Bachelorprojekt

im Studiengang
Bachelor of Science in Informatik

Dynamic Deadlock Detection in Go

Erik Daniel Kassubek

10.08.2022

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 existierender Detektor „Go-Deadlock“ betrachtet und anschließend ein eigener Detektor, basierend auf dem UNDEAD-Detektor [9], entwickelt. Dieser 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[4]. 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 [9]. 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 dasselbe 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 z.B. dazu, dass in einer 2008 durchgeführten Stichprobe in vier open-source Anwendungen (MySql, Apache, Mozilla und OpenOffice) etwa 30% aller Concurrency-Probleme auf Deadlocks zurückzuführen waren [5].

Go besitzt eine sogenannte Go Runtime Deadlock Detection um Deadlocks zu erkennen. Dazu 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 [7]. Allerdings kann dies nur tatsächlich auftretende Deadlocks erkennen. Eine Erkennung, ob in dem Code ein Deadlock potenziell möglich ist findet hierbei nicht statt. Die Detektion erkennt auch nur Deadlocks, wenn sie alle Routinen betreffen. Wenn es Routinen gibt, die sich zyklisch blockieren, andere Routinen aber nicht blockiert sind, wird solch eine Situation nicht erkannt.

Da das Auftreten von Deadlocks sehr stark von dem tatsächlichen Ablauf eines Programms abhängt, kann ein Programm, bzw. eine Implementierung von Locks, welche potenzielle Deadlocks erkennen kann sehr hilfreich sein. Die Betrachtung solcher Programme stellt den Hauptteil dieses Projekts da. Dabei werden dynamische Programme betrachtet, also Programme, 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.

Der Bericht ist folgendermaßen aufgebaut: Zuerst wird auf den theoretischen Hintergrund bezüglich Locks und Deadlocks, sowie die Erkennung von potenziellen Deadlocks mit Lock-Graphen und Lock-Bäumen eingegangen. Außerdem wird die Funktionsweise des UNDEAD-Algorithmus [9] betrachtet. Anschließend wird die Funktionsweise eines Detektors für Deadlocks “go-deadlock” [6] analysiert, welcher auf Lock-Graphen basiert. Der Hauptteil des Projekts beschäftigte sich mit der Entwicklung und Implementierung eines auf Lock-Bäumen basierten Deadlock-Detektors für Go. Dieser basiert auf dem UNDEAD Algorithmus. Er wurde allerdings um weitere Funktionen, wie z.B. RW-Locks erweitert. Dessen Aufbau und Funktionsweise wird in Kapitel 5 beschrieben. Zum Schluss werden die beiden betrachteten Detektoren bezüglich ihrer Fähigkeit potenzielle Deadlocks bzw. Situationen, welche nicht zu Deadlocks führen können, korrekt zu erkennen, verglichen. Außerdem wird der Einfluss der Detektoren auf die Laufzeit der Programme analysiert.

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 Routinen nicht gleichzeitig in bestimmten Bereichen aufhalten dürfen.
Locks gehören zu den am weitesten verbreiteten Mechanismen, um solche kritische Bereiche in einem Programm zu schützen [9]. 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 das Lock von R_0 wieder freigegeben 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 beanspruchen 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 möglich 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 beansprucht 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 momentan hält.

2.1.2 RW-Locks

Neben den allgemeinen Locks gibt es noch s.g. Reader-Writer Locks. Diese haben im Vergleich zu den allgemeinen Locks zwei Lock und TryLock Operation, R-Lock und W-Lock (analog für TryLock). W-Lock, im folgenden einfach Lock genannt funktioniert identisch zu Lock in den allgemeinen Locks. Bei R-Locks hingegen ist es, anders als bei Lock, möglich, 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

2.2.1 Zyklischer Deadlock

Ein Deadlock ist ein Zustand in einem nebenläufigen Programm, also einem Programm, in dem mehrere Routinen nebenläufig ausgeführt werden, wobei alle laufenden Routinen zyklisch auf die Freigabe von Ressourcen warten. Ein solcher Deadlock kann nun entstehen, wenn all Routinen vor einem Lock warten müssen, wobei die Locks immer von einer anderen Routine gehalten werden.

Man betrachte dazu das folgende Beispiel mit den Locks x und y und den beiden Routinen R_1 und R_2.

```
1  func cyclicLockingExample() {
2      var x Mutex
3      var y Mutex
4
5      go func() { // R1
6          y.Lock()
7          x.Lock()
8          x.Unlock()
9          y.Unlock()
10     }()
11
12     go func() { // R2
13         x.Lock()
14         y.Lock()
15         y.Unlock()
16         x.Unlock()
17     }()
18 }
```

Betrachtet man nun den zeitlichen Ablauf der Operationen auf den Locks in der Ausführung des Programm, kann man den folgenden Ablauf erhalten. $acq(l)$ bezeichnet dabei, dass das Lock l erfolgreich beansprucht wird (l.Lock()) und $rel(l)$, dass l wieder freigegeben wird (l.Unlock()). Die erste Spalte gibt die Zeile an, durch welche die Operation in dem Programm verursacht wurde. Die restlichen Spalten entsprechen den betrachteten Routinen.

	R_1	R_2
6.	$acq(y)$	
7.	$acq(x)$	
8.	$rel(x)$	
9.	$rel(y)$	
14.		$acq(x)$
15.		$acq(y)$
16.		$rel(y)$
17.		$rel(x)$

Da R_1 und R_2 gleichzeitig ablaufen ist es möglich, dass sich die Reihenfolge der einzelnen Operationen zwischen den Routinen ändert (die Reihenfolge innerhalb einer Routine ist immer gleich). Damit ist für das Programm auch folgender Ablauf möglich. Dabei impliziert $B_j - acq(l)$, dass $acq(l)$ nicht ausgeführt werden konnte, bzw. dass die Routine R_i vor der Lock-Operation halten muss, da das Lock bereits von Routine R_j beansprucht wird.

	R_1	R_2
6.	$acq(y)$	
14.		$acq(x)$
7.	$B_2 - acq(x)$	
15.		$B_1 - acq(y)$

In diesem Beispiel wartet nun R_1 darauf, dass das Lock x freigegeben wird und R_2 wartet darauf, dass Lock y freigegeben wird. Da alle Routinen warten müssen, bis ein Lock freigegeben wird, allerdings keine der Routinen weiter laufen kann, um ein Lock freizugeben, kommt es zum Stillstand. Dieser Zustand wird als zyklischer Deadlock bezeichnet.

2.2.2 Deadlock durch doppeltes Locking

Eine andere Situation, bei der ein Deadlock entstehen kann tritt auf, wenn eine Routine das selbe Lock mehrfach beansprucht, ohne es zwischendurch wieder freizugeben. Dies ist sogar möglich, wenn in dem Programm zu diesem Zeitpunkt nur eine Routine aktiv ist. Ein Beispiel dafür gibt das folgende Programm:

```
1 func doubleLockingExample() {
2     var x Mutex
3
4     go func() { // R1
5         x.Lock()
6         x.Lock()
7         x.Unlock()
8     }()
9 }
```

mit dem folgenden Programmablauf

	R_1
5.	$acq(x)$
6.	$B_1 - acq(x)$

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

2.3 Deadlock-Detektion

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 Programms 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, kann versucht werden, den Ablauf der in dem Programm vorkommenden Lock- und Unlock-Operationen aufzuzeichnen und basierend auf diesem Trace Lock-Graphen oder Lock-Bäume aufzubauen. Durch Zyklen in diesen Strukturen lassen sich Rückschlüsse auf potentielle Deadlocks ziehen.

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 [2]. Mathematisch ausgedrückt gilt also

$$(l_1, l_2) \in E \Leftrightarrow \exists t_1, t_3 \nexists t_2 \; \exists i \; ((t_1 < t_2 < t_3) \wedge acq_i(l_1)[t_1] \wedge rel_i(l_1)[t_2] \wedge acq_i(l_2)[t_3])$$

wobei $acq_i(l_j)[t_k]$ bedeutet, dass $acq(l_i)$ zum Zeitpunkt t_k auf Routine j ausgeführt wird und equivalent für $rel_i(l_j)[t_k]$.
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 [2]. Es sei zusätzlich noch gesagt, dass ein Zyklus in einem Lockgraphen nicht immer auf ein potenzielles 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:

```
1  func gateLock() {
2      var x Mutex
3      var y Mutex
4      var z Mutex
5
6      go func() {   // R1
7          z.Lock()
8          y.Lock()
9          x.Lock()
10         x.Unlock()
11         y.Unlock()
12         z.Unlock()
13     }
14
15     go func() {   // R2
16         z.Lock()
17         x.Lock()
18         y.Lock()
19         y.Unlock()
20         x.Unlock()
21         z.Unlock()
22     }
23 }
```

In diesem Fall bilden die Locks x und y in dem Lock-Graphen 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

Der Detektor, welcher im Laufe des Projects entwickelt werden soll, basiert auf dem UNDEAD Algorithmus, welcher in folgenden Abschnitten (Kap. 3) noch genauer betrachtet werden soll. Dieser arbeitet nicht mit Lock-Graphen, sondern mit Lock-Bäumen. Anders als bei Lock-Graphen, speichert hierbei jede Routine seine eigenen Abhängigkeiten. Die Knoten des Baums stellen dabei die Locks l_i in der Routine da. Es gibt eine Kante von l_1 nach l_2 wenn l_1 das von der Routine momentan gehaltene Lock repräsentiert, welches zuletzt von der Routine beansprucht worden ist, währen das Lock l_2 beansprucht wird[1]. Mathematisch ausgedrückt ist ein Lock-Baum $B_i = (L_i, E_i)$ für Routine i also ein Baum, wobei L_i eine Teilmenge der in der Routine vorkommenden Locks is, und die Menge der Kanten E_i über

$$\begin{aligned}(l_1, l_2) \in E_i \Leftrightarrow & \exists t_1, t_4 \nexists t_2, t_3, l_3 \\ & ((t_1 < t_2 < t_3 < t_4 \vee t_1 < t_3 < t_2 < t_4) \wedge (\neg l_1 = l_3 \wedge \neg l_2 = l_3) \\ & \wedge acq_i(l_1)[t_1] \wedge (rel_i(l_1)[t_2] \vee acq_i(l_3)[t_3]) \wedge acq_i(l_2)[t_4])\end{aligned}$$

gegeben ist.
Im folgenden wird, wie in UNDEAD, ein Lock-Baum als eine Menge von Dependencies betrachtet. Eine solche Dependency besteht aus einem Lock `mu` und einer Menge von Locks `hs`

(`holdingSet`), von denen `mu` anhängt, für die es also einen Pfad von x nach mu mit $x \in hs$ gibt. Ein potenzielles Deadlock liegt nun vor, wenn es in der Menge aller Dependencies eine gültige, zyklische Kette gibt.
In UNDEAD werden keine RW-Locks betrachtet. In diesem Fall ist ein Pfad mit n Elementen ist eine gültige Kette, wenn die folgenden Eigenschaften gelten:

$$\forall\, i, j \in \{1, ..., n\}\ (dep_i = dep_j) \rightarrow (i = j) \tag{2.3.2.a}$$
$$\forall\, i \in \{1, ..., n\}\ mu_i \in hs_{i+1} \tag{2.3.2.b}$$

Dabei bezeichnet mu_i den Mutex und hs_i das holdingSet der i-ten Dependency dep_i in der Kette.
(2.3.2.a) stellt sicher, dass die selbe Dependency nicht mehr als ein mal in der Kette auftauchen kann. (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.
Ein potenzielles Deadlock ergibt sich nun, wenn diese Kette einen Zyklus bildet, wenn als zusätzlich

$$mu_n \in hs_1 \tag{2.3.2.c}$$

gilt.
Man betrachte dazu das folgende Beispiel:

```
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     }
34 }
```

Dabei gibt sich die Menge der Dependencies für $R1$ zu

$$B_1 = \{(w, \{v\}), (y, \{z\}), (y, \{x\})\}$$

für R_2 zu

$$B_2 = \{(x, \{w\})\},$$

und für R_3 zu

$$B_3 = \{(v, \{x\})\},$$

wobei eine Dependency immer als Tuple der Form (mu, hs) angegeben wird. Dies lässt sich folgendermaßen graphisch darstellen:

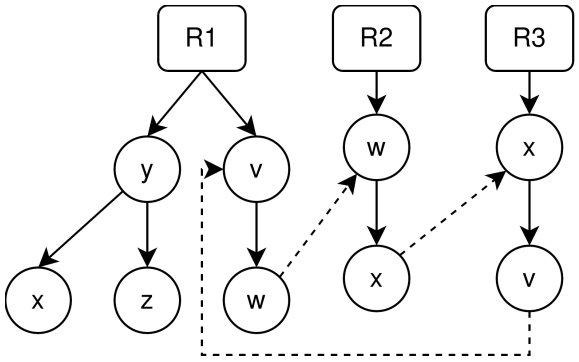

Abbildung 2.1: Graphische Darstellung der Lock-Bäume aus dem Beispiel. Die Knoten entsprechen dabei Locks. Die durchgezogenen Pfeile repräsentieren die Kanten der Lock-Bäume. Die gestichelten Kanten zeigen den enthaltenen Zyklus an.

Verbindet man nun gleiche Locks zwischen den einzelnen Bäumen (im Bild mit gestrichelten Pfeilen), besitzt der entstehende Graph einen Zyklus zwischen mu_1, mu_2 und mu_3. Solch ein Zyklus indiziert einen potentiellen Deadlock.

Möchte man zusätzlich auch RW-Lock betrachten, müssen diese Regeln erweitert werden. Es sei $read(mu)$ wahr, wenn das Locking von mu, welches in der Dependency abgebildet ist, durch ein R-Lock zustande gekommen ist. In diesem Fall gibt es nur dann einen Deadlock, wenn zusätzlich die folgenden Eigenschaften gelten:

$$\forall i \in \{1, \ldots, n\} read(mu_i) \rightarrow (\forall mu \in hs_{i+1}(mu = mu_i) \rightarrow \neg read(mu)) \quad \text{(2.3.2.d)}$$

$$read(mu_n) \rightarrow (\forall\ mu \in hs_1 : (mu = mu_n) \rightarrow \neg read(mu)) \quad \text{(2.3.2.e)}$$

$$\forall\ i, j \in \{1, ..., n\}\ \neg(i = j) \rightarrow (\exists\ mu_1 \in hs_i\ \exists\ mu_2 \in hs_j((mu_1 = mu_2) \rightarrow (read(mu_1) \wedge read(mu_2)))) \quad \text{(2.3.2.f)}$$

Auch wenn (2.3.2.a) - (2.3.2.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. Dass solche Pfade ausgeschlossen werden wird durch (2.3.2.d) und (2.3.2.e) sichergestellt. Formel (2.3.2.f) beschäftigt sich mit Gate-Locks. Sie besagt, 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.
Man betrachte das folgende Beispiel:

```
1  func readerLock() {
2      var x RWMutex
3      var y RWMutex
4      var z RWMutex
5
6      go func() {   // R1
7          x.RLock()
8          y.RLock()
9          y.RUnLock()
10         x.RUnLock()
11     }
12
13     go func() {   // R2
14         y.RLock()
15         x.RLock()
16         x.RUnLock()
17         y.RUnLock()
18     }
19 }
```

Die Dependencies geben sich dabei zu

$$B_1 = \{(y, \{x\})\}$$
$$B_2 = \{(x, \{y\})\}$$

Die Formeln (2.3.2.a) - (2.3.2.c), sind mit diesem Beispiel erfüllbar, d.h. ein potentielles Deadlock würde erkannt werden. Durch die RLock-Operationen sind aber keine Deadlocks möglich. Mit (2.3.2.d) - (2.3.2.f) wird die falsche Meldung verhindert.

Die Verwendung von Lock-Bäumen hat mehrere Vorteile gegenüber Lock-Graphen. Da jede Routine ihre eigene Datenstruktur hat, wird verhindert, dass es durch gleichzeitigen Zugriff verschiedener Routinen auf die selbe Datenstruktur zu Problemen kommt. Zudem muss die Zugehörigkeit der Abhängigkeiten zu verschiedenen Routinen nicht explizit gespeichert werden. Ein weiterer Vorteil besteht darin, dass bei der Erkennung von Deadlocks mit Lock-Bäumen Situationen, die aufgrund von Gate-Locks nicht zu Deadlocks führen können, automatisch nicht als Deadlock erkannt werden, während es bei Lock-Graphen in solchen Fällen ohne zusätzliche Tests zu falschen Meldungen kommt. Man betrachte dazu erneut das Beispielprogramm zu Gate-Locks aus 2.3.1. In diesem Fall erhält man die folgenden Lock-Bäume, repräsentiert durch die Menge ihrer Dependencies:

$$B_1 = \{(y, \{z\}), (x, \{z, y\})\}$$
$$B_2 = \{(x, \{z\}), (y, \{z, x\})\}$$

Es ist nun leicht sich zu überzeugen, dass es mit den oben genannten Formeln nicht zu einer Deadlock-Meldung kommt.

2.3.3 Doppeltes Locking

Doppeltes Locking, wie in Kapitel 2.2.2 beschrieben, lässt sich nicht aus den Lock-Graphen oder Lock-Bäumen erkennen, da diese nur den Ablauf der tatsächlich ausgeführten Operationen speichern. Eine Situation, die in solch einer Struktur als doppeltes Locking erkannt werden könnte, führt aber immer zu einem tatsächlich auftretenden Deadlock. Es ist daher notwendig diese gesondert zu betrachten.
Dazu speichert jedes Lock, von welchen Routinen es momentan gehalten wird, und für welche dieser Beanspruchungen es über R-Lock beansprucht worden ist.

Währen dem (R-)(Try-)Lock Vorgang, noch bevor es zu der eigentlichen Beanspruchung kommt, wird überprüft, ob das Lock in der selben Routine bereits gehalten wird, ohne dass es sich bei beiden Operationen um R-Locks handelt. In diesem Fall kommt es zu einem Deadlock.

3 UNDEAD

UNDEAD [9] ist ein Algorithmus zur Detektion tatsächlicher und potenzieller Deadlocks. Zusätzlich versucht er potenzielle Deadlocks in weiteren Durchläufen zu verhindern. Da sich dieses Projekt aber nur mit der Detektion von Deadlocks befasst, soll darauf nicht weiter eingegangen werden.

UNDEAD implementiert Drop-in-Replacements für Locks mit Lock, TryLock und Unlock Operationen, mit denen nach potenziellen oder tatsächlichen Deadlocks gesucht werden soll. Der Algorithmus benutzt dabei eine Implementierung von Lock-Bäumen bestehend aus Dependencies.
Die Detektion in UNDEAD läuft in drei Phasen ab. Diese sind die Logging-Phase, die periodische Detektion und die abschließende Detektion.

3.1 Logging-Phase

Die Logging-Phase läuft während dem Ablauf des eigentlichen Programms und sammelt Informationen, die zur Erkennung von potenziellen Deadlocks notwendig sind. Wird eine Lock-Operation oder eine erfolgreiche Try-Lock Operation ausgeführt, wird eine neue Dependency erstellt. Diese werden für jede Routine in einer separaten Hashtabelle gespeichert, um gleichzeitigen Zugriff mehrerer Routinen auf solche Strukturen zu verhindern. Als Schlüssel für die Tabelle wird dabei der XOr Wert der Speicheradressen zweier Locks verwendet. Wenn z.B. das Lock l_1 von den Locks $\{l_2, l_3\}$ abhängt, wird der XOr Wert der Speicheradressen der beiden innersten Locks (hier z.B. l_1 und l_2) als Schlüssel verhindert, um Konflikte zwischen den verschiedenen Schlüsseln zu verhindern. Diese Hashtabellen bilden damit die Implementierung der Lock-Bäume.

Bei einer erfolgreichen Beanspruchung eines Locks speichert UNDEAD außerdem den momentanen Call-Stack der Routine, in der das Lock beansprucht wurde. Dieser ist für die eigentliche Detektion nicht relevant, wird aber dem Nutzer zur Verfügung gestellt, um die Situation, die zu dem potentiellen Deadlock führt besser zu verstehen.
UNDEAD versucht die Menge der gesammelten Daten, sowie die Aufrufe zum Abrufen der Call-Stacks so weit wie möglich zu reduzieren. Dazu werden diese nur abgerufen, wenn es wirklich notwendig ist, also z.B. nicht wenn die Beanspruchung eines Locks keine neue Dependency erzeugt. Es werden keine Informationen gespeichert, wenn in dem Programm momentan nur eine Routine läuft. Außerdem versucht UNDEAD, mehrfach auftretende Dependencies nicht mehrfach zu speichern, und in diesem Fall auch die Call-Stack Informationen nicht mehrfach abzufragen.

3.2 Periodische Detektion

Die periodische Detektion wird verwendet, um tatsächlich auftretende Deadlocks bereits während dem Ablauf des Programms zu erkennen. Dazu wird in regelmäßigen, zeitlichen Abständen eine Routine für die Detektion gestartet.
Jede Routine speichert eine aktuelle Dependency, also diejenige Dependency, die zuletzt erkannt wurde. Wenn die periodische Routine gestartet wird, macht der Detektor einen Snapshot dieser aktuellen Dependencies und sucht in diesen nach Zyklen, wie in Kap. 2.3.2 beschrieben. Wird

ein solcher Zyklus gefunden, wird überprüft, ob sich die Menge der aktuellen Dependencies seit dem Start der periodischen Detektion geändert hat. Ist dies der Fall, so geht UNDEAD davon aus, dass es sich nur um ein potenzielles Deadlock handelt, welches aber nicht aufgetreten ist. In diesem Fall wird das Programm normal weiter ausgeführt. Andernfalls geht UNDEAD davon aus, das ein tatsächliches Deadlock aufgetreten ist. Tritt dieser Fall auf, wird die abschließende Detektion (Kap. 3.3) gestartet und das Programm im Anschluss abgebrochen. Man betrachte dabei das folgende Beispiel:

```
1  func periodicalDetection() {
2      var w Mutex
3      var x Mutex
4      var y Mutex
5      var z Mutex
6
7      go func() { // R1
8          w.Lock()
9          x.Lock()
10         x.UnLock()
11         w.UnLock()
12         y.Lock()
13         z.Lock()
14         z.UnLock()
15         y.UnLock()
16         // periodical detection is started here
17     }
18
19     go func() { // R2
20         x.Lock()
21         w.Lock()
22         w.UnLock()
23         x.UnLock()
24         // periodical detection is started here
25         z.Lock()
26         y.Lock()
27         y.UnLock()
28         z.UnLock()
29     }
30 }
```

Zum Zeitpunkt, an dem die periodische Detektion gestartet wird, wurden folgende Dependencies aufgezeichnet, wobei der Stern symbolisiert, dass die entsprechende Dependency als letztes in die Liste eingefügt worden ist.

$$B_1 = \{(x, \{w\}), (z, \{y\})^*\}$$
$$B_2 = \{(w, \{x\})^*\}$$

Für die periodische Detektion werden nun nur die mit dem Stern gekennzeichneten Dependencies betrachtet. Dies bedeutet, dass das potentielle Deadlock zwischen x und w nicht erkannt wird. Dies ist aber auch nicht notwendig, da es in diesem Durchlauf nicht aufgetreten ist und das Programm somit nicht abgebrochen werden muss. Das potentielle Deadlock zwischen y und z wird in diesem Fall auch nicht erkannt, da R_2 den entsprechenden Teil noch nicht durchgeführt hat.
Auch wenn diese potentiellen Deadlocks hier nicht erkannt wurden, werden sie in der abschließenden Detektion erkannt.

3.3 Abschließende Detektion

Die abschließende Detektion wird ausgeführt, nachdem das eigentliche Programm abgeschlossen wurde, oder wenn sie durch die periodische Detektion gestartet wurde. Die Detektion basiert dabei auf dem iGoodLock Methode zur Detektion von Deadlocks, basierend aud DeadlockFuzzer [4] und MagicFuzzer [3]. Für die Detektion sucht UNDEAD nach Pfaden wie in Kap. 2.3.2 ((2.3.2.a), (2.3.2.b)) beschrieben. Dazu sucht es nach Dependency-Ketten durch Betrachtung aller möglichen Permutationen von Dependencies in zwei Routinen, dann in drei, usw. Es nutzt dabei eine Tiefensuche, um diese Ketten iterativ zu durchsuchen. Wenn eine solche Kette gefunden wurde, wird überprüft, ob sie einen Zyklus bildet ((2.3.2.c)). Ein solcher Zyklus deutet auf einen potenziellen Deadlock hin, was dazu führt, dass UNDEAD eine Information über den gefundenen potenziellen Deadlock ausgibt. Da UNDEAD keine RWLocks implementiert, müssen (2.3.2.d) - (2.3.2.f) nicht erfüllt werden um eine Deadlock-Meldung auszulösen.

4 Review für go-deadlock

Im folgenden soll eine Software zur Erkennung von Deadlocks analysiert werden. Dazu wird die Software "sasha-s/go-deadlock" [6], 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 Locks als auch RW-Locks implementiert. Dies wirkt sich allerdings nur auf die Anwendung des eigentlichen Locking aus, nicht aber auf die Erkennung von Deadlocks. Aus diesem Grund wird hierauf im Folgenden nicht weiter eingegangen, und die Methoden für die Erkennung von Deadlocks bezieht sich sowohl auf die allgemeinen als auch die RW-Locks.
Für die Erkennung werden drei verschiedene Fälle betrachtet:

- Doppeltes Locking
- Zyklisches Locking
- Timeout

Diese sollen im folgenden genauer betrachtet werden.

4.1 Doppeltes Locking

Um doppeltes 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 `cur` auftaucht. Ist dies der Fall, dann nimmt das Programm an, dass es sich hierbei um ein mögliches Deadlock durch doppeltes Locking handelt und führt entsprechende Schritte aus, um den Nutzer zu warnen. Wird ein Lock wieder freigegeben, wird der entsprechende Eintrag aus *cur* entfernt.

4.2 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 4.1 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 von Routine R beanspruchen, so wird für jedes Lock b, welches sich momentan in `cur` befindet und von R gehalten wird ein neuer Eintrag $beforeAfter\{b, p\}$ in `order` hinzugefügt. Die Werte, die für jeden Key in *order* gespeichert werden, entsprechen den Informationen, 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.

4.3 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 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 Go-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.

4.4 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 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 4.2. 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 dieser Detektor nicht in der Lage zu erkennen, wenn ein potenzielles Deadlock durch Gate-Locks unmöglich gemacht wird.

5 Implementierung des Deadlock-Detektors „Deadlock-Go “

Im Folgenden soll die Funktionsweise und Implementierung es selbst implementierten Detektors „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 [10] gefunden werden.
Der Detektor basiert zu größten Teil auf dem UNDEAD-Algorithmus (s. Kap. 3). Zusätzlich wurde er um RW-Locks 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 gespeichert, wo in dem Programm-Code die Initialisierung sowie die (Try-)(R-)Lock Operationen aufgerufen werden. Diese werden dem Nutzer bereitgestellt, wenn ein potenzielles Deadlock erkannt wurde, um das Auffinden und Korrigieren des entsprechenden Deadlocks im Programmcode zu erleichtern.
Das folgende Programm ist ein Beispiel zur Verwendung des Detektors:

```
1  import "github.com/ErikKassubek/Deadlock-Go"
2
3  func main() {
4      x := deadlock.NewLock()
5      y := deadlock.NewLock()
6
7      // make sure, that the program does not terminate
8      // before all routines have terminated
9      ch := make(chan bool, 2)
10
11     go func() {
12         x.Lock()
13         y.Lock()
14         y.Unlock()
15         x.Unlock()
16         ch <- true
17     }()
18
19     go func() {
20         y.Lock()
21         x.Lock()
22         x.Unlock()
23         y.Unlock()
24         ch <- true
25     }()
26     <-ch
27     <-ch
28
29     // start the final deadlock detection
30     deadlock.FindPotentialDeadlocks()
31 }
```

Es erzeugt den folgenden Output:

```
POTENTIAL DEADLOCK

Initialization of locks involved in potential deadlock:

/home/***/undead_test.go 4
/home/***/undead_test.go 5

Calls of locks involved in potential deadlock:

Calls for locks created at /home/***/undead_test.go 4
/home/***/undead_test.go 21
/home/***/undead_test.go 12

Calls for locks created at /home/***/undead_test.go 5
/home/***/undead_test.go 20
/home/***/undead_test.go 13
```

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

5.1 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.

5.1.1 (RW-)Mutex

Die Strukturen für die (RW-)Mutexe sind so implementiert, dass sie als Drop-In-Replacements für die klassischen sync.(RW)Mutexe 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 Programm 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 *var x Mutex* oder *var x RWMutex* 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 W-Lock gehalten werden. Werden diese allerdings von R-Locks gehalten, kann das selbe Lock von mehreren Routinen 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.
- `isRLock` (map[int]bool) (nur für RW-Mutex) speicher, von welcher Routine der Mutex als R-Lock gehalten wird.

Zusätzlich enthalten die (RW)-Locks noch Locks, die bei der gleichzeitigen Verwendung des Locks durch R-Lock eine gleichzeitiges schreiben in maps verhindern.

5.1.2 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.

5.1.3 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 die momentan von der Routine gehaltenen Locks enthält.
- `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` (map[string][]int) speichert Informationen über single-level Locks, also Locks, welche von keinem andere Lock abhängen. Auch dies ist wie dependencyMap dazu da, um zu verhindern, dass Informationen, die bereits bekannt sind, mehrfach gespeichert werden.

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

Im Folgenden sollen die Implementierungen der verschiedenen, auf den Mutexen implementierten Operationen beschrieben werden.

Sind alle Teile des Detektors deaktiviert, werden nur die entsprechenden Operationen auf den sync.(RW)Mutex Locks ausgeführt. Eine weitere Sammlung von Daten findet in diesem Fall nicht statt.

Ist der Detektor nicht vollständig deaktiviert, werden bei den verschiedenen Operationen Daten gesammelt, um die Detektion von Deadlocks zu ermöglichen. Bei dieser Sammlung werden die Datenstrukturen, welche in 5.1 beschrieben wurden, verändert oder neu erzeugt um Lock-Bäume wie in 2.3.2 beschrieben zu erzeugen.
Diese Vorgänge sollen im Folgenden beschrieben werden.

5.2.1 (R-)Lock

(R-)Lock wird verwendet, um die Locks zu beanspruchen.
Das Locking von RW-Mutexes unterscheidet sich von dem Locking der Mutexe nur dadurch, das isRLock in dem Mutex entsprechend gesetzt wird, und 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 überprüft, ob für für die entsprechende Routine bereits ein Lock-Baum existiert, die Routine also bereits initialisiert worden ist. Ist dies nicht der Fall wird die Routine initialisiert und damit ein leere Baum für diese Routine erzeugt.
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 es momentan beansprucht nicht die selben sind oder wenn sie die selben sind und 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. 5.5). Anschließend wird die abschließende Detektion gestartet und das Programm 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 nebenläufigen 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 so-

mit keine Dependencies bilden, muss der Lock-Baum der Routine nicht verändert werden. 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. Dazu wird jeder Situation ein Key zugeordnet, die sich durch eine xOr Verknüpfung der Speicheradressen des aktuellen und des zuletzt gelockten Locks in der Routine gebildet wird. Existiert der Schlüssel nicht, kam die Situation noch nicht vor. Andernfalls wird überprüft, ob die momentane Situation tatsächlich einer bereits vorkommenden Situation entspricht, welche in der `dependencyMap` als mit dem Key verknüpften Value gespeichert werden. Existiert die Situation 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. Außerdem werden in `context` die Informationen darüber gespeichert, in welcher Datei und Zeile die Lock-Operation ausgeführt worden ist. Ist es nicht deaktiviert, wird außerdem der Call-Stack der Routine zur Zeit der Operation gespeichert.

Sobald die Aktualisierung der Datenstrukturen abgeschlossen wurde wird die eigentliche Beanspruchung des Locks ausgeführt und in.

5.2.2 (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

5.2.3 (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` der Routine entfernt und das eigentliche Lock wieder freigegeben.

5.3 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 weitere abschließende Detektion von Deadlocks möglich.

Die Detektion ist, bis auf die Betrachtung von RW-Locks identisch mit der periodischen Detektion in UNDEAD (Kap. 3). Für die Detektion wird von jeder Routine nur `curDep` betrachtet,

also diejenige Dependency, 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. In 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.b), (2.3.2.d) und (2.3.2.f) immer noch gelten. Ist dies der Fall, so wird überprüft, ob die Kette einen Kreis bildet, also ob die Formeln (2.3.2.c) und (2.3.2.e) 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 vorerst 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 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, sodass 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 Programm nicht in einem lokalen Deadlock befindet.

5.4 Abschließende Detektion

Die abschließende Detektion wird am Ende des Programs durchgeführt um potenzielle Deadlock zu finden, auch wenn diese in dem Durchlauf nicht tatsächlich aufgetreten sind. Sie muss vom Nutzer manuell in seinem Code gestartet werden, nachdem die Ausführung des eigentlichen Programms abgeschlossen wurde. Sie wird in diesem Fall nur ausgeführt, wenn sie nicht deaktiviert ist und die Anzahl der Routinen, die in dem Programm vorkamen, sowie die Anzahl der einzigartigen Dependencies mindestens zwei ist.
Wie bereits in Kap. 3 beschrieben, verläuft die Detektion ähnlich wie die periodischen Detektion ab. 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 potenziellen Deadlocks beendet, sondern erst, wenn alle möglichen Pfade betrachtet wurde.

5.5 Meldung gefundener Deadlocks

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

5.5.1 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
```

5.5.2 Deadlocks

Bei einem tatsächlichen oder potenziellen Deadlock werden die Locks, welche in dem Zyklus, welcher den Deadlock erzeugt vorkommen, mit den Positionen ihrer Initialisierung und ihren (Try-)(R-)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
```

5.6 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)

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

Im Folgenden soll der in Kap. 5 beschriebene und in [10] implementierte Detektor „Deadlock-Go“ analysiert und mit dem in 4 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 [11] implementiert sind. Diese bestehen zum einen aus verschiedenen Standartsituationen, zum anderen auf Programmen, die aus tatsächlichen Programmen übernommen wurden. Diese werden aus Gobench [8], einer Benchmark Suite für Parallelitätsfehler in Go ausgewählt.

6.1 Funktionale Analyse

Man analysiere die Programme zuerst funktional, d.h. man überprüfe, ob sie zu den gewünschten Ergebnissen führen.
Man betrachte zuerst die Standardsituationen. In der folgenden Tabelle 6.1 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 Zyklen der Länge zwei sucht (2). Deadlock-Go hingegen ist in der Lage auch diese potenziellen 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 potenzielles Deadlock ausgegeben. sasha-s hingegen erkennt nicht, dass solch ein Deadlock nicht auftreten kann und gibt somit eine false-positive Mitteilung über ein potenzielles 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. Ist die Erkennung von doppeltem Locking nicht deaktiviert, ist Deadlock-Go also auch in der Lage solche Deadlocks durch doppeltes Locking zu erkennen (5).
Bei verschachtelten Routinen ist keines der Tools in der Lage das potenzielle 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 potenzielle 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 potenzieller 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	potenzielles Deadlock durch zyklisches Locking von zwei Locks	Ja	Ja
1.2	Kein potenzielles Deadlock, da die Locks nicht zyklisch sind	Ja	Ja
2	potenzielles 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	potenzielles 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	potenzielles Deadlock mit RW-Lock in zwei Routinen (R1: x.RLock → y.Lock, R2: y.Lock → x.Lock)	Ja	Ja
9.2	potenzielles Deadlock mit RW-Lock in zwei Routinen (R1: x.RLock → y.Lock, R2: y.RLock → x.Lock)	Ja	Ja
9.3	Kein potenzielles Deadlock mit RW-Lock in zwei Routinen (R1: x.RLock → y.RLock, R2: y.RLock → x.RLock)	Ja	Nein
9.4	Kein potenzielles Deadlock mit RW-Lock in zwei Routinen (R1: x.Lock → y.RLock, R2: y.RLock → x.Lock)	Ja	Nein
9.5	Kein potenzielles Deadlock mit RW-Lock in zwei Routinen (R1: x.RLock → y.Lock, R2: y.RLock → x.RLock)	Ja	Nein
9.6	Kein potenzielles Deadlock mit RW-Lock in zwei Routinen (R1: x.Lock → y.RLock, R2: y.RLock → x.RLock)	Nein	Nein
10.1	Kein potenzielles Deadlock, wegen Lock von RW-Locks als Gate-Locks.	Ja	Nein
10.2	potenzielles 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 6.1: 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 auf Grund von R-Lock-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). Deadlock-Go 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 GoBench [8] betrachtet werden. Dabei handelt es sich um Programme mit etwa 100 Zeilen und 1 bis 3 Routinen. Die folgende Tabele 6.2 gibt für jedes Programm an, ob es von den Detektoren erkannt wurde.

ID	**Typ**	**Deadlock-Go**	**sasha-s**
Cockroach584	Doppeltes Locking	Ja	Ja
Cockroach9935	Doppeltes Locking	Ja	Ja
Cockroach6181	Zyklische Locking mit RW-Locks	Ja	Ja
Cockroach7504	Zyklische Locking	Ja	Ja
Cockroach10214	Zyklische Locking	Ja	Ja
Etcd5509	Doppeltes Locking	Nein	Nein
Etcd6708	Doppeltes Locking	Ja	Ja
Etcd10492	Doppeltes Locking	Ja	Ja
Kubernetes13135	Zyklisches Locking	Ja	Ja
Kubernetes30872	Zyklisches Locking	Ja	Ja
Moby4951	Zyklisches Locking	Ja	Ja
Moby7559	Doppeltes Locking	Ja	Ja
Moby17176	Doppeltes Locking	Nein	Nein
Moby36114	Doppeltes Locking	Ja	Ja
Syncthing4829	Doppeltes Locking	Ja	Ja

Tabelle 6.2: Funktionale Analyse der Beispielprogramme

Beide Detektoren waren in der Lage die potenziellen Deadlocks in 13 der 15 Beispielprogramme zu erkennen. Bei zwei der Programm war keiner der beiden Detektoren in der Lage, einen potenziellen Deadlock zu finden. Bei dieses beiden Programmen handelt es sich um Programme, bei denen es bei besonderen Abläufen dazu kommt, das vergessen wird Locks freizugeben, 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. 6.1 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, während Deadlock-Go vor allem bei der Verhinderung von false-positives besser abschneidet als

sasha-s. Diese Situation kommen in dem Beispielen allerdings nicht vor.

6.2 Laufzeitanalyse

Im Folgenden soll der Einfluss der Detektoren auf die Laufzeit eines Programmes betrachtet werden. Dazu werden Programme basierend auf Programm 1.1 aus Tabelle 6.1 verwendet. Da die Laufzeit stark von der Anzahl der Lock und der Anzahl der Routinen abhängt, werden vier verschiedene Fälle analysiert. Als erstes wird Programm 1.1 mit zwei Routinen und zwei Locks (2×2) verwendet. Zudem wird das Programm für die restlichen drei Fälle verändert. Für den zweiten Fall werden ebenfalls zwei Routinen verwendet, allerdings mit 100 Locks (2×100). Als drittes werden 100 Routinen mir zwei Locks (100×2) und zum Schluss 100 Routinen mit 100 Locks (100×100) betrachtet. Dabei erhält man folgende Werte:

	2×2 [ms]	2×100 [ms]	100×2 [ms]	100×100 [ms]
ohne Detektor	0.006	0.010	0.110	0.244
Deadlock-Go ohne periodischer Detektion	0.070+0.002 0.072	0.740 + 41.325 42.065	2.175 + 0.260 2.535	41.532 + 113687.142 113728.674
Deadlock-Go mit periodischer Detektion	0.072+0.002 0.079	0.804+41.585 42.368	2.178 + 0.263 2.441	60.509 + 106799.742 106860.251
Sasha-s	0.062	12.622	11.541	507.628

Tabelle 6.3: Gesammtlaufzeiten der einzelnen Programm für bzw. ohne die Detektoren. Für Deadlock-Go sind dabei sowohl die Laufzeit des eigentlichen Programms, die Laufzeit der abschließenden Detektion, sowie die Gesammtlaufzeit angegeben.

Betrachtet man nur die Laufzeit des eigentlichen Programms ohne die abschließende Detektion in Deadlock-Go ergibt sich damit folgendes Bild:

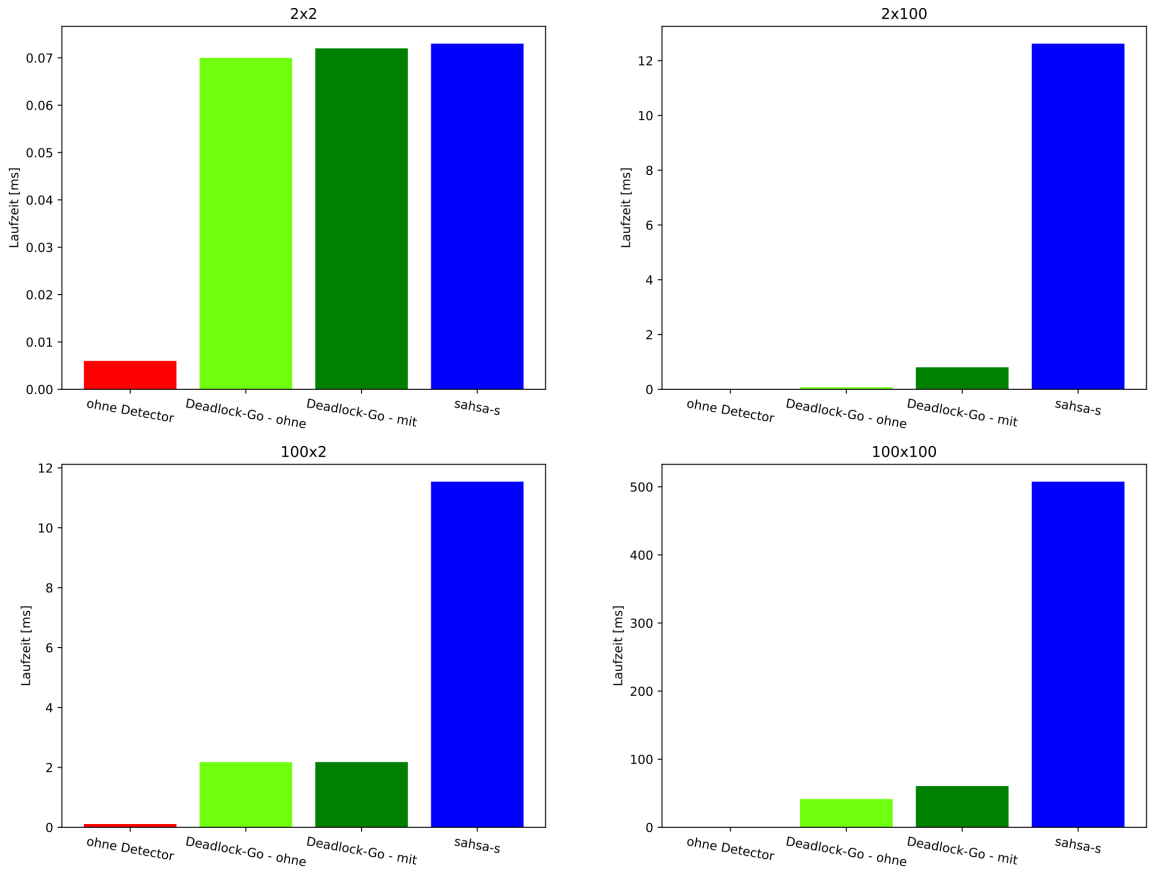

Abbildung 6.1: Laufzeiten des eigentlichen Programms (ohne abschließende Detektion in Deadlock-Go) der betrachteten Situationen ohne Detektor, mit Deadlock-Go und sasha-s. “ohne” und “mit” für Deadlock-Go bedeutet dabei, dass der Detektor ohne bzw. mit periodischer Detektion abgelaufen ist.

Es ist klar zu sehen, dass beide Detektoren die Laufzeit des eigentlichen Programms verlängern. Allerdings hat sasha-s dabei, sobald entweder die Anzahl der Locks oder Routinen größer wird, einen deutlich größeren Einfluss auf die Laufzeit. Dies liegt daran, dass Deadlock-Go während der Laufzeit des eigentlichen Programms nur Daten sammelt. Die periodische Detektion läuft dabei in einer eigenen Routine und hat daher nur einen geringen Einfluss auf die Laufzeit. Sasha-s hingegen sucht bereits während dem Ablauf des Programm nach potentiellen Deadlocks, und nutzt dabei die selben Routinen, mit denen das eigentliche Programm abläuft. Aus diesem Grund hat sasha-s einen signifikant größeren Einfluss auf die Laufzeit des eigentlichen Programms.

Betrachtet man die gesamte Laufzeit, also mit der abschließenden Detektion in Deadlock-Go, gibt sich allerdings folgendes Bild:

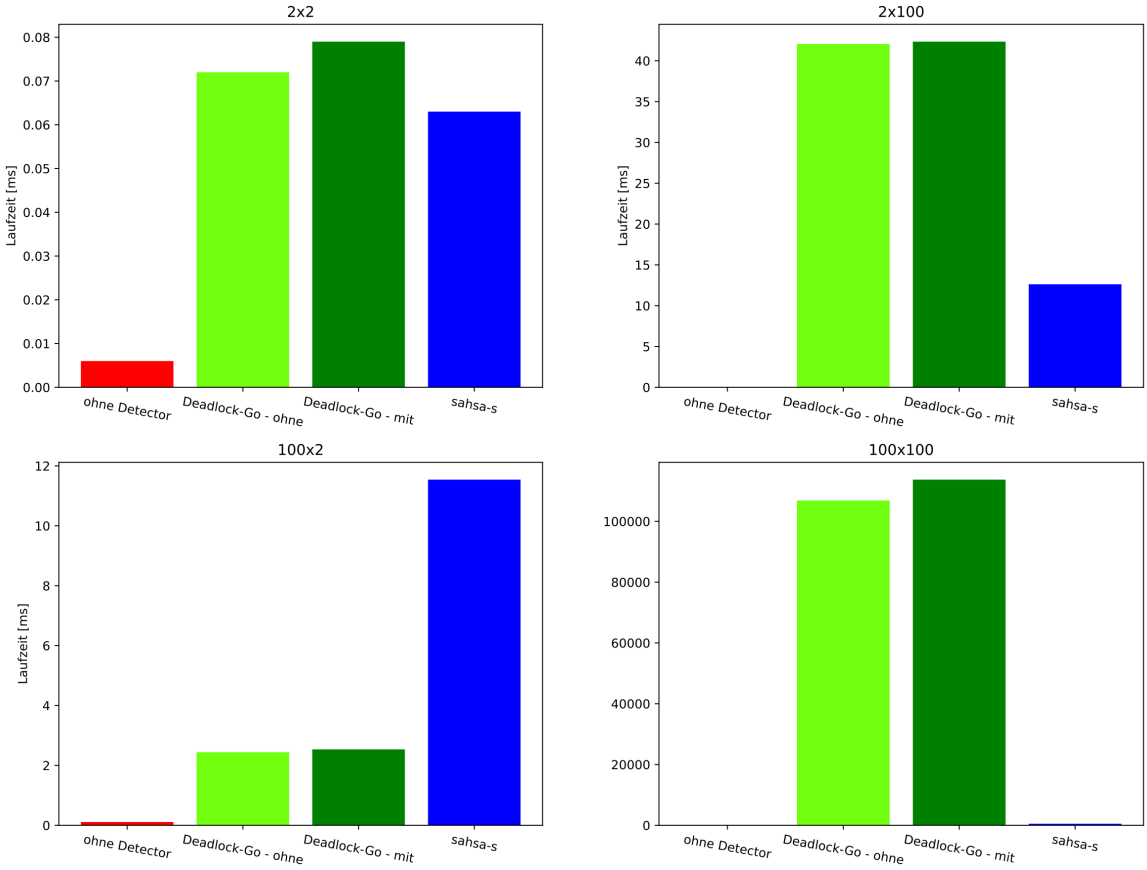

Abbildung 6.2: Gesammtlaufzeiten der betrachteten Situationen ohne Detektor, mit Deadlock-Go und sasha-s. “ohne” und “mit” für Deadlock-Go bedeutet dabei, dass der Detektor ohne bzw. mit periodischer Detektion abgelaufen ist.

Bei dem Vergleich der beiden Detektoren bezüglich ihrer Laufzeit kommt es hier nun auf das betrachtete Programm an. Sind sowohl die Anzahl der Routinen als auch die Anzahl der Dependencies klein, ist der Unterschied zwischen den beiden Detektoren relativ gering. Sasha-S hat einen klaren Vorteil, wenn es sich um ein Programm mit vielen Locks und damit vielen Dependencies handelt. Da sasha-s immer nur Pfade der Länge zwei in seinem Lock-Graphen betrachtet, währen Deadlock-Go alle möglichen Pfade in allen Kombinationen von Lock-Bäumen betrachtet, muss Deadlock-Go deutlich mehr Pfade analysieren, und hat daher eine signifikant längere Laufzeit. Deadlock-Go hat dafür einen Vorteil, wenn die Anzahl der Routinen groß, die Anzahl der Locks aber kein ist. Da die Lock-Bäume in Deadlock-go in diesem Fall alle sehr klein sind, verliest sasha-s seine Vorteil, dass es nur Pfade der Länge 2 betrachtet. Allerdings überwiegt dieser wieder deutlich, wenn sowohl die Anzahl der Routinen als auch die Anzahl der Lock groß ist.

Beide Detektoren sind in der Lage, die Detektion zu deaktivieren, so dass sie dann nur noch einen sehr geringe Verlängerung der Laufzeiten haben. Nachdem ein Programm zu Ende entwickelt wurde, und alle potenziellen 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.

7 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 und Routinen abhängt. Nachdem die Detektoren in der Softwareentwicklung verwendet wurden, um potenzielle 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] R. Agarwal, L. Wang und S. D. Stoller, „Detecting Potential Deadlocks with Static Analysis and Runtime Monitoring“, in *Proceedings of the Parallel and Distributed Systems: Testing and Debugging (PADTAD) Track of the 2005 Haifa Verification Conference*, Ser. Lecture Notes in Computer Science, Bd. 3875, Springer-Verlag, Nov. 2006, S. 191–207. Adresse: http://dx.doi.org/10.1007/11678779_14.

[2] 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.

[3] Y. Cai und W. K. Chan, „MagicFuzzer: Scalable deadlock detection for large-scale applications“, in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, S. 606–616. DOI: 10.1109/ICSE.2012.6227156.

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

Referenzen

[10] E. Kassubek, *Deadlock-Go*, https://github.com/ErikKassubek/Deadlock-Go, 2021. (besucht am 09. 08. 2022).

[11] E. Kassubek, *DeadlockExamples*, https://github.com/ErikKassubek/BachelorProjektGoDeadlockDetection/tree/main/DeadlockExamples, 2021. (besucht am 09. 08. 2022).