

Bachelorarbeit

Dynamic Analysis of Message-Passing Go Programs

Erik Daniel Kassubek

Gutachter: Prof. Dr. Thiemann

Betreuer: Prof. Dr. Thiemann

Prof. Dr. Sulzmann

Albert-Ludwigs-Universität Freiburg

Technische Fakultät

Institut für Informatik

Lehrstuhl für Programmiersprachen

14. Februar 2023

Bearbeitungszeit

14. 11. 2022 – 14. 02. 2023

Gutachter

Prof. Dr. Thiemann

Betreuer

Prof. Dr. Thiemann

Prof. Dr. Sulzmann

ERKLÄRUNG

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

Ort, Datum

Unterschrift

Zusammenfassung

(TODO: Zusammenfassung (Abstract) schreiben)

Inhaltsverzeichnis

1	Einführung	1
2	Hintergrund	2
3	Tracer	3
3.1	Trace	3
3.2	Instrumenter	8
3.3	Laufzeit	9
4	Analyze	12
4.1	Deadlock durch Mutext	12
4.2	Deadlock durch Channels	14
4.2.1	Hängende ungepufferte Channel-Operationen	14
4.3	Gepufferte Kanäle	19
5	Vergleich mit anderen Detektoren	21
5.1	GFuzz	21
6	Zusammenfassung	22
7	Acknowledgments	23
	Literaturverzeichnis	28

1 Einführung

(TODO: Einführung schreiben)

2 Hintergrund

(TODO: Hintergrund) (TODO: Beschreibung von ht-cat, go-dsp, goker)

3 Tracer

Um ein Program analysieren zu können, wird der Ablauf eines Programmdurchlaufs aufgezeichnet. Dazu wurde ein Tracer implementiert, durch welchem die Channel- und Lock-Operationen, sowie andere Operationen wie Select und das Erzeugen einer neuen Routine, ersetzt, bzw. erweitert werden. Diese führen die eigentlichen Operationen aus und zeichnen gleichzeitig den Trace auf. Die Ersetzung durch die Drop-In Replacements kann dabei automatisch durch einen Instrumenter erfolgen, welche mit Hilfe des Abstract Syntax Trees die Ersetzungen vornimmt.

Anders als in vielen anderen Programmen, welche den Trace von Go-Programmen analysieren, wie z.B. [1] oder [2] wird dabei der Tracer selbst implementiert und basiert nicht auf dem Go-Runtime-Tracer [3]. Dies ermöglicht es, den Tracer genau auf die benötigten Informationen zuzuschneiden und so einen geringeren negativen Einfluss auf die Laufzeit des Programms zu erreichen.

(EXTEND: Tracer Einführung)

3.1 Trace

(DRAFT: Trace) Der Aufbau des Trace basiert auf [4]. Er wird aber um Informationen über Locks erweitert. Der Trace wird für jede Routine separat aufgezeichnet. Außerdem wird, anders als in [4] ein globaler Program-Counter für alle Routinen und nicht ein separater Counter für jede Routine verwendet. Dies ermöglicht es bessere Rückschlüsse über den

genauen Ablauf des Programms zu ziehen. Die Syntax des Traces in EBNF gibt sich folgendermaßen:

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

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

- **signal(t , i)**: In der momentanen Routine wurde eine Fork-Operation ausgeführt, d.h. eine neue Routine mit Id i wurde erzeugt.
- **wait(t , i)**: Die momentane Routine mit Id i wurde soeben erzeugt. Dies ist in allen Routinen außer der Main-Routine das erste Event in ihrem lokalen Trace.
- **pre(t , as)**: Die Routine ist an einer Send- oder Receive-Operation eines Channels oder an einem Select-Statement angekommen, dieses wurde aber noch nicht ausgeführt. Das Argument as gibt dabei die Richtung und den Channel an. Ist $as = x!$, dann befindet sich der Trace vor einer Send-Operation, bei $as = x?$ vor einer Receive-Operation. Bei einem Select-Statement ist as eine Liste aller Channels für die es einen Case in dem Statement gibt. Besitzt das Statement einen Default-Case, wird dieser ebenfalls in diese List aufgenommen.
- **post(t , i , $x!$)**: Dieses Event wird in dem Trace gespeichert, nachdem eine Send-Operation auf x erfolgreich abgeschlossen wurde. i gibt dabei die Id der sendenden

Routine an.

- `post(t, i, x?, t')`: Dieses Event wird in dem Trace gespeichert, nachdem eine Receive-Operation des Channels x erfolgreich abgeschlossen wurde. i gibt dabei die Id der sendenden Routine an. t' gibt den Zeitstempel an, welcher bei dem Pre-Event der sendenden Routine galt. Durch die Speicherung der Id und des Zeitstempels der sendenden Routine bei einer Receive-Operation lassen sich die Send- und Receive-Operationen eindeutig zueinander zuordnen.
- `post(t, default)`: Wird in einem Select-Statement der Default-Case ausgeführt, wird dies in dem Trace der entsprechenden Routine durch $post(t, default)$ gespeichert.
- `close(t, x)`: Mit diesem Eintrag wird das schließen eines Channels x in dem Trace aufgezeichnet.
- `lock(t, y, b, c)`: Der Beanspruchungsversuch eines Locks mit id y wurde beendet. b gibt dabei die Art der Beanspruchung an. Bei $b = r$ war es eine R-Lock Operation, bei $b = t$ eine Try-Lock Operation und bei $b = tr$ ein Try-R-Lock Operation. Bei einer normalen Lock-Operation ist $b = -$. Bei einer Try-Lock Operation kann es passieren, dass die Operation beendet wird, ohne das das Lock gehalten wird. In diesem Fall wird c auf 0, und sonst auf 1 gesetzt.
- `unlock(t, y)`: Das Lock mit id y wurde zum Zeitpunkt t wieder freigegeben.

Um diesen Trace zu erzeugen, werden die Standartoperation aud Go durch Elemente des Tracers ersetzt. Die Funktionsweisen dieser Ersetzungen sind im folgenden angegeben. Dabei werden nur solche Ersetzungen angegeben, welche direkt für die Erzeugung des Traces notwendig sind. Zusätzlich werden noch weitere Ersetzungen durchgeführt, wie z.B. die Ersetzung der Erzeugung von Mutexen und Channel von den Standardvarianten zu den Varianten des Tracers. Hierbei wird auch die Größe jedes Channels gespeichert. Dies werden in der Übersicht zur Vereinfachung nicht betrachte. Auch werden in der Übersicht nur die Elemente betrachtet, die für die Durchführung der Operation und dem Aufbau des Traces

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

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

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

figure

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

Abb. 1: Beispielprogramm für Tracer

Trace:

```
[[signal(1,2), signal(2,3), signal(3,4), pre(4,a?,default), post(5,default)]
[wait(8,2), lock(9,1,-,1), pre(10,x!), post(16,2,x!,1), unlock(17,1)]
[wait(11,3), pre(12,y!), post(13,3,y!,1), pre(14,x?), post(15,2,x?,10,1)]
[wait(6,4), pre(7,y?), post(18,3,y?,12,1)]]
```

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

3.2 Instrumenter

(DRAFT: Instrumenter) Um den Trace zu erzeugen, müssen verschiedene Operationen durch Funktionen des Tracers ersetzt bzw. erweitert werden. Wie man an dem Beispiel

unschwer erkennen kann, besitzt die Version mit dem Tracer einen deutlich längeren Programmcode, was bei der Implementierung zu einer größeren Arbeitslast führen kann. Da sich der Tracer auch negativ auf die Laufzeit des Programms auswirken kann, ist es in vielen Situationen nicht erwünscht, ihn in den eigentlichen Release-Code einzubauen, sondern eher in eine eigenständige Implementierung, welche nur für den Tracer verwendet werden. Um dies zu automatisieren wurde ein zusätzliches Programm implementiert, welches in der Lage ist, den Tracer in normalen Go-Code einzufügen. Die Implementierung, welche ebenfalls in [5] zur Verfügung steht, arbeitet mit einem Abstract Syntax Tree. Bei dem Durchlaufen dieses Baums werden die entsprechenden Operationen in dem Programm erkannt, und durch ihre entsprechenden Tracer-Funktionen ersetzt bzw. ergänzt. Neben dem Ersetzen der verschiedenen Operationen werden außerdem einige Funktionen hinzugefügt. Zu Beginn der Main-Funktion des Programms wird der Tracer initialisiert. Zusätzlich wird eine zusätzliche Go-Routine gestartet, in welcher ein Timer läuft. Ist dieser abgelaufen, wird die Analyse gestartet, auch wenn das Programm noch nicht vollständig durchlaufen ist. Dies führt dazu, dass auch Programme, in welchen ein Deadlock aufgetreten ist, analysiert werden können. Endet das Programm in der vorgegebenen Zeit, wird der Analyzer nach der Beendigung des Programms gestartet. **(EXTEND: Instrumenter)**

3.3 Laufzeit

Instrumenter Zuerst soll die Laufzeit des Instrumenters betrachtet werden. Es ist erwartbar, dass sich die Laufzeit linear in der Anzahl der Ersetzungen in dem AST, also der Anzahl der Mutex- und Channel-Operationen verhält. Dies bestätigt sich auch durch die Messung der Laufzeit des Programms (vgl. Abb. 2)

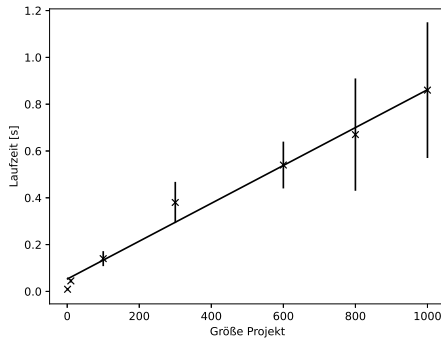


Abb. 2: Laufzeit des Instrumenters

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

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

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

Tab. 1: Laufzeit des Instrumenters für ausgewählte Programme

dort gemessenen Werte befinden sich in Tabelle 1. Gerade in Abhängigkeit von der Anzahl der Ersetzungen, stimmen die hier gemessenen Werte mit denen in Abb. 2 gut überein, während es bei den anderen Parametern größere Abweichungen gibt. Dies bestätigt dass der dominante Faktor für die Laufzeit des Programms die Anzahl der Ersetzungen in dem AST ist, und die Laufzeit linear von dieser abhängt.

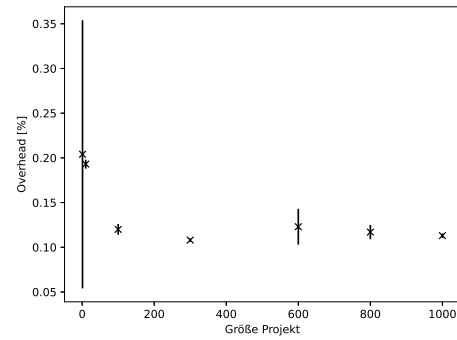


Abb. 3: Prozentualer Overhead des Tracers ohne Analyse

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

4 Analyze

Das folgende Kapitel soll sich nun mit der Analyse des in Kap 3 erstellten Trace befassen. Sie befasst sich dabei mit der Erkennung potentieller Deadlocks durch die vorkommenden Mutexe und Routinen. **(EXTEND: Mehr zur Einführung)**

4.1 Deadlock durch Mutex

(DRAFT: Deadlock Mutex Draft) Als erstes sollen Deadlocks betrachtet werden, welche nur von (RW-)Mutexen erzeugt werden. Dabei wird vor allem zyklischen Locking betrachtet, bei dem sich mehrere Routinen gegenseitig blockieren. Abb. 4 zeigt ein Beispiel, in welchem es zu zyklischem Locking kommen kann. Routine 0 und Routine 1 können dabei gleichzeitig ausgeführt werden. Man betrachte den Fall, in dem Zeile 7 und 14 gleichzeitig ausgeführt werden, also Lock *y* von Routine 0 und Lock *x* von Routine 1 gehalten wird. In diesem Fall kann in keiner der Routinen die nächste Zeile ausgeführt werden, da das jeweilige Locks, welches beansprucht werden soll bereits durch die andere Routine gehalten wird. Da sich diese Situation auch nicht von alleine auflösen kann, blockiert das Programm, befindet sich also in einem zyklischen Deadlock.

Da solche Situationen nur in ganz besonderen Situationen auftreten (in dem obigen Beispiel müssen Zeilen 7 und 14 genau gleichzeitig ausgeführt werden, ohne dass Zeile 8 oder 15 ausgeführt werden), muss ein Detektor, welcher vor solchen Situationen warnen soll, nicht nur tatsächliche Deadlocks, sondern vor allem potenzielle, also nicht tatsächlich aufgetretene

```

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

```

Abb. 4: Beispielprogramm zyklisches Locking

Deadlocks erkenne. Die Erkennung der potenziellen Deadlocks basiert hierbei auf iGoodLock [6] und UNDEAD [7]. Dabei wird ein Lockgraph aufgebaut. Dieser Speichert die in dem Programm vorkommenden Knoten, sowie ihre Abhängigkeiten. Dies bedeutet, dass die Knoten des Graphen gerade die (RW-)Locks representieren. Es gibt dabei genau dann eine Kante von Knoten x nach y , wenn das Lock y beansprucht wird, während das Lock x gerade von der selben Routine gehalten wird. Eine genauere Erklärung der Implementierung des Locks findet sich in [8]. **(TODO: ist des erlabut, oder soll ich es nochmal komplett beschreiben)**

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

Nachdem der Trace einer Routine durchlaufen wurde, wird überprüft ob sich noch Elemente

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

4.2 Deadlock durch Channels

Im folgenden sollen Deadlocks betrachtet werden, welche durch die Verwendung von Channels entstehen. Dabei sollen verschiedene Szenarien betrachtet werden, die auf das Auftreten von Deadlocks hindeuten können. **(TODO: Betrachtete Szenarien beschreiben) (TODO: Einschränkungen beschreiben, z.B. nur unbuffered channels) (TODO: select)**

4.2.1 Hängende ungepufferte Channel-Operationen

Man betrachte das Programm in Abb. 5. Es gibt in diesem zwei mögliche Ausführungspfade. Man betrachtet zuerst den Fall, in dem 1 mit 3 synchronisiert. Da eine go-Routine automatisch abgebrochen wird, wenn die Main-Routine terminiert, entsteht hierbei kein Deadlock. Anders ist es, wenn 1 mit 2 synchronisiert. In diesem Fall wird die Main-Routine blockiert, ohne dass es eine Möglichkeit gibt, dass sie sich wieder befreit. Es kann also,

```

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

```

Abb. 5: Beispielprogramm mit hängendem Channel

abhängig davon, ob 1 oder 2 die Nachricht erhält zu einem Deadlock kommen. Was allerdings beide Fälle gemeinsam haben ist, dass sie eine Channel-Operation besitzen, welche zwar gestartet, allerdings nie ausgeführt wird. In diesen Fällen gibt es in dem Trace eine Channel-Information, welche ein Pre- aber kein Post-Event besitzt. Solch eine Situation bezeichnen wir als hängenden Channel. Durch eine einfache traversierung des Traces können solche Situationen erkannt werden. Solche hängenden Channel können auf einen potenziellen oder tatsächlich auftretenden Channel hindeuten. Es sei allerdings dazu gesagt, dass eine solche hängende Operation nicht immer zu einem Deadlock führen muss. Man betrachte dazu das Beispiel in Abb. 6. Da auf dem Channel `x` nie gesendet wird, kommt es in Zeile 3 zu

```

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

```

Abb. 6: Hängender Channel ohne Deadlock

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

Wir sind nun also in der Lage solche Situationen zu erkennen. Um solche Situationen verhindern zu können, kann es aber auch sinnvoll sein für Channels, bei welchen eine solche Situation aufgetreten ist alle potenziellen Kommunikationspartner anzugeben, um dem Nutzer

bei der Suche und Beseitigung solcher Situationen zu helfen. Für Abb. 5 soll also angegeben werden können, dass die Send-Operation in 1 sowohl mit 2 als auch mit 3 synchronisieren kann. Dabei sei allerdings zu beachten, dass nur weil eine Send- und eine Receive-Operation auf dem selben Channel und in unterschiedlichen Routine geschehen, nicht in jedem Fall eine Kommunikation zwischen diesen möglich ist. Man betrachte dazu das Beispiel in Abb. ??.

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

Um mögliche Kommunikationspartner zu erkennen, nicht mögliche Kommunikationspartner wie in Abb ?? aber auszuschließen, werden Vector-Clocks verwendet. Die grundlegende Idee basiert auf [4].

In einem ersten Durchlauf wird dabei der Trace mit Vector-Clock Informationen nach der Methode von Fidge [9] erweitert. Dazu werden alle Elemente des Trace in der Reihenfolge durchlaufen, in denen sie in den Trace eingefügt wurden. Die korrekte Reihenfolge zwischen den Traces ist dabei durch die Speicherung eines globalen Timers in den Elementen des Trace gegeben. Für jede Routine wird eine Vector-Clock gespeichert, welche für jede Routine einen Wert enthält. Zu Beginn werden all diese Werte auf 0 gesetzt. Bei jedem Post-Event, sowohl für Send als auch Receive und für Signal und Wait Elemente wird der Wert der eigenen Routine in der lokalen Vector-Clock um eins erhöht. Bei einem Post-Receive und einem Wait Element wird die Vectorclock vc' betrachtet, welche in der sendenden Routine zum Zeitpunkt des Post-Send- bzw. Signal-Elements vorlag. Da ein Send- bzw. Signal-Event immer vor dem Receive- bzw. Wait-Element erzeugt wird, wurde die entsprechende Vectorclock in jedem Fall bereits bestimmt. Die Zuordnung der Trace-Elemente ist möglich, da der globale Counter bei einem Send an den Empfangenden Channel mitgesendet wird,

und in dem entsprechenden Post-Receive- bzw. Wait-Trace-Element gespeichert wird. Bei einem Select-Statement wird nur derjenige Fall betrachtet, der auch tatsächlich ausgeführt wurde. Diese Vector-Clock vc wird nun mit der lokalen Vector-Clock vc der empfangenden Routine, bzw. der Wait-Routine q verrechnet und ersetzt diese. Dabei gilt

(TODO: Code aktivieren wenn synchronisiert)

wobei n die Anzahl der Routinen ist.

Für alle andern Elemente, z.B. Pre usw. wird einfach die lokale Vector-Clock der Routine übernommen, ohne diese zu verändern. Da nun die Vector-Clocks zu jedem Zeitpunkt bestimmt wurde, kann jedem Send- und Receive-Trace-Element eine Pre- und eine Post-Vector-Clock zugeordnet werden. Dabei handelt es sich um die Vector-Clocks, die bei Erzeugung des Pre- bzw. Post-Events in der Routine, in der die Operation ausgeführt wurde vorlag. Für hängende Operationen, bei denen kein Post-Element existiert, werden alle Werte der Post-Vector-Clock auf $\max(\text{Int32})$ gesetzt. Man betrachte das Beispiel in Abb. 7. **(TODO: Bild**

Abb. 7: Beispielprogramm für die Betrachtung der Vector-Clocks

aktivieren wenn synchronisiert) Man betrachte den Fall, in dem 3 mit 4 und dann 1 mit 5 synchronisiert und 2 eine hängende Operation bildet. In diesem Fall erhält man folgenden Trace:

$$[[\text{signal}(1, 2), \text{signale}(2, 3), \text{pre}(1, x?), \text{post}(7, 1, x?, 6), \text{pre}(8, x?), \text{post}(13, 1, x?, 11)]] \quad (1)$$

$$[\text{wait}(9, 2), \text{pre}(10, x!), \text{post}(11, 1, x!), \text{pre}(12, 1?)] \quad (2)$$

$$[\text{wait}(4, 3), \text{pre}(5, 1!), \text{post}(6, 2, 1!),] \quad (3)$$

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

Vector-Clock vc' und der Channel-Operation a . Sie geben sich also als

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

Man bezeichne zwei Vector-Clocks vc und vc' als vergleichbar, wenn $\forall i : vc[i] \leq vc'[i]$ oder $\forall i : vc[i] \geq vc'[i]$. Man schreibe in diesem Fall $vc \leq vc'$ bzw. $vc \geq vc'$ bzw. allgemein $vc \preceq vc'$. Andernfalls bezeichnen man vc und vc' als unvergleichbar $vc \not\preceq vc'$. Sind zwei Vector-Clocks unvergleichbar, sind sie unabhängig und können somit gleichzeitig auftreten. Man erkennt also zwei Operationen, welche eine mögliche Kommunikation durchführen können daran, dass sie auf dem selben Channel definiert sind, eine Send- und eine Receive-Operation definieren und dass entweder ihre Pre- oder Vector-Clock (oder beide) unvergleichbar sind. Um alternative Kommunikationspartner für hängende Kanäle zu finden, werden also die Pre- und Post-Vector-Clocks der Channels mit dem selben Channel verglichen.

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

4.3 Gepufferte Kanäle

Anders als in ungepufferten Kanälen müssen in gepufferten Kanälen Send und Receive einer Nachricht nicht gleichzeitig ablaufen. Hierbei kann es zu Situationen kommen, in dem eine Nachricht zwar erfolgreich gesendet, aber nie ausgelesen wird. Man betrachte dazu das Beispiel in Abb. 8. Es besitzt 2 Send- aber nur eine Receive-Operation. Wäre der Channel

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

Abb. 8: Beispielprogramm für nicht gelesenen Nachricht in gepuffertem Channel

nicht gepuffert, und hätte 1 mit 3 synchronisiert, würde 2 blockieren und es würde zu einem Deadlock, bzw. dadurch dass 2 nicht in der Main-Routine liegt zu einer hängenden Operation kommen. Dadurch dass der Channel allerdings gepuffert ist können sowohl 1 als auch 2 senden ohne zu blockieren. Da der Trace somit ein gültiges Post-Event dieser Operation besitzt, wird dieses Problem nicht durch die in Abschnitt 4.2.1 beschriebenen Methode erkannt. Solche Situation lassen sich allerdings aus Trace einfach erkennen. Dazu wird der mit Vector-Clocks annotierte Trace durchlaufen. Auf diesem wird nun für jeden Channel gezählt, wie oft erfolgreich gesendet bzw. erfolgreich empfangen wurde (Anzahl der entsprechenden Elemente, bei denen die Post-Vector-Clock nicht $\max(\text{Int})$ ist). Ist die Anzahl der erfolgreichen Send größer als die Anzahl der erfolgreichen Receive, bedeutet diese, dass nach Abschluss des Programms auf dem entsprechenden Channel noch nicht gelesene Nachrichten vorhanden sind.

In dem Beispiel in Abb. 4.2.1 wird eine weitere Problematik mit gepufferten Channels deutlich. Betrachtet man nur die Vector-Clock Informationen, dann scheint es, als wären 2 und 3 potenzielle Kommunikationspartner. Dies ist allerdings nicht der Fall. In Go sind

gepufferte Channels als FIFO Queue implementiert [10]. Das bedeutet, dass bei einem Receive immer diejenige Nachricht ausgegeben wird, welche bereits am längsten in dem Puffer des Channels gehalten wird. Da 1 in dem Beispiel immer vor 2 ausgeführt wird und demnach in 3 immer die Nachricht aus 1 empfangen wird, bilden 2 und 3 kein mögliches Kommunikationspaar und sollte somit auch nicht als solches ausgegeben werden. Um solche Situationen zu erkennen, wird in dem Trace für jedes Post-Event gespeichert, die wievielte Send- bzw. Receive-Operation auf diesem Channel bereits erfolgreich ausgeführt worden sind. Bei der Suche nach möglichen Kommunikationspartnern wird nun für gepufferte Channels überprüft, ob die Anzahl der gesendeten Nachrichten in dem Send-Statement mit der Anzahl der empfangenen Nachrichten in dem Receive-Statement übereinstimmt. Da die Send- und Receive-Operation nicht gleichzeitig ausgeführt werden müssen, ist es nicht notwendig die Vector-Clocks zu betrachten. **(TODO: continue)**

5 Vergleich mit anderen Detektoren

5.1 GFuzz

6 Zusammenfassung

(TODO: Zusammenfassung schreiben)

7 Acknowledgments

(TODO: Acknowledgments schreiben)

Abbildungsverzeichnis

1	Beispielprogramm für Tracer	8
2	Laufzeit des Instrumenters	10
3	Prozentualer Overhead des Tracers ohne Analyse	10
4	Beispielprogramm zyklisches Locking	13
5	Beispielprogramm mit hängendem Channel	15
6	Hängender Channel ohne Deadlock	15
7	Beispielprogramm für die Betrachtung der Vector-Clocks	17
8	Beispielprogramm für nicht gelesenen Nachricht in gepuffertem Channel . .	19

Tabellenverzeichnis

1	Laufzeit des Instrumenters für ausgewählte Programme	10
---	--	----

ToDo Counters

(TODO: Remove ToDo Counter List)

To Dos: 16; 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16

Parts to extend: 5; 1, 2, 3, 4, 5

Draft parts: 3; 1, 2, 3

Literaturverzeichnis

- [1] S. Taheri and G. Gopalakrishnan, “Goat: Automated concurrency analysis and debugging tool for go,” in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 138–150, 2021.
- [2] I. Danyliuk, “Visualizing concurrency in go,” 2016. https://divan.dev/posts/go_concurrency_visualize/.
- [3] The Go Team, “Go documentation: trace,” 2022. <https://pkg.go.dev/cmd/trace>.
- [4] M. Sulzmann and K. Stadtmüller, “Two-phase dynamic analysis of message-passing go programs based on vector clocks,” *CoRR*, vol. abs/1807.03585, 2018.
- [5] E. Kassubek, “GoChan.” <https://github.com/ErikKassubek/GoChan>, 2022. v 0.1.
- [6] P. Joshi, C.-S. Park, K. Sen, and M. Naik, “A randomized dynamic program analysis technique for detecting real deadlocks,” *SIGPLAN Not.*, vol. 44, p. 110–120, jun 2009.
- [7] J. Zhou, S. Silvestro, H. Liu, Y. Cai, and T. Liu, “Undead: Detecting and preventing deadlocks in production software,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, (Los Alamitos, CA, USA), pp. 729–740, IEEE Computer Society, 11 2017.
- [8] E. Kassubek, “Dynamic deadlock detection in go.” <https://github.com/ErikKassubek/BachelorProjektGoDeadlockDetection/blob/main/Bericht/Bericht.pdf>, 22 08.

- [9] C. J. Fidge, “Timestamps in message-passing systems that preserve the partial ordering,” 1988.
- [10] The Go Authors, “Go-standart library: src/runtime/chan.go,” 2014. <https://go.dev/src/runtime/chan.go>.

