erkannt werden (1.1). Ebenso erkenne beide, dass kein Deadlock vorliegt, wenn das Locking von mehreren Locks nicht zyklisch verläuft (1.2). Sobald diese Zyklen allerdings eine Länge von 3 oder mehr erreichen, werden diese durch sasha-s nicht mehr erkannt, da dieser Detektor nur nach Loops der Länge zwei sucht (2). Deadlock-Go hingegen ist in der Lage auch diese potentiellen Deadlocks zu erkennen.
Auch bei Deadlocks, in denen zwar zyklisches Locking auftritt, welche aber auf Grund von Gate-Locks nicht zu einem tatsächlich Deadlock führen können, schneidet Deadlock-Go besser ab (3). Da dieser mit Lock-Bäumen und nicht mit einem Lock-Graphen implementiert ist, wird in diesem Fall kein potentielles Deadlock ausgegeben. sasha-s hingegen erkennt nicht, dass solch ein Deadlock nicht auftreten kann und gibt somit eine false-positive Mitteilung über ein potentielles Deadlock aus.
sasha-s ist standardmäßig dazu in der Lage doppeltes Locking zu erkennen. Bei Deadlock-Go hängt die Erkennung solcher Deadlocks von den Optionen ab. UNDEAD, auf dem die Implementierung basiert ist nicht in der Lage, doppeltes Locking zu erkennen. Da diese aber dennoch auftreten können, wurde diese Implementierung um eine solche Erkennung erweitert. Diese kann allerdings in den Optionen deaktiviert werden. Standardmäßig ist Deadlock-Go also auch in der Lage solche Deadlocks durch doppeltes Locking zu erkennen (5).
Bei verschachtelten Routinen wie in 3.1.2 beschrieben ist keines der Tools in der Lage das potentielle Deadlock zu erkennen (4).
Für tatsächlich auftretende Deadlocks, welche durch zyklisches Locking auftreten sind beide Detektoren in der Lage, solche Situationen zu erkennen (6).

Für sasha-s sind keine Try-Lock-Operationen implementiert. Aus diesem Grund können potentielle Deadlocks, die solche Try-Lock Operationen besitzen nicht wirklich erkannt werden. Deadlock-Go besitzt eine Implementierung von Try-Lock- Operationen. Doppeltes Locking kann für Try-Lock erkannt werden, sowohl wenn es auch bei ein Try-Lock zu doppeltem Locking kommt (7.1), als auch wenn es durch die Try-Lock-Operation doppeltes Locking verhindert wird (7.2). Deadlock-Go ist nicht in der Lage Deadlocks zu erkennen, wenn dieses durch zyklisches Locking auftritt, wobei der Zyklus ein Try-Lock enthält. Es wird kein potentieller Deadlock erkannt, unabhängig davon, ob der Zyklus mit dem Try-Lock zu einem Deadlock führen kann oder nicht (8).

ID	**Typ**	**Deadlock-Go**	**sasha-s**
1.1	Potentielles Deadlock durch zyklisches Locking von zwei Locks	Ja	Ja
1.2	Kein Potentielles Deadlock, da die Locks nicht zyklisch sind	Ja	Ja
2	Potentielles Deadlock durch zyklisches Locking von drei Locks	Ja	Nein
3	Zyklisches Locking welches aber durch Gate-Locks nicht zu einem Deadlock führen kann	Ja	Nein
4	Potentielles Deadlock, welches durch Verschachtlung mehrerer Routinen (fork/join) verschleiert wird	Nein	Nein
5	Deadlock durch doppeltes Locken	Ja	Ja
6.1	Tatsächliches Deadlock durch zyklisches Locking von Locks in zwei Routinen	Ja	Ja
6.2	Tatsächliches Deadlock durch zyklisches Locking von Locks in drei Routinen	Ja	Ja
7.1	Doppeltes Locking mit TryLock (TryLock → Lock)	Ja	Nein*
7.2	Kein Doppeltes Locking mit TryLock (Lock → TryLock)	Ja	Nein*
8.1	Deadlock durch zyklisches Locking mit TryLock	Nein	Nein*
8.2	Zyklisches Locking, welches durch TryLock nicht zu einem Deadlock führt	Ja	Nein*
9.1	Potentielles Deadlock mit RW-Lock in zwei Routinen (R1: x.RLock → y.Lock, R2: y.Lock → x.Lock)	Ja	Ja
9.2	Potentielles Deadlock mit RW-Lock in zwei Routinen (R1: x.RLock → y.Lock, R2: y.RLock → x.Lock)	Ja	Ja
9.3	Kein potentielles Deadlock mit RW-Lock in zwei Routinen (R1: x.RLock → y.RLock, R2: y.RLock → x.RLock)	Ja	Nein
9.4	Kein potentielles Deadlock mit RW-Lock in zwei Routinen (R1: x.Lock → y.RLock, R2: y.RLock → x.Lock)	Ja	Nein
9.5	Kein potentielles Deadlock mit RW-Lock in zwei Routinen (R1: x.RLock → y.Lock, R2: y.RLock → x.RLock)	Ja	Nein
9.6	Kein potentielles Deadlock mit RW-Lock in zwei Routinen (R1: x.Lock → y.RLock, R2: y.RLock → x.RLock)	Nein	Nein
10.1	Kein Potentielles Deadlock, wegen Lock von RW-Locks als Gate-Locks.	Ja	Nein
10.2	Potentielles Deadlock, da R-Lock von Deadlock nicht als Gate-Lock funktioniert.	Ja	Ja
11.1	Doppeltes Locking von RW-Locks, welches zu Deadlock führt (Lock→Lock, RLock→Lock, Lock→Rlock)	Ja	Ja
11.2	Doppeltes Locking von RW-Locks, welches nicht zu einem Deadlock führt(RLock→Rlock)	Ja	Nein

Tabelle 5.1: Funktionale Analyse der Standartsituationen
*: sasha-s implementiert keine Try-Locks

Für RW-Locks gibt es in beiden Detektoren ein Implementierung. Allerdings unterscheidet sich die Detektion von Deadlocks in sasha-s für RW-Locks nicht von der für normale Locks. Aus diesem Grund kommt es bei diesem Detektor zu false-positives, wenn ein Zyklus in den Lock-Operationen aufgrund von RLock-Operationen nicht zu einem Deadlock führen kann. Deadlock-Go kann in vielen dieser Situationen eine false-positive Meldung verhindern (9.1-9.5), allerdings gibt es auch hier Situationen, in denen eine false-positive Meldung nicht verhindert werden kann (9.6).
Wie schon bei normalen Locks ist sasha-s auch bei RW-Locks nicht in der Lage zu erkennen, wenn zyklisches Locking, durch Gate-Locks nicht zu einem Deadlock führen kann, während dies für Deadlock-Go möglich ist (10.1). Es ist auch in der Lage zu erkennen, dass R-Locks nicht als Gat-Locks funktionieren. Da sasha-s allgemein nicht in der Lage ist Gate-Locks zu erkennen, wird dies automatische erkannt. (10.2).
Ähnliches gilt auch für doppeltes Locking. Sasha-s ist nicht in der Lage zu erkennen, wenn doppeltes Locking wegen R-Lock nicht auftreten können (11.2), erkennt aber doppeltes Locking allgemein, und damit auch, wenn es mit R-Locks zu doppeltem Locking kommt (11.1). Deadlock-Go kann beide Situationen korrekt erkennen.

Im folgenden sollen nun Beispielprogramme aus [5] betrachtet werden. Für Deadlock-Go wurden dabei alle Optionen aktiviert, um ein optimales Detektionsergebnis zu erhalten. Die folgende Tabele 5.2 gibt für jedes Programm an, ob es von den Detektoren erkannt wurde.

ID	**Deadlock-Go**	**sasha-s**
Cockroach584	Ja	Ja
Cockroach9935	Ja	Ja
Cockroach6181	Ja	Ja
Cockroach7504	Ja	Ja
Cockroach10214	Ja	Ja
Etcd5509	Nein	Nein
Etcd6708	Ja	Ja
Etcd10492	Ja	Ja
Kubernetes13135	Ja	Ja
Kubernetes30872	Ja	Ja
Moby4951	Ja	Ja
Moby7559	Ja	Ja
Moby17176	Nein	Nein
Moby36114	Ja	Ja
Syncthing4829	Ja	Ja

Tabelle 5.2: Funktionale Analyse der Beispielprogramme

Beide Detektoren waren in der Lage die potentiellen Deadlocks in 13 der 15 Beispielprogramme zu erkennen. Bei zwei der Programm war keiner der beiden Detektoren in der Lage, einen potentiellen Deadlock zu finden. Bei dieses beiden Programmen handelt es sich um Programme, bei denen es bei besonderen Abläufen dazu kommt, das vergessen wird Locks frei zu geben, was zu einem Deadlock führen kann. Da diese Pfade aber nur in sehr speziellen Situationen abgelaufen werden ist es verständlich, dass die Detektoren nicht in der Lage sind, diese zu erkennen.

Es ist erstaunlich, dass sasha-s gleich viele Probleme richtig erkannt hat, obwohl Deadlock-Go in den Standartproblemen in Tab. 5.1 bessere Ergebnisse gezeigt hat. Dies lässt darauf schließen, dass solche Situationen, in denen Deadlock-Go besser abschneidet nicht sehr häufig in tatsächlichen Situationen auftreten. Es muss allerdings auch beachtet werden, dass es sich bei

den betrachteten Beispielen um Situationen handelt, in denen Deadlocks tatsächlich auftreten, wären Deadlock-Go vor allem bei der Verhinderung von false-positives besser abschneidet als sasha-s. Diese Situation kommen in dem Beispielen allerdings nicht vor.

5.2 Zeitliche Analyse

Im folgenden soll der Einfluss der Detektion auf die Laufzeit eines Programms für beide Detektoren betrachtet werden. Dafür werden die Laufzeiten der Standardprogramme aus Tab. 5.1 betrachtet, welche nicht zu einem tatsächlichen Deadlock führen und von keinem Detektor (fälschlicherweise) als solches erkannt wird. Außerdem werden keine Beispiele für Beispiele mit Try-Locks betrachtet, da diese von sasha-s nicht unterstützt werden.
Für jedes dieser Programme wird in der folgenden Tabelle die Laufzeit ohne sowie mit Detektor angezeigt. Für Deadlock-Go wird jeweils sowohl die Laufzeit der Programmausführung (ohne abschließende Detektion) und die Gesamtlaufzeit angegeben. Dabei wird auf verschiedene Weisen gemessen. Für die Messung ohne Detektor wird die Messung 1000 mal wiederholt der Mittelwert angegeben. Dabei erhält man folgende Laufzeiten (normiert auf durchschnittliche Zeit pro Durchlauf):

ID	Laufzeit [μs]
1.1	1.7
1.2	1.6
2	2.6
3	1.5
4	2.0
9.1	1.4
9.2	1.4
9.3	1.4
9.4	1.5
9.5	1.6
9.6	1.4
10.1	1.6
10.2	1.5
∅	1.63

Tabelle 5.3: Laufzeit der Programme ohne Detektor pro Durchlauf

Für die Messung mit den Detektoren werden jeweils mehrere Messungen betrachtet. Für den Deadlock-Go Detektor werden jeweils 4 Messungen gemacht: Messung der Laufzeit von 1000 Durchläufen (mit und ohne periodische Detektion) ohne abschließend Detektion, Messung der Laufzeit der abschließenden Detektion nach einem Durchlauf und Messung der Laufzeit nach 1000 Durchläufen. Die Laufzeit wir jeweils wieder normiert auf die durchschnittliche Zeit pro Durchlauf angegeben. Für die Messungen wurde die Ausgabe des Warntextes deaktiviert. Für die Zeit der abschließenden Detektion wird die Gesamtzeit angegeben.

ID	Laufzeit ohne periodischer Detektion [ms]	Laufzeit mit periodischer Detektion [ms]	Abschließende Detektion nach einem Durchlauf [ms]	Abschließende Detektion nach 1000 Durchläufen [ms]
1.1	2.496	2.586	0.031	96.496
1.2	2.523	2.668	0.056	112.773
2	4.212	4.215	0.040	229.732
3	2.908	3.237	0.010	200.216
4	3.981	4.151	0.026	104.504
9.1	2.574	2.747	0.044	95.580
9.2	2.706	2.773	0.045	117.593
9.3	2.921	2.973	0.028	140.848
9.4	2.901	2.986	0.031	118.540
9.5	3.004	3.228	0.038	101.845
9.6	2.750	2.967	0.010	128.422
10.1	2.743	2.922	0.009	119.455
10.2	2.724	2.725	0.041	196.584
∅	2.957	3.091	0.0315	135.583

Tabelle 5.4: Laufzeit der Programme mit Deadlock-Go. Laufzeit Werte pro Durchlauf

Für sasha-s werden die Programme wieder 1000 mal durchlaufen und anschließend der Mittelwert pro Durchlauf betrachtet. Dabei erhält man folgende Werte.

ID	Laufzeit [ms]
1.1	12.782
1.2	8.687
2	12.696
3	18.451
4	7.953
9.1	8.621
9.2	11.566
9.3	8.583
9.4	12.169
9.5	9.219
9.6	13.545
10.1	7.737
10.2	9.184
∅	10.861

Tabelle 5.5: Laufzeit der Programme mit sasha-s. Laufzeit Werte pro Durchlauf

Es ist sehr offensichtlich, dass die Programme mit den Detektoren eine signifikant längere Laufzeit haben. Während die Programme ohne Detektor im Schnitt eine Laufzeit von 1.63μs haben, haben sie mit sasha-s ein durchschnittliche Laufzeit von 10.861 ms und für Deadlock-Go mit periodischer Detektion 3.123 ms - 138.674 ms, Es sollte beachtet werden, dass die Differenz zwischen der Laufzeit mit und ohne Detektor stark von der Anzahl der auf Locks ausgeführten Optionen ist. Da die hier verwendeten Beispielprogramme nahezu ausschließlich aus solchen Operationen bestehen, ist der Effekt besonders hoch.

Wie die beiden Detektoren gegenseitig abschneiden hängst sehr stark von der Anzahl der betrachteten Routinen und Locks bzw. Dependencies ab. Während bei kleinen Programmen Deadlock-

Go besser abschneidet, (vgl. Tab. 5.4 mit Abschließende Detektion nach einem Durchlauf), schneidet bei großen Programmen mit vielen Routinen und Dependencies sasha-s deutlich besser ab (vgl. Tab 5.4 Abschließende Detektion nach 1000 Durchläufen). Dabei fällt besonders die abschließende Detektion ins Gewicht, welche sehr stark von der Anzahl der Dependencies und Routinen abhängt.
Beide Detektoren sind in der Lage, die Detektion zu deaktivieren, so dass die dann nur noch einen sehr geringe Verlängerung der Laufzeiten haben. Nachdem ein Programm zu Ende entwickelt wurde, und alle potentiellen Deadlock beseitigt wurden, kann die Detektion somit deaktiviert werden um die schnelleren Laufzeiten zu erreichen, ohne dass der Detector aus dem Code entfernt werden muss.

6 Zusammenfassung

Das Projekt betrachtete Möglichkeiten zur Detektion von Ressourcen-Deadlocks in Go und entwickelte, sowie implementierte einen eigenen Detektor. Neben dem selbst implementierten Detektor Deadlock-Go wurde ein weiterer Detektor go-deadlock betrachtet. Beide Detektoren benutzen dynamische Detektion um Deadlocks, welche durch Locks und RW-Locks erzeugt werden zu erkennen. Dabei nutzt go-deadlock Lock-Graphen, während Deadlock-Go auf Lock-Bäumen arbeitet. Beide Programme sind in der Lage viele solche Deadlocks zu erkennen, wobei Deadlock-go in den Betrachteten Standartsituationen besser abschneidet, gerade auch was die Vermeidung von Fasle-Positives bei RW-Locks angeht. Beide Detektoren haben einen negativen Einfluss auf die Laufzeit der Programme, wobei der genau Einfluss von der Anzahl der Locks, Routinen und Dependencies abhängt. Nachdem die Detektoren in der Softwareentwicklung verwendet wurden um potentielle Deadlocks zu verhindern, können sie beide deaktiviert werden. Dadurch ist es möglich, dass die Programme nahezu ohne Verschlechterung der Laufzeit laufen können, ohne dass das Programm umgeschrieben werden muss.

Quellen

[1] P. Joshi, C.-S. Park, K. Sen und M. Naik, „A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks“, *SIGPLAN Not.*, Jg. 44, Nr. 6, S. 110–120, Juni 2009, ISSN: 0362-1340. Adresse: https://doi.org/10.1145/1543135.1542489.

[2] J. Zhou, S. Silvestro, H. Liu, Y. Cai und T. Liu, „UNDEAD: Detecting and preventing deadlocks in production software“, in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2017, S. 729–740. DOI: 10.1109/ASE.2017.8115684.

[3] S. Lu, S. Park, E. Seo und Y. Zhou, „Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics“, Jg. 43, Nr. 3, S. 329–339, März 2008, ISSN: 0362-1340. DOI: 10.1145/1353536.1346323. Adresse: https://doi.org/10.1145/1353536.1346323.

[4] sasha-s, *go-deadlock*, https://github.com/sasha-s/go-deadlock, 2018. (besucht am 03. 05. 2022).

[6] M. Sulzmann, *Dynamic deadlock prediction*, https://sulzmann.github.io/AutonomeSysteme/lec-deadlock.html. (besucht am 03. 05. 2022).

[7] S. Bensalem und K. Havelund, „Dynamic Deadlock Analysis of Multi-threaded Programs“, in *Hardware and Software, Verification and Testing*, S. Ur, E. Bin und Y. Wolfsthal, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, S. 208–223, ISBN: 978-3-540-32605-2. DOI: 10.1007/11678779_15.

[8] The Go Authors, *Go Runtime Library src/runtime/proc.go*, https://go.dev/src/runtime/proc.go#L4935. (besucht am 10. 05. 2022).

[10] T. Yuan, G. Li, J. Lu, C. Liu, L. Li und J. Xue, „GoBench: A Benchmark Suite of Real-World Go Concurrency Bugs“, in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, S. 187–199. DOI: 10.1109/CGO51591.2021.9370317.

Referenzen

[5] E. Kassubek, *Deadlock-Go*, https://github.com/ErikKassubek/Deadlock-Go, 2021. (besucht am 11. 07. 2022).

[9] E. Kassubek, *DeadlockExamples*, https://github.com/ErikKassubek/BachelorProjektGoDeadlockDetection/tree/main/DeadlockExamples, 2021. (besucht am 11. 07. 2022).