

Sound Dynamic Deadlock Prediction in Linear Time

Florian Rudaj

*Fakultät für Informatik und
Wirtschaftsinformatik, Hochschule Karlsruhe*

3. Dezember 2023

Inhaltsverzeichnis

1	Einleitung	1
2	Nebenläufige Programme	2
3	Die Vorhersage von Deadlocks	10
4	Dynamische Deadlock-Analyse	12
5	Sync-preserving Deadlocks	16
6	Fazit	24

1 Einleitung

Sei es in Betriebssystemen, Web-Servern oder Echtzeitsystemen – in fast jeder modernen Applikation ist die Nebenläufigkeit (engl. „Concurrency“) von Tasks von großer Bedeutung. Beispielsweise könnte ohne Nebenläufigkeit ein Betriebssystem nicht mehr als ein Programm gleichzeitig laufen lassen. Ein IoT-Gerät müsste für die Netzwerkkommunikation die Aufnahme von Sensordaten stoppen. Allein diese Beispiele zeigen die Relevanz von Nebenläufigkeit in heutigen Systemen.

Nichtsdestotrotz gibt es neben den Vorteilen der Nebenläufigkeit, wie z.B. dem Performancegewinn durch das Verteilen von Tasks auf mehreren Prozessorkernen, auch Probleme, die mit ihr einhergehen – sog. „Concurrency Bugs“. Grundlage dieser Concurrency Bugs sind die nebenläufigen Zugriffe verschiedener Tasks auf dieselben Ressourcen. Diese Zugriffe können dazu führen, dass ein Programm, welches die Tasks nebenläufig ausführt, abstürzt oder sogar in einer Endlosschleife verweilt, ohne tatsächlich Arbeit zu verrichten.

Es gibt mehrere Typen von Concurrency Bugs. Der wohl bekannteste von allen ist der sog. „Deadlock“. Dieser kommt beispielsweise zustande, wenn zwei Threads bereits eine Ressource reserviert haben, die der jeweils andere zum gleichen Zeitpunkt ebenso reservieren will. Allgemeiner bezeichnet man den Zustand eines Programms als Deadlock, wenn jeder Thread des Programms darauf wartet, dass ein anderer Thread Ressourcen freigibt. Wenn ein Deadlock vorkommt stürzt das Programm in Folge ab, da kein Thread mehr Arbeit verrichtet. Die Forschung beschäftigt sich schon sehr lange damit, Deadlocks zuverlässig und effizient zu erkennen bzw. vorherzusagen. Dabei wurden statische und dynamische Lösungsansätze entwickelt. Statische Lösungsansätze versuchen einen Deadlock anhand des Quellcodes zu erkennen, wohingegen dynamische Ansätze die Ausführung des Programms analysieren. Probleme dieser Lösungsansätze waren jedoch häufig, dass sie entweder zu viele false positives (Deadlocks, die aber keine sind) angezeigt oder eine zu schlechte Laufzeit haben[1].

Die vorliegende Arbeit beschäftigt sich jedoch mit dem kürzlich erbrachten und bemerkenswerten Fortschritt „Sound Dynamic Deadlock Prediction in Linear Time“[1]. Nachdem in den nachfolgenden Kapiteln zuerst auf Concurrency Bugs und Deadlocks im Besonderen eingegangen wird, werden Methoden vorgestellt, die es ermöglichen, in linearer Laufzeit sowie mit sehr hoher Präzision Deadlocks vorherzusagen[1].

2 Nebenläufige Programme

Man bezeichnet ein System als nebenläufig, wenn es dazu in der Lage ist mehrere Tasks simultan abzuarbeiten. Bei dieser Definition wird oft fälschlicherweise angenommen, dass die Nebenläufigkeit der Parallelität gleichzusetzen ist. Allerdings heißt Nebenläufigkeit jedoch nicht gleichzeitig, dass die Tasks auch parallel bearbeitet werden. Ein System kann nämlich auch dann schon als nebenläufig bezeichnet werden, wenn es lediglich mehrere Tasks in einer Warteschlange hält und zwischen diesen hin und her wechselt, bevor diese vollständig bearbeitet wurden[2]. Genau so macht es ein Betriebssystem, welches auf einer Maschine mit nur einem Prozessorkern läuft. Es definiert Zeitschlitze, in denen jeweils ein Prozess Zugriff auf den Prozessorkern hat, bevor der nächste Prozess an der Reihe ist. Ob der jeweilige Prozess fertig bearbeitet wurde, ist keine Bedingung dafür, dass der nächste Prozess bearbeitet werden kann. Unfertig bearbeitete Prozesse werden wieder in die Warteschlange eingereiht.

Allerdings sind Systeme, die Tasks parallel bearbeiten, auch gleichzeitig nebenläufig. Angenommen die Maschine, auf der unser Betriebssystem läuft, hat nun mehrere Prozessorkerne zur Verfügung. Damit hat unser Betriebssystem die Möglichkeit so viele Tasks gleichzeitig bearbeiten zu lassen, wie es Prozessorkerne gibt. Mit der gewachsenen Anzahl der Prozessorkerne und die dadurch ermöglichte Parallelität verfügt das System auch über die Fähigkeit die gleiche Anzahl an Tasks in viel kürzerer Zeit zu bearbeiten. Die parallele Bearbeitung bedeutet also einen Performancegewinn.

Dies ist jedoch nicht der einzige Vorteil den Nebenläufigkeit mit sich bringt. Man stelle sich zum Beispiel einen Browser vor über den der User durch Eingabe einer URL zu einer Website gelangt. Nach Eingabe der URL und betätigen der Enter-Taste bemerkt der User, dass das Laden der Website sehr lange dauert und entscheidet sich stattdessen eine andere Website zu besuchen. Die Eingabe der anderen URL ist dem User nur möglich, weil das Laden der Website und das Erkennen von User-Eingaben im UI parallel, oder zumindest nebenläufig, bearbeitet wird. Ansonsten würde der User warten müssen, bis die Website vollständig geladen ist oder ein Fehlerfall auftritt. Neben dem Performancegewinn bietet die Nebenläufigkeit also auch noch den Vorteil der Reaktionsfähigkeit.

Nebenläufigkeit kann auch dabei helfen eine Anwendung skalierbarer zu machen. Ein Beispiel dafür sind Webserver, die die Anfragen von mehreren hundert Clients gleichzeitig bearbeiten müssen. Durch die Nebenläufigkeit können auf unterschiedlichen Prozessorkernen mehrere Anfragen parallel verarbeitet werden. Falls eine dieser Anfragen beispielsweise einen länger dauernden I/O-Zugriff ausführt kann während der dadurch auftretenden War-

tezeit die Anfrage eines anderen Clients bearbeitet werden. Nebenläufigkeit stellt also auch sicher, dass die zur Verfügung stehenden Ressourcen möglichst effizient ausgeschöpft werden.

Doch der Einsatz von Nebenläufigkeit bringt auch Gefahren mit sich. Ein Beispiel dafür ist der folgende in der Programmiersprache Go geschriebene Code. Die Funktion „raceCondition“ verfügt über eine Variable „data“, die auf den Wert 0 geprüft wird. Wenn der Wert 0 ist, soll der Wert zurückgegeben werden. Wenn der Wert jedoch nicht 0 ist, wird -1 zurückgegeben. In den Zeilen 3 – 5 startet die Funktion einen neuen Thread, der die Variable „data“ um eins erhöht. Bei Ausführung der Funktion „raceCondition“ ergeben sich drei mögliche Ausgabewerte. Die beiden offensichtlichsten Ausgabewerte sind 0 und -1. Wenn der nebenläufige Thread aus den Zeilen 3-5 es noch nicht geschafft „data“ um eins zu erhöhen wird 0 ausgegeben. Der Wert -1 wird zurückgegeben, wenn der nebenläufige Thread es vor der Prüfung auf 0 geschafft hat „data“ hochzuzählen. Der dritte Fall ist weniger offensichtlich und gleichzeitig sehr problematisch. Er tritt ein, wenn die Prüfung von „data“ auf den Wert 0 erfolgreich ist, jedoch zwischen der Prüfung und der Rückgabe von „data“ der nebenläufige Thread das Inkrement um eins durchgeführt hat. In diesem Fall gibt „raceCondition“ 1 zurück.

```
1 func raceCondition() int {
2     data := 0
3     go func() {
4         data++
5     }()
6     if data == 0 {
7         fmt.Println("")
8         return data
9     }
10    return -1
11 }
12
13 func main() {
14     var outputs []int
15     for i := 0; i < 100000; i++ {
16         outputs = append(outputs, raceCondition())
17     }
18     print_shares([]int{-1, 0, 1}, outputs)
19 }
```

Listing 1: Beispiel einer Race Condition (abgeleitet von [2])

Um zu beweisen, dass bei jeder Ausführung völlig unklar ist welches Ergebnis zurückgegeben wird, wird in der Main-Funktion die Race Condition 100.000 mal ausgeführt. Die Funktion „print_shares“ in Zeile 18 nimmt die Liste der Ausgaben von „raceCondition“ entgegen und gibt die Häufigkeit sowie Anteile der Ergebnisse -1, 0 und 1 in der Konsole aus. Für eine Ausführung dieses Programms ergibt sich beispielhaft die folgende Ausgabe:

```
Counts of -1: 118
Share of -1: 0.118000 Percent

Counts of 0: 99876
Share of 0: 99.876000 Percent

Counts of 1: 6
Share of 1: 0.006000 Percent
```

Listing 2: Konsolenausgabe bei Race Condition

Hier ist zu erkennen, dass vor allem der Fall, bei dem „raceCondition“ 0 zurückgibt, auftritt. Aber auch die anderen beiden Fälle treten auf. Besonders ärgerlich ist, dass der Fall, bei dem 1 zurückgegeben wird, auftritt und gleichzeitig sehr selten ist. Damit das Programm in die Zeilen 7 und 8 gelangen darf, muss die Variable „data“ den Wert 0 enthalten. Allerdings hat der nebenläufige Thread durch die simulierte Arbeit in Zeile 7 manchmal genügend Zeit den Inhalt von „data“ vor der Rückgabe zu ändern.

Ein solcher Concurrency Bug nennt sich Race Condition. Genauer gesagt handelt es sich hier um einen sog. „Data Race“, da es darauf ankommt, welcher Thread seine Arbeit mit den Daten in der Variable „data“ zuerst verrichtet. In Unit-Tests wird ein solcher Concurrency Bug nicht erkannt, da er, wie in der Konsolenausgabe zu erkennen ist, dafür viel zu selten auftritt. In Produktionsumgebungen jedoch, wo eine solche Funktion tausende Male am Tag ausgeführt wird, kann ein solcher Bug auftreten. Dieser ist dann aber schwer oder gar nicht zu replizieren, da er so selten vorkommt. Bei Concurrency Bugs kann es auch vorkommen, dass sie erst ab einem gewissen Skalierungsniveau auftreten. Beispielsweise wird ein Dienst, der einen solchen Bug enthält, 1-mal pro Tag aufgerufen. Wenn die Wahrscheinlichkeit, zu der der Concurrency Bug auftritt, die gleiche ist wie in unserem obigen Beispiel kann dieser Bug jahrelang unentdeckt bleiben. Wenn die Anfragen auf den Dienst jedoch zunehmen, tritt dieser Bug immer häufiger auf und es kommt dadurch wohlmöglich zu Inkonsistenzen in den Daten und Ausgaben des Dienstes.

Um Race Conditions zu verhindern, können Zugriffe auf Variablen, die sich während der Ausführung eines bestimmten Abschnitts im Code nicht verändern dürfen, synchronisiert werden. Dies geschieht durch die Verwendung eines sog. „Mutex“. Nebenläufige Threads können sich durch das Erwerben eines Locks auf einen Mutex den Zugriff auf eine Variable, die zu dem Mutex gehört, reservieren[2]. Angewandt auf das obige Beispiel sieht der Code der Funktion „raceCondition“ folgendermaßen aus:

```
1 func raceCondition() int {
2     data := 0
3     var dataAccess sync.Mutex
4     go func() {
5         dataAccess.Lock()
6         data++
7         dataAccess.Unlock()
8     }()
9     dataAccess.Lock()
10    if data == 0 {
11        print("")
12        return data
13    }
14    dataAccess.Unlock()
15    return -1
16 }
```

Listing 3: Durch Mutex verhinderte Race Condition (abgeleitet von [2])

Hier ist in den Zeilen 5 - 7 zu sehen, dass der nebenläufige Thread den Mutex zuerst sperrt, bevor er „data“ inkrementiert. Anschließend wird der Mutex wieder freigegeben, da die Änderung an der geteilten Variable vollständig durchgeführt wurde. Der Main-Thread hingegen sperrt in Zeile 9 den Mutex, um sicher zu gehen, dass sich der Inhalt von „data“ in den Zeilen 10 – 12 nicht verändert.

Wenn nun die gleiche Main-Funktion wie bereits in dem vorigen Code-Beispiel ausführt, werden folgende Ergebnisse in die Konsolenausgabe geschrieben:

```
Counts of -1: 2063
Share of -1: 2.063000 Percent
```

```
Counts of 0: 97937
Share of 0: 97.937000 Percent
```

```
Counts of 1: 0
Share of 1: 0.000000 Percent
```

Listing 4: Konsolenausgabe ohne Race Condition

Wie anhand der Konsolenausgabe zu erkennen ist, gibt es keinen einzigen Fall mehr, bei dem sich der Wert von „data“ nach der Prüfung auf 0 doch noch zu 1 ändert. Das Reservieren und Freigeben eines Mutex sichert also die Konsistenz einer Variable innerhalb einer kritischen Sektion. Der Begriff „kritische Sektion“ bezeichnet dabei einen Abschnitt im Code, der Lese- oder Schreiboperationen auf eine zwischen Threads geteilte Variable ausführt.

Allerdings bringt diese Methode der Konsistenzsicherung weitere Probleme mit sich. Eines dieser Probleme ist der sog. Deadlock. Er tritt dann auf, wenn alle Threads eines Programms zur Bearbeitung ihres Tasks einen Mutex reservieren müssen, der aber bereits von einem anderen Thread gehalten wird. Das bedeutet, dass im Falle eines Deadlocks alle Threads aufeinander warten und keine Arbeit verrichtet wird. Ein einfaches Beispiel eines Deadlocks zeigt der folgende Code:


```

1 func simple_deadlock() {
2     a := 0
3     b := 0
4     var aMutex sync.Mutex
5     var bMutex sync.Mutex
6     go func() {
7         aMutex.Lock()
8         bMutex.Lock()
9         //simulate work
10        if a == 1 && b == 1 {
11            a++
12            b++
13        }
14        bMutex.Unlock()
15        aMutex.Unlock()
16    }()
17    bMutex.Lock()
18    aMutex.Lock()
19    //simulate work
20    if a == 0 && b == 0 {
21        b++
22        a++
23    }
24    aMutex.Unlock()
25    bMutex.Unlock()
26 }

```

Listing 5: Beispiel für einen Deadlock

Im obigen Codebeispiel gibt es zwei Threads. Beide Threads benötigen die Variablen a und b innerhalb einer If-Bedingung. Um zu sichern, dass die Werte, auf die die Variablen geprüft wurden, innerhalb der If-Bedingung gleichbleiben, gibt es für jede Variable einen Mutex. Nun ist es so, dass die beiden Threads eine unterschiedliche Reihenfolge haben den Mutex zu reservieren. Dies kann dazu führen, dass der nebenläufige Thread erst den Mutex für a und der Main-Thread den Mutex für b reserviert. Im nächsten Schritt versuchen beide Threads den Mutex, der vom jeweils anderen Thread bereits reserviert wurde, selbst zu reservieren. Da dies jedoch nicht möglich ist kommt es zu einem Deadlock, da alle Threads warten. Infolgedessen stürzt das Programm ab.

Natürlich können nicht nur mit zwei Threads Deadlocks erzeugt werden.

An einem Deadlock können beliebig viele Threads beteiligt sein. Das Beispiel aus Listing 5 lässt sich auf drei Threads erweitern, indem eine weitere Variable und ein weiterer Mutex hinzugefügt wird. Ein solches Beispiel ist in Listing 6 zu sehen. Hier muss jeder Thread zwei Locks auf einen Mutex reservieren, bevor dieser seine Arbeit verrichten kann. Die Besonderheit hierbei ist, dass der zweite Lock in jedem Thread in einem der anderen Threads der erste Lock ist. Beispielsweise wird in Thread 1 in Zeile 11 als zweiter Mutex „lockB“ reserviert, wohingegen in Thread 2 in Zeile 21 „lockB“ als erster Mutex reserviert wird. Angenommen das Scheduling der drei Threads führt dazu, dass jeweils die erste Lock-Operation ausgeführt wird. Dann sind anschließend alle Mutexes reserviert und keiner der Threads kann im nächsten Schritt die benötigte Ressource reservieren. Schlussendlich kommt es wieder zu einem Deadlock und das Programm stürzt ab.

```

1  func deadlock3Threads() {
2      a := 0
3      b := 0
4      c := 0
5      lockA := &sync.Mutex{}
6      lockB := &sync.Mutex{}
7      lockC := &sync.Mutex{}
8      //Thread 1
9      go func() {
10         lockA.Lock()
11         lockB.Lock()
12         if a == 0 && b == 0 {
13             a++
14             b++
15         }
16         lockB.Unlock()
17         lockA.Unlock()
18     }()
19     //Thread 2
20     go func() {
21         lockB.Lock()
22         lockC.Lock()
23         if b == 0 && c == 0 {
24             b++
25             c++
26         }
27         lockC.Unlock()
28         lockB.Unlock()
29     }()
30     //Thread 3
31     lockC.Lock()
32     lockA.Lock()
33     if c == 0 && a == 0 {
34         c++
35         a++
36     }
37     lockA.Unlock()
38     lockC.Unlock()
39 }

```

Listing 6: Deadlock mit drei beteiligten Threads

3 Die Vorhersage von Deadlocks

Um den Absturz von Programmen durch Deadlocks zu verhindern, wurde in der Vergangenheit viel Forschung betrieben, um Deadlocks vorherzusagen. Dies geschieht durch die Analyse des Programms. Bei den Analysemethoden für Deadlocks unterscheidet man zwischen der statischen und der dynamischen Deadlock-Analyse[1].

Die statische Deadlock-Analyse betrachtet ausschließlich den Quellcode für die Vorhersage von Deadlocks. Es gibt statische Ansätze, die dazu in der Lage sind, die Abwesenheit von Deadlocks zu beweisen. Allerdings sind statische Analysemethoden nicht gut skalierbar und zeigen in Programmen oft Deadlocks an, wo jedoch keine sind[1].

Eine Analysemethode, die solche „False-Positives“ anzeigt, wird auch „unsound“ genannt. Mit dem Begriff „sound“ wird in der Deadlock-Analyse eine Methode bezeichnet, wenn diese keine false positives vorhersagt.

Die dynamische Deadlock-Analyse betrachtet eine beispielhafte Ausführung des Programms in Form eines Trace als Basis für die Vorhersage. Die Ereignisse innerhalb des Trace werden auf bestimmte Weise umgeordnet, sodass Deadlocks erkannt werden können. Im Gegensatz zur statischen Deadlock-Analyse wird hierbei nicht das Ziel verfolgt, einen Beweis für die Abwesenheit von Deadlocks zu erbringen. Stattdessen wird sich darauf konzentriert durch Umordnungen von Traces theoretisch mögliche Deadlocks zu entdecken. Dynamische Methoden erkennen zwar nicht alle, aber dennoch viele Deadlocks. Außerdem sind sie dadurch, dass sie keinen Beweis erbringen müssen, skalierbarer und liefern nur wenige bis keine False-Positives. Ein solcher dynamischer Ansatz wird in der vorliegenden Arbeit im Kapitel „Sync-preserving Deadlocks“ vertieft beschrieben. Er erkennt die meisten Deadlocks, ist sehr gut zu skalieren und ist darüber hinaus sound[1].

Die gerade beschriebenen Analysemethoden müssen in der Lage sein, verschiedene Typen von Deadlocks vorherzusagen. Zum einen gibt es den sog. „Resource-Deadlock“, bei dem alle Threads des Programms auf die Freigabe einer Ressource warten, die von einem anderen Thread des Programms bereits reserviert wurde. Das Deadlock-Beispiel aus dem vorherigen Kapitel ist ein solcher Resource -Deadlock.

Weiterhin gibt es den sog. „Communication-Deadlock“, der dann auftritt, wenn alle Threads darauf warten mit einem anderen Thread kommunizieren zu können. Der folgende Code enthält einen solchen Communication-Deadlock:

```

1 func communication_deadlock() {
2     ch := make(chan int)
3
4     go func() {
5         data := <-ch
6         fmt.Println("Received:", data)
7     }()
8
9     data := <-ch
10    fmt.Println("Received:", data)
11 }

```

Listing 7: Communication Deadlock

Ein Channel in Go ist ein Objekt, welches ermöglicht, Daten zwischen Threads für nebenläufige Berechnungen auszutauschen. In Zeile 2 des obigen Codebeispiels wird ein solcher Channel erstellt. Der nebenläufige Thread in den Zeilen 4 - 7 möchte Daten aus dem Channel empfangen. Da es keinen Sender gibt, der Daten in dem Channel platziert, blockiert der nebenläufige Thread. Da der Main-Thread in Zeile 9 ebenfalls Daten aus demselben Channel empfangen will, aber kein Sender vorhanden ist, blockiert auch dieser. Dies führt dazu, dass sich beide Threads des Programms in einem Wartezustand befinden und ein Communication-Deadlock auftritt.

4 Dynamische Deadlock-Analyse

Bei der dynamischen Deadlock-Analyse wird versucht, theoretisch mögliche Deadlocks anhand eines Trace des Programms zu erkennen. Um einen solchen Trace zu erhalten, muss das zu analysierende Programm zuerst instrumentiert werden. Damit ist gemeint, dass das Programm um Code erweitert wird, der aufzeichnet, in welchem Thread und zu welchem Zeitpunkt ein Mutex reserviert bzw. freigegeben wird.

Wie ein Trace aus dem Code entsteht, lässt sich anhand von Listing 5 und Listing 8 gut erkennen. Hierbei ist zu beachten, dass die Variable `a` in Listing 5 analog zu der Variablen `y` in Listing 8 ist. Genau so verhält es sich auch mit `b` in Listing 5, welches in Listing 8 durch `x` repräsentiert wird. Es wird angenommen, dass der Code in Listing 5 ausgeführt wird und dabei instrumentiert ist. Die Instrumentierung zeichnet in diesem Fall nur die Reservierungen und Freigaben, die auf einen Mutex ausgeführt werden, auf. Lese- und Schreiboperationen werden nicht aufgezeichnet, da es sich in Listing 8 um einen Trace für die „Lock-Dependency-Methode“ handelt. Diese Methode beachtet das Auslesen und Beschreiben von Variablen für die Vorhersage des Deadlocks nicht. Für andere Methoden der Deadlock-Vorhersage, wie bspw. bei den „Sync-preserving Deadlocks“ in Kapitel 5, müssen auch Lese- und Schreiboperationen im Trace aufgezeichnet worden sein.

In der Aufzeichnung in Listing 8 ist jede Reservierung bzw. jedes Lock auf einen Mutex durch ein „acq“, also einer Acquire-Operation gekennzeichnet. Analog dazu werden Freigaben bzw. Unlocks durch ein „rel“, also einer Release-Operation repräsentiert.

Wenn nun nach der Definition von Variablen und Mutexes zuerst der nebenläufige Thread aus Listing 5 (Zeilen 6 - 16) komplett ausgeführt wird, werden zuerst die beiden Locks auf die Mutexes der Variablen `a` und `b` durchgeführt. Diese werden aufgezeichnet und im Trace aus Listing 8 anschließend als die Zeilen 1 und 2 dargestellt. Außerdem werden sie dem Trace T1 zugeordnet. T1 ist in Listing 5 der nebenläufige Thread, der diese Operationen ausführt. Der nebenläufige Thread prüft als nächstes die Bedingungen für einen If-Block. Diese werden nicht aufgezeichnet, da es Leseoperationen sind. Da die If-Bedingung nicht erfüllt wurde, werden als nächstes die beiden Mutexes wieder freigegeben. Dies wird wieder aufgezeichnet und im Trace aus Listing 8 in den Zeilen 4 und 5 vermerkt. Der nebenläufige Thread ist damit fertig ausgeführt worden. Als nächstes wird der Main-Thread aus Listing 5 (Zeilen 17-25), der in Listing 8 als T2 bezeichnet wird, ausgeführt. Auch hier werden die Lock-Operationen in derselben Reihenfolge im Trace aufgezeichnet, wie sie ausgeführt wurden. In Listing 8 werden sie durch die

Zeilen 6 und 7 dargestellt. Ebenso werden die Lese- und Schreiboperationen, die im produktiven Code vonstatten gehen, nicht aufgezeichnet. Die danach durchgeführten Freigaben der Mutexes, werden aufgezeichnet und in Listing 8 in den Zeilen 8 und 9 dargestellt.

Nach der Ausführung des Programms und der damit verbundenen Aufzeichnung des Trace kann dieser dazu verwendet werden sog. „Deadlock-Patterns“ zu erkennen. Deadlock-Patterns sind Muster, die darauf hindeuten, dass ein Trace einen potenziellen Deadlock enthält.

Formell betrachtet ist ein Deadlock-Pattern der Größe k eines Trace δ eine Sequenz $D = \langle e_0, e_1, \dots, e_{k-1} \rangle$ mit k eigenständigen Threads t_0, \dots, t_{k-1} und k individuellen Locks l_0, \dots, l_{k-1} . Jede der in der Sequenz D vorhandenen Events e_i stellt dabei eine Acquire-Operation auf einen Lock aus der Menge der bereits gehaltenen Locks zum Zeitpunkt i dar. Weiterhin gilt, dass keiner der Threads Locks hält, die auch von einem anderen Thread gehalten werden.

Ein Deadlock-Pattern zu erkennen ist eine notwendige, aber keine hinreichende Bedingung für die Detektion eines Deadlocks[1]. Das bedeutet, dass zwar ein Deadlock-Pattern für die Vorhersage eines Deadlocks vorhanden sein muss, jedoch nicht hinter jedem Deadlock-Pattern ein tatsächlicher Deadlock stehen muss.

Ein einfacher Algorithmus, um dynamisch Deadlocks zu erkennen ist die „Lock-Dependency-Methode“. Sie basiert auf der Idee von sog. „Lock-Graphen“. Ein Lock-Graph baut sich aus den Lock-Operationen nebenläufiger Threads auf gemeinsame Ressourcen bzw. Mutexes auf. Ein Lock auf eine Ressource stellt hierbei einen Knoten im Graph dar. Eine Kante von einem Lock-Knoten zu einem anderen Lock-Knoten entsteht dann, wenn ein nebenläufiger Thread einen der beiden Locks bereits hält und den anderen reservieren möchte. Nachdem der Lock-Graph aus dem Trace aufgebaut wurde, wird nach Zyklen im Lock-Graph gesucht. Der Zyklus im Lock-Graphen ist das Deadlock-Pattern dieses Algorithmus. Wenn mindestens ein Zyklus vorhanden ist, dann geht die Lock-Dependency-Methode davon aus, dass ein Deadlock vorhanden ist[3]. Man betrachte den folgenden Trace:

```

1   y -> x
2   x -> y

```

Listing 9: Lock-Graph zum obigen Trace[3]

```

1       T1           T2
2   acq(y)
3   acq(x)
4   rel(x)
5   rel(y)
6
7           acq(x)
8           acq(y)
9           rel(y)
           rel(x)

```

Listing 8: Trace, der einen Deadlock enthält[3]

Die Operationen `acq` (acquire) und `rel` (release) stehen hier für die Reservierung und die Freigabe eines Mutexes. Es ist zu sehen, dass Thread T1 erst `y` und dann `x` reserviert. Unser Graph bekommt dadurch die Knoten `x` und `y`, sowie eine Kante von `y` nach `x`. In Thread T2 wird erst `x` und dann `y` reserviert. Wir haben also wieder die Knoten `x` und `y`, die aber bereits in unserem Lock-Graph existieren. Hinzu kommt aber eine Kante von `x` nach `y`, da T2 `x` bereits hält, bevor `y` reserviert wird. Daraus ergibt sich der folgende Lock-Graph:

Der Lock-Graph wird im nächsten Schritt auf Zyklen untersucht. Wie im obigen Beispiel leicht zu erkennen ist, gibt es einen Zyklus zwischen den Knoten `x` und `y`. Die Lock-Dependency-Methode sagt hier also einen Deadlock voraus. Dies ist auch richtig, da der obige Trace auch so verlaufen kann, dass im ersten Schritt T1 `y` reserviert und im zweiten Schritt T2 `x` reserviert. Der nächste Schritt für beide Threads wäre dann die Reservierung des Mutex, der vom jeweils anderen Thread bereits gehalten wird. Da dies für beide Threads jedoch nicht möglich ist, resultiert diese Umordnung des Trace in einem Deadlock[3].

Allerdings sagt die Lock-Dependency-Methode auch häufig False-Positives vorher. So wie in dem folgenden Beispiel:

Hier werden alle Reservierungen und Freigaben der Ressourcen innerhalb von T1 ausgeführt. T2 benötigt keine Ressourcen, die zu reservieren wären. Aus dem obigen Trace resultiert, wie in Listing 11 zu erkennen ist, der gleiche Lock-Graph wie schon im ersten Trace[3].

	T1	T2
1		
2	acq(y)	
3	acq(x)	
4	rel(x)	
5	rel(y)	
6	acq(x)	
7	acq(y)	
8	rel(y)	
9	rel(x)	

Listing 10: Trace, bei dem nur ein Thread Arbeit verrichtet[3]

```

1  y -> x
2  x -> y

```

Listing 11: Lock-Graph zu obigem Trace[3]

Da in diesem Lock-Graph ein Zyklus steckt sagt die Lock-Dependency-Methode wieder einen Deadlock vorher. Dieser kann in der Realität jedoch niemals auftreten, da nur T1 Ressourcen reserviert und freigibt.

Das Vorhersagen von False-Positives ist problematisch, da Softwareentwickler anschließend viel Zeit mit der Suche und Verhinderung des Deadlocks verbringen würden, ohne dass dieser überhaupt existiert. Aus diesem Grund wurde eine weitere dynamische Analysemethode entwickelt, die keine False-Positives vorhersagt und dennoch einen Großteil der potenziellen Deadlocks erkennt. Auf diese Methode wird im nachfolgenden Kapitel eingegangen.

5 Sync-preserving Deadlocks

„Synchronization-preserving Deadlocks“ (auch „Sync-preserving Deadlocks“) sind eine Teilmenge aller vorhersagbaren Deadlocks. Sie stellen auch die Deadlocks dar, die in der Praxis am häufigsten vorkommen. Durch den SPD(Sync-preserving-Deadlock)-Algorithmus ist es möglich, sie ohne die Gefahr von False-Positives und darüber hinaus effizient sowohl in einem Offline- als auch in einem Online-Szenario anhand eines Beispiel-Trace zu erkennen. Mit Offline- und Online-Szenario ist hierbei gemeint, dass der SPD-Algorithmus sowohl auf Traces einer vergangenen Ausführung als auch auf Traces zur Laufzeit angewendet werden kann[1].

Um den Algorithmus zur Vorhersage von Sync-preserving Deadlocks nachvollziehen zu können, muss zuerst die Idee hinter Sync-preserving Deadlocks erläutert werden. Ein Sync-preserving Deadlock ist ein Deadlock-Pattern, welches der SPD-Algorithmus versucht innerhalb eines Trace zu erkennen. Dafür wird im ersten Schritt versucht, aus dem Trace ein sog. „Sync-preserving Reordering“ zu bilden. Um festzustellen, ob es ein Sync-preserving Reordering von einem Trace gibt, muss zuerst ein „Correct Reordering“ gefunden werden. Ein Correct Reordering, also eine korrekte Umordnung eines Trace, kann durch die Einhaltung folgender Regeln erzeugt werden[1]:

- Lock-Semantic: Der originale Trace muss wohlgeformt sein. Das bedeutet, dass bei jedem von einem Thread t_0 durch ein Event e_0 reservierten Mutex l zuerst eine Freigabe des Mutex von t_0 durch ein Event e_1 ausgeführt werden muss, bevor ein weiterer Thread t_1 durch ein weiteres Event e_2 den Mutex l wieder reservieren kann[1]
- Subset: Es besteht ausschließlich aus Teilen des originalen Trace[1]
- Thread-Order: Die Reihenfolge der Operationen innerhalb eines Threads wurde nicht verändert[1]
- Last-Write: Für jede Leseoperation auf einer bestimmten Variable existiert in der Umordnung ebenfalls die letzte zuvor ausgeführte Schreiboperation auf diese Variable[1]

Diese Regeln stellen sicher, dass die resultierende(n) Umordnung(en) eines Trace genauso auch von dem Programm generiert werden könnten, welches den ursprünglichen Trace ausgegeben hat. Dieser Schritt generiert also viele mögliche Ausführungen eines Programms[1].

0	T1	T2	T3	T4
1	acq(L1)			
2	rel(L1)			
3		acq(L2)		
4		acq(L3)		
5		w(z)		
6		rel(L3)		
7		rel(L2)		
8				acq(L1)
9				w(y)
10				r(z)
11				rel(L1)
12	acq(L3)			
13	w(x)			
14	r(y)			
15	rel(L3)			
16			acq(L3)	
17			r(x)	
18			acq(L2)	
19			rel(L2)	
20			rel(L3)	

Listing 12: Trace δ , der einen Sync-preserving Deadlock enthält

Der Trace δ in Listing 12 enthält einen Sync-preserving Deadlock und damit auch ein Correct Reordering $\rho = e_1e_2e_3e_8e_9e_{12}..e_{15}e_{16}e_{17}$. Dieses ist nochmal in Listing 13 als Trace zu sehen. Durch Überprüfung der oben genannten Regeln ist einfach zu erkennen, dass ρ ein Correct Reordering ist. δ unterliegt der Lock-Semantic, da nach jeder Acq-Operation zuerst eine Release-Operation folgt, bevor von einem anderen Thread eine Acquire-Operation ausgeführt wird. ρ ist Teil von δ , da alle Events aus δ stammen. Weiterhin wird die Ordnung der Operationen innerhalb der Threads beibehalten. Dies ist einfach anhand der durchgehend aufsteigenden Nummerierung der Events in ρ zu erkennen. In Umordnungen, die nicht einer solchen Sortierung unterliegen muss diese Eigenschaft natürlich genauer überprüft werden. Schlussendlich existiert auch für jede Leseoperation die letzte Schreiboperation auf derselben Variable. Die einzigen Leseoperationen in ρ sind e_{17} , bei dem x gelesen wird, und e_{14} , bei dem y gelesen wird. Zugehörig zu e_{17} ist die Schreiboperation e_{13} , sowie e_9 zu e_{14} . Damit wären alle drei Kriterien für ein Correct Reordering erfüllt[1].

0	T1	T2	T3	T4
1	e1: acq(L1)			
2	e2: rel(L1)			
3		e3: acq(L2)		
4				e8: acq(L1)
5				e9: w(y)
6	e12: acq(L3)			
7	e13: w(x)			
8	e14: r(y)			
9	e15: rel(L3)			
10			e16: acq(L3)	
11			e17: r(x)	
12		e4: acq(L3)		
13			e18: acq(L2)	

Listing 13: Trace des Correct Reordering ρ samt Deadlock-Pattern (rot gekennzeichnet)

Im nächsten Schritt muss ρ darauf geprüft werden, ob es auch ein Sync-preserving Reordering ist. Ein Correct Reordering ist genau dann sync-preserving, wenn auch die Acquire-Events auf denselben Lock in der gleichen Reihenfolge wie in δ beibehalten werden. Beispielsweise würde diese Regel durch eine Umordnung verletzt werden, die e_8 vor e_1 ausführt, da in δ die Acquire-Operationen auf L1 in der Reihenfolge e_1, e_8 ausgeführt werden. In ρ jedoch kann die Ordnung der Acquire-Events, wie auch schon die Ordnung der Operationen innerhalb der Threads, sehr einfach anhand der Nummerierung der Events geprüft werden. Da die Nummerierung der Events in ρ aufsteigend sortiert ist, kann die Ordnung der Acquire-Events auf die jeweiligen Locks nicht verändert worden sein. Auch hier gilt, dass diese Charakteristik in Umordnungen, die nicht einer solchen Sortierung unterliegen, natürlich genauer untersucht werden muss. Da in ρ alle Acquire-Events auf den jeweiligen Lock jedoch noch genau so geordnet sind wie in δ ist ρ ein Sync-Preserving Reordering[1].

Anschließend kann geprüft werden, ob nach der Ausführung von ρ ein Deadlock-Pattern existiert. Dazu wird untersucht, welche Threads welche Locks bereits halten und welche sie anschließend reservieren wollen. Nach der Ausführung von ρ kann observiert werden, dass T2 bereits L2 hält und im nächsten Schritt e_4 L3 reservieren möchte. T3 hält aber bereits L3 und möchte in seiner nächsten Operation e_{18} L2 besetzen, welches aber schon von T2 gehalten wird. Da alle Operationen von T1 bereits durchgeführt wurden und keine Operationen aus T4 in ρ vorhanden sind befinden sich an diesem

Punkt alle Threads in einem Wartezustand, was folglich in einem Deadlock endet. Somit wurde das Deadlock-Pattern $D = \langle e_4, e_{18} \rangle$ gefunden und δ enthält einen Sync-Preserving Deadlock[1].

Nicht jedes Correct Reordering ist sync-preserving. Ein Beispiel dafür ist $\rho_2 = e_3..e_7e_8..e_{11}e_1e_2$, welches in Listing 14 nochmal als Trace dargestellt wird. Anhand der Regeln Lock-Semantic, Subset, Thread-Order und Last-Write kann bestätigt werden, dass ρ_2 ein Correct Reordering ist:

- Lock-Semantic: Der originale Trace δ ist wohlgeformt. Dieses Wissen stammt bereits aus der Überprüfung von ρ
- Subset: Alle in ρ_2 aufgeführten Events stammen aus δ , somit ist ρ_2 eine Teilmenge von δ
- Thread-Order: Die Reihenfolge der Operationen innerhalb der einzelnen Threads wurde nicht verändert. Dies ist leicht anhand von Listing 14 zu erkennen, da die Nummerierung der Events innerhalb jedes Threads aufsteigend sortiert ist.
- Last-Write: Die einzige Leseoperation in ρ_2 ist e_{10} . Die letzte vorherige Schreiboperation ist e_5 , welche in ρ_2 ebenfalls vorhanden ist und vor e_{10} ausgeführt wird. Somit hält ρ_2 auch die Last-Write-Regel ein.

Da alle oben aufgelisteten Regeln eingehalten werden kann ρ_2 als Correct Reordering bezeichnet werden.

	T1	T2	T3	T4
0				
1		e3: acq(L2)		
2		e4: acq(L3)		
3		e5: w(z)		
4		e6: rel(L3)		
5		e7: rel(L2)		
6				e8: acq(L1)
7				e9: w(y)
8				e10: r(z)
9				e11: rel(L1)
10	e1: acq(L1)			
11	e2: rel(L1)			

Listing 14: Trace des Correct Reordering ρ_2

Nun gilt es zu prüfen, ob ρ_2 auch sync-preserving ist. Dazu muss, wie in einem vorherigen Abschnitt bereits genauer beschrieben, überprüft werden,

ob sich die Reihenfolge der Acquire-Operationen von verschiedenen Threads auf denselben Lock verändert hat. In ρ_2 ist es tatsächlich der Fall, dass die Reihenfolge von Acquire-Operationen auf L1 im Vergleich zu δ vertauscht wurde. In δ ist es so, dass T1 L1 reserviert bevor T4 L1 reserviert. In ρ_2 ist dies jedoch genau umgekehrt, weshalb ρ_2 nicht die Kriterien für ein Sync-preserving Correct Reordering erfüllt[1].

Nun ist klar, wie Sync-preserving Deadlocks anhand von Correct Reorderings erkannt werden können. Allerdings gilt es noch zu klären, wie effizient Correct Reorderings gefunden werden können. Hierzu müssen im Folgenden zuerst einige Notationen erklärt werden:

- δ beschreibt den original aufgezeichneten Trace[1]
- $Events_\delta$ beschreibt die Menge der Events im originalen Trace δ [1]
- Die „Trace-Order“ $\leq_{tr}^\delta \subseteq Events_\delta \times Events_\delta$ beschreibt die absolute Ordnung der die Events in δ unterliegen[1]
- Die „Thread-Order“ \leq_{TO}^δ ist eine partielle Ordnung über $Events_\delta$, sodass $e \leq_{TO}^\delta e'$ dann und nur dann wenn $thread(e) = thread(e')$ und $e \leq_{tr}^\delta e'$. Es wird $e \leq_{TO}^\delta e'$ geschrieben wenn $e \leq_{tr}^\delta e'$ aber $e \neq e'$ [1]
- Die „Reads-From-Funktion“ rf_δ ist eine Zuordnung von Lese- zu Schreiboperationen in δ . Für jede Leseoperation e auf der Variablen x gilt, dass $e' = rf_\delta(e)$ die letzte Schreiboperation auf x ist, sodass $e' \leq_{TO}^\delta e$ [1]
- Die „Match-Funktion“ $match_\delta$ gibt bei Eingabe eines Acquire- bzw. Release-Events das nach der Lock-Semantic dazugehörige vorherige Acquire- bzw. erste nachfolgende Release-Event zurück. Zum Beispiel seien $e, e' \in Events_\delta$ zwei aufeinander folgende Events auf dem gleichen Lock, sodass $op(e) = acq(l)$ und $op(e') = rel(l)$, dann gilt $e' = match_\delta(e)$ bzw. $e = match_\delta(e')$, wenn $thread(e) = thread(e')$ [1]
- Die „Predecessor-Funktion“ $pred_\delta(S)$ bezeichnet die Menge der unmittelbaren Vorgänger-Events innerhalb des Threads indem sich das Event befindet. Die formale Definition lautet wie folgt: $pred_\delta(S) = \{e \in Events_\delta \mid \exists f \in S, e \leq_{tr}^\delta f \text{ and } \forall e' \leq_{tr}^\delta f, e' \leq_{TO}^\delta e\}$ [1]

Um ein Correct Reordering, welches zu einem Deadlock führt, zu finden, müssen zwei Probleme gelöst werden. Das erste Problem ist die Bestimmung der Events, die in das Correct Reordering einfließen sollen. Das zweite Problem ist die Bestimmung der Reihenfolge der ausgewählten Events. Für Sync-preserving Deadlocks lässt sich die Suche nach einem Correct Reordering auf

das Überprüfen, ob eine wohldefinierte Menge an Events, die nicht die Events im Deadlock Pattern enthält, reduziert werden. Außerdem kann diese Menge effizient berechnet werden. Diese Menge wird auch als „Sync-preserving Closure“ oder, in der Funktionsschreibweise, als SPClosure_δ bezeichnet[1].

Wenn δ der originale Trace ist und $S \subseteq \text{Events}_\delta$, dann definiert sich $\text{SPClosure}_\delta(S)$ durch die kleinste Menge S' , sodass $S \subseteq S'$ und für alle $e, e' \in \text{Events}_\delta$ gilt, dass $e \leq_{TO}^\delta e'$ oder $e = \text{rf}_\delta(e')$. Wenn $e' \in S'$, dann gilt auch $e \in S'$. Außerdem gilt für jeden Lock l und zwei unterschiedliche Events $e, e' \in \text{Events}_\delta$ mit $\text{op}(e) = \text{op}(e') = \text{acq}(l)$, wenn $e \leq_{tr}^\delta e'$ dann $\text{match}_\delta(e) \in S'[1]$

Um die Sync-preserving Closure besser zu verstehen wird sie anhand des Beispiels aus Listing 15 vorgeführt. Trace δ_2 aus Listing 15 enthält zwar das Deadlock-Pattern $D = \langle e_4, e_{11} \rangle$. Dies liegt daran, dass T2 durch e_2 und e_3 erst z und y reserviert und in e_4 x reservieren will. Wenn T1 aber schon x in e_{10} reserviert hat und nun in e_{11} y, welches bereits von T2 gehalten wird, reservieren will, könnte es nach der Lock-Dependency-Methode zu einem Deadlock kommen. Diese Vorhersage ist aber ein False-Positive. Das liegt daran, dass nur die Acquire-Events in der Lock-Dependency-Methode betrachtet werden.

Allerdings gibt es noch „fork“- und „join“-Operationen in δ_2 , die eine gewisse Reihenfolge der Events erzwingen. „fork“ bedeutet im Trace, dass der Thread, der diese Operation ausführt, einen weiteren Thread startet. Dieser ist innerhalb der runden Klammern angegeben. Die „join“-Operation hingegen bewirkt, dass der ausführende Thread auf das Ende des in den runden Klammern angegebenen Thread wartet. In δ_2 verhält sich dies so, dass der Thread T2 entweder komplett vor oder nach, aber nicht während der Ausführung von T1 abgearbeitet werden kann. Dies liegt daran, dass T0 einen sog. „Guard-Lock“ enthält. Angenommen e_8 wurde bereits ausgeführt, bevor e_2 ausgeführt werden konnte. Dann muss T2 für den Lock auf z in e_2 warten und kann daher nicht y in e_3 reservieren, welches im Deadlock-Pattern D von T1 in e_{11} reserviert werden soll. In diesem Fall kann es also zu gar keinem Deadlock kommen. Andersherum angenommen, dass e_2 vor e_8 ausgeführt wurde, gilt andersherum, dass T0 für den Lock auf z in e_8 warten muss und damit nicht in e_9 T1 starten kann, wodurch es ebenso keine Gefahr für einen Deadlock gibt.

	T0	T1	T2
e1.	fork(T2)		
e2.			acq(z)
e3.			acq(y)
e4.			acq(x)
e5.			rel(x)
e6.			rel(y)
e7.			rel(z)
e8.	acq(z)		
e9.	fork(T1)		
e10.		acq(x)	
e11.		acq(y)	
e12.		rel(y)	
e13.		rel(x)	
e14.	join(T1)		
e15.	rel(z)		

Listing 15: Trace δ_2 , der ein Deadlock-Pattern vorweist, aber ein False-Positive darstellt.

Die Sync-preserving Closure ist dazu in der Lage nicht auf das False-Positive-Deadlock-Pattern in δ_2 hereinzufallen. Als Eingabe dient der SPClosure die Menge der Vorgänger-Events aus dem Deadlock, also $\text{pred}_{\delta_2} = S = \{e_3, e_{10}\}$. Es wird im Folgenden also $\text{SPClosure}_{\delta_2}(\{e_3, e_{10}\})$ berechnet. Die $\text{SPClosure}_{\delta_2}(S) = S'$ besteht anfänglich aus der Menge $\{e_3, e_{10}\}$. Da e_3 und e_{10} nun nicht ausgeführt werden können, bevor die vorherigen Events aus dem jeweiligen Thread ausgeführt wurden, müssen alle dafür notwendigen Vorgänger-Events nach der Thread-Order zur Closure hinzugefügt werden. Dazu gehören auch e_9 und e_8 , da die Events in T1 ohne diese nicht durchgeführt werden könnten. Daraus resultiert die Menge $S' = \{e_3, e_2, e_{10}, e_9, e_8\}$. In S' sind nun e_2 und e_8 enthalten, welches beide Acquire-Operationen auf den Lock z sind. Nach der Lock-Semantic muss zusätzlich noch e_7 hinzugefügt werden, da $e_3 \leq_{tr}^{\delta} e_8$ und $e_7 = \text{match}_{\delta_2}(e_3)$, somit gilt $S' = \{e_3, e_2, e_{10}, e_9, e_8, e_7\}$. Nun hat e_7 wieder Vorgänger-Operationen, die ausgeführt werden müssen bevor e_7 ausgeführt werden kann. Nach der Thread-Order kommen zu S' also noch e_6, e_5 , und e_4 hinzu. Anschließend gilt $S' = \{e_3, e_2, e_{10}, e_9, e_8, e_7, e_6, e_5, e_4\}$. Durch das letzte Hinzufügen von Events ist e_4 in S' gekommen. Allerdings ist das Ziel der Sync-preserving Closure ein Sync-preserving Correct Reordering zu finden, welches zu einem Sync-preserving Deadlock führt. Da e_4 nun aber Teil des Correct Reorderings wäre, und somit nicht mehr zu D führen kann, ist D kein Sync-preserving Dead-

lock. Somit hat die Sync-preserving Closure es geschafft trotz des Deadlock-Patterns kein False-Positive vorherzusagen.

Allerdings kann die Sync-preserving Closure False-Negatives vorhersagen. Wenn das vorliegende Deadlock-Pattern kein Sync-preserving Deadlock, aber dennoch ein Deadlock ist, sagt die Sync-preserving Closure voraus, dass kein Deadlock vorliegt. Dies liegt daran, dass Deadlocks existieren die in der Praxis vorkommen können, jedoch nicht der Klasse der Sync-preserving Deadlocks angehören[1].

6 Fazit

In nebenläufigen Programmen gibt es häufig Ressourcen, die zwischen den einzelnen Threads geteilt werden müssen. Um Inkonsistenzen in diesen Ressourcen zu vermeiden, dürfen diese durch die Verwendung eines Mutex nur einem Thread gleichzeitig zur Verfügung gestellt werden. Der damit einhergehende Lock-Mechanismus kann bei einem unglücklichen Scheduling der Threads dazu führen, dass jeder Thread für die Weiterverarbeitung die Ressourcen eines anderen Threads benötigt, wodurch ein Deadlock entsteht. Ein solcher Deadlock führt zu dem Absturz des Programms. Fehler dieser Art lassen sich sehr schwer reproduzieren, da sie durch einen Nichtdeterminismus entstehen und gegebenenfalls extrem selten vorkommen können. Umso wichtiger sind deshalb Methoden, die solche potenziellen Deadlocks in einem Programm erkennen können[2].

Neben der statischen Deadlock-Analyse, welche ausschließlich den Quellcode analysiert, gibt es die deutlich effizientere dynamische Deadlock-Analyse. In der dynamischen Deadlockanalyse werden Traces des nebenläufigen Programms analysiert. Viele Methoden der dynamischen Deadlock-Analyse, wie z.B. die Lock-Dependency-Methode, haben das Problem, dass sie auch False-Positives vorhersagen[1]. In der Praxis ist dieses Verhalten äußerst unerwünscht. Die Arbeitszeit, die für das Verhindern eines Deadlocks, der gar nicht existiert, aufgewendet wird, erzeugt hohe Kosten.

Glücklicherweise wurde durch die Forschung eine neue Klasse von Deadlocks gefunden - die Sync-preserving Deadlocks. Die meisten in der Praxis vorkommenden Deadlocks sind Sync-preserving Deadlocks. Es existiert ein Algorithmus, mit dem sie sehr effizient in linearer Laufzeit erkannt werden können, ohne dass dabei False-Positives geliefert werden. Der Algorithmus bekommt als Eingabe die Vorgänger-Events (aus demselben Thread) der Events im Deadlock-Pattern, welches überprüft werden soll und gibt an, ob es sich tatsächlich um einen Deadlock handelt. Dafür extrahiert er nach bestimmten Regeln weitere Events aus dem originalen Trace und überprüft, ob ein Event aus dem Deadlock-Pattern ebenfalls extrahiert wurde. Wenn dies der Fall ist handelt es sich nicht um einen Sync-preserving Deadlock. Dies ist jedoch ein False-Negative und es kann sich dennoch um einen Deadlock handeln, der aber nicht der Klasse der Sync-preserving Deadlocks angehört. Wenn jedoch ein Sync-preserving Deadlock erkannt wurde, entspricht dies aber der Aussage, dass dieser Deadlock in der Praxis tatsächlich vorkommen kann[1].

Literatur

1. Tunç, H.C., Mathur, U., Pavlogiannis, A., Viswanathan, M.: Sound dynamic deadlock prediction in linear time. Proc. ACM Program. Lang. **7**(PLDI) (jun 2023)
2. Cox-Buday, K.: Concurrency in Go: Tools and Techniques for Developers. 1st edn. O'Reilly Media, Inc. (2017)
3. Sulzmann, M.: Dynamic deadlock prediction. <https://sulzmann.github.io/AutonomieSysteme/lec-deadlock.html> Accessed: 2023-11-01.