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

(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 Tress 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. Anders als in [4] wird hingegen ein Program-Counter für alle Routinen und nicht ein separater Counter für

jede Routine verwendet. Dies ermöglicht es bessere Rückschlüsse über den genauen Ablauf des Programms zu ziehen. Die Syntax des Traces in EBNF gibt sich folgendermaßen:

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

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

- `signal(t, i)`: In der momentanen Routine wurde eine Fork-Operation ausgeführt, d.h. eine neue Routine mit Id i wurde erzeugt.

- `wait(t, i)`: Die momentane Routine mit Id i wurde soeben erzeugt. Dies ist in allen Routinen außer der Main-Routine das erste Event in ihrem lokalen Trace.

- `pre(t, as)`: Die Routine ist an einer Send- oder Receive-Operation eines Channels oder an einem Select-Statement angekommen, dieses wurde aber noch nicht ausgeführt. Das Argument as gibt dabei die Richtung und den Channel an. Ist $as = x!$, dann befindet sich der Trace vor einer Send-Operation, bei $as = x?$ vor einer Receive-Operation. Bei einem Select-Statement ist as eine Liste aller Channels für die es einen Case in dem Statement gibt. Besitzt das Statement einen Default-Case, wird dieser ebenfalls in diese List aufgenommen.

- `post(t, i, x!)`: 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. Diese 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, das in die Liste der Traces ein neuer, leerer Trace eingefügt wird, welcher für die Speicherung des Traces der Routine i verwendet wird. $\langle a|b\rangle$ bedeutet, dass ein Wert je nach Situation auf a oder b gesetzt wird. Welcher Wert dabei verwendet wird, ist aus der obigen Beschreibung der Trace-Elemente erkennbar. e_1 bis e_n bezeichnet die Selektoren in einem Select statement. e_i^* bezeichnet dabei einen Identifier für einen Selektor, der sowohl die Id des beteiligten Channels beinhaltet, als auch die Information, ob es sich um ein Send oder Receive handelt und e_i^m die Message, die in einem Case empfangen wurde.

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

Man betrachte als Beispiel das folgende Programm in Go:

```
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  }
```

Dieser ergibt den folgenden Trace:

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

Aus diesem lässt sich der Ablauf des Programms eindeutig nachverfolgen. **(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 Begin 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. 1) 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 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

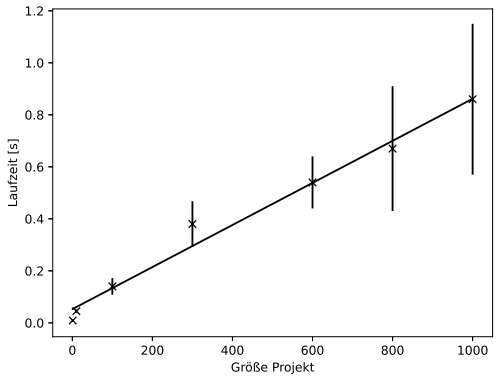

Abbildung 1: Laufzeit des Instrumenters in Abhängigkeit von der Größe des Programms

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

Tabelle 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. 1 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. **(EXTEND: Laufzeit)**

4 Analyze

Das folgende Kapitel soll sich nun mit der Analyze 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 Mutext

(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. Ein solches zyklisches Locking kann z.B. in der folgenden Situation auftreten.

```
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  }
```

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 augeführt werden, da das jeweilige Locks, welches beansprucht werden soll bereits durch die andere Routine gehalten wird. Da sich diese Situation auch nicht von alleine auflößen kann, blockiert dass Programm, befindet sich also in einem zyklischen Deadlock.

Da solche Situtationen nur in ganz besonderen Situation 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 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 Implemenierung des Locks findet sich in [8]. **(TODO: ist des erlabut, oder soll ich es nochmal komplett beschreiben)**

Der Graph wird basierend auf dem zufgezeichneten 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 existert. 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 erzeugen Baum. Wird ein solcher Zyklus erkannt, wird ebenfalls eine Warnung ausgegeben.

5 Zusammenfassung

(TODO: Zusammenfassung schreiben)

6 Acknowledgments

(TODO: Acknowledgments schreiben)

Abbildungsverzeichnis

Tabellenverzeichnis

ToDo Counters

(TODO: Remove ToDo Counter List)

To Dos: 8; 1, 2, 3, 4, 5, 6, 7, 8

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.