Chapter 4

Deadlocks in executions

When running a program, we want to be able to determine, if a *deadlock* occurred in a program execution E.

To define *deadlock*s, we classify all *active* routines (meaning routines that have not finished yet) by the following terms. We call routines that are running or runnable, or routines that are *waiting* on an operations, that will be conclude without being awoken from the outside (such as timers), *running* routines. Routines that are currently *blocked* on a concurrency operation like a *channel* or *mutex* are called *waiting* routines. If the operation, that blocks a *waiting* routine has no possibility of being woken up again, we call it a *dead* routine. We define

$$\mathcal{G}_R = \{G_i \mid \mathit{running}(G_i)\} \tag{4.1}$$
$$\mathcal{G}_W = \{G_i \mid \mathit{waiting}(G_i)\} \tag{4.2}$$
$$\mathcal{G}_D = \{G_i \mid \mathit{dead}(G_i)\} \tag{4.3}$$
$$\tag{4.4}$$

It is easy to see, that

$$\mathcal{G}_D \subseteq \mathcal{G}_W \tag{4.5}$$
$$\mathcal{G}_A = \mathcal{G}_R \cup \mathcal{G}_W, \tag{4.6}$$

where $\mathcal{G}_A$ is the set of all *active* routines.

4.1 Total deadlock

A *total deadlock* is a situation, where

$$\mathcal{G}_R = \emptyset, \tag{4.7}$$

Since a *waiting* routine can only be woken up by a *running* routine, it follows that

$$\mathcal{G}_R = \emptyset \Rightarrow \mathcal{G}_A = \mathcal{G}_W = \mathcal{G}_D, \tag{4.8}$$

or in other words, if there are no *running* routines, then all *active* routines are *dead*. This will necessary cause the program to get stuck and to never terminate. An example for this can be seen in code 4.1. The program consists of the *main* routine (G_0) and one other routine ($G1$). The trace depicts an execution, where the first *lock* operations in each routine can *acquire* the *lock* (indicated by !), but then it tries but is not able to *acquire* the second *lock* (indicated by ?), since it is held by the other routine. Since all routines, and especially the *main* routine G_0 are involved in this *deadlock*, the program will never terminate and we have created a *total deadlock*.

```
1  func main() { // G0
2    m := sync.Mutex{}
3
4    go func() { // G1
5      m.Lock()
6      n.Lock()
7      n.Unlock()
8      m.Unlock()
9    }()
10
11   n.Lock()
12   m.Lock()
13   m.Unlock()
14   n.Unlock()
15 }
```

	G_0	$G1$
1	$fork(G1)$	
2	$lock(m)!$	
3		$lock(n)!$
4	$lock(n)?$	
5		$lock(m)?$

Code 4.1: Example for program with *total deadlock*

Detecting *total deadlock* in Go is easy and already automatically done by the *scheduler* in the Go *runtime*. Since the *scheduler* can only run functions in $\mathcal{G}_R$, it knows a *total deadlock* has occurred, if it is not able to any routine. In this case,

the program is terminated automatically.

4.2 Partial deadlock

*Partial deadlock*s only affect a subset of *active* routines. In this case, the program will terminate as soon as the *main* routine is finished. For this, an example can be found in code 4.2. The code is almost the same as in code 4.1. The only difference is that the second *block*of lock and unlock is not longer in the *main* routine, but in its own routine. While the *lock* operations still create a *deadlock*, this block does not effect the *main* routine, which can simply return, making this kind of *deadlock* much more difficult to detect.

*Partial deadlock*s are situations, where

$$\mathcal{G}_D \neq \emptyset \wedge \exists G \in \mathcal{G}_A : G \notin \mathcal{G}_D, \tag{4.9}$$

meaning there are *dead* routines, but not all *active* routines are *dead*.

While *total deadlock*s are the more important type of *bug* to avoid, since only they can cause the program to get stuck, we also want to detect *partial deadlock*s. Since *partial deadlock*s are generally not a situation a developer intentionally adds to the code, it generally indicates an unwanted situation and may result in the program not performing as expected.

4.2.1 Idea

Unfortunately *partial deadlock*s are much more difficult to detect compared to *total deadlock*, since we do not only need to find the routines that are *waiting*, but als need to determine, if there is a possibility of them being released again. While for a *total deadlock* we only need to distinguish between *running* and *waiting* routines, we now have to additionally divide the *waiting* routines into those that are *dead* and those that are not.

The idea we implement is based on Golf [6] but differs especially in its implementation.

The general idea is to determine for each *waiting* routine, if the *blocking* element is also referenced in a non-*dead* routine.

We can illustrate this on the examples in code 4.3.

```
1  func main() {
2    m := sync.Mutex{}
3
4    go func() { // G1
5      m.Lock()
6      n.Lock() // blocks
7      n.Unlock()
8      m.Unlock()
9    }()
10
11   go func() { // G2
12     n.Lock()
13     m.Lock() // blocks
14     m.Unlock()
15     n.Unlock()
16   }()
17 }
```

	G_0	$G1$	$G2$
1	*fork*($G1$)		
1	*fork*($G2$)		
3		*lock*(m)!	
4			*lock*(n)!
5		*lock*(n)?	
6			*lock*(m)?
7	*return*		

Code 4.2: Example for program with *partial deadlock*

In code 4.3a, the program blocks on the *send* on *channel* c in $G1$. There is no other routine, that has a reference to this channel c. It is therefore impossible for another routine to wake up the *blocked* routines, and we have a deadlock. In code 4.3b on the other hand, we have a different situation. Lets assume, we call the *deadlock* detection, while the first routine is *waiting* on the *channel send*, but before the second routine has reached the corresponding *receive*. In this case, there is a reference to the *blocking* element in a *running* routine. The first routine is therefore not part of a deadlock.

Lets assume routine G_i is *waiting*. We call $\mathcal{E}_{Wi}$ the set of elements, G_i is *waiting* on (with elements we mean the underlying element, e.g. a specific mutex or channel, not the operation). In most cases, this set contains exactly one element. The number of elements can only be greater than one for *select*s without a default case and multiple non-default cases. Let Ref_i be the set of routines, that have a reference to any element $e \in \mathcal{E}_{Wi}$, not including G_i. For this, we use the same idea of reference as the Go GC [7].

Since we want to reason about the properties of the routines, we want to distinguish between properties that are *true* and properties where we known they are *true*. For a property P and routine G_i we write $P(G_i)$ if G_i is P, $!P(G_i)$ if we know, that G_i is P and $?P(G_i)$ if we do not know if G_i is P or not.

We can follow some assumptions directly from this definition. If the analysis is sound, Equation (4.10) states, that if we know that proposition is *true*, if must follows that the proposition is *true*. Equation (4.11) states, that we cannot, at the same time, know and not know whether a proposition is *true*. And eq. (4.12) indicates that if we don't know if G_i is P, we also don't know if G_i is not P.

$$\frac{!P(G_i)}{P(G_i)} \tag{4.10}$$

$$\frac{}{!P(G_i) \wedge ?P(G_i)} \tag{4.11}$$

$$\frac{?P(G_i)}{?(\neg P(G_i))} \tag{4.12}$$

Let a routine G_i have the property *suspect*, if we can determine, that our analysis cannot determine if the routine is *dead* or not.

At the start of our reasoning, we know for each routine, either, that it is *running* or that it is *waiting*. A routine cannot be *dead* and *suspect* at the same time, but a *waiting* routine is either *suspect* or *suspect*. A *running* can neither be *dead* nor *suspect*.

$$\frac{!waiting(G_i)}{!(dead(G_i) \veebar suspect(G_i))}, \tag{4.13}$$

$$\frac{!running(G_i)}{!(\neg suspect(G_i)) \qquad !(\neg dead(G_i))}, \tag{4.14}$$

We define the property *waiting'* of a routine as

$$waiting'(G_i) \equiv waiting(G_i) \wedge (?dead(G_i)) \wedge (?suspect(G_i)) \tag{4.15}$$

$$\mathcal{G}_W{}' = \{G_i \mid waiting'(G_i)\} \tag{4.16}$$

We can than determine if a *waiting* routine is part of a *partial deadlock*, by applying the following rules:

$$\frac{\text{NoReference:}\quad G_i \in \mathcal{G}_W{}' \qquad Ref_i = \emptyset}{!dead(G_i)} \tag{4.17}$$

$$\text{DeadReference:}\quad \frac{G_i \in \mathcal{G_W}' \qquad \forall G_j \in Ref_i : !dead(G_j)}{!dead(G_i)} \tag{4.18}$$

$$\text{NonDeadReference:}\quad \frac{G_i \in \mathcal{G_W}' \qquad \exists G_j \in Ref_i : (!running(G_j)) \vee (!suspect(G_j))}{!suspect(G_i)} \tag{4.19}$$

$$\text{NoOtherRule:}\quad \frac{G_i \in \mathcal{G_W}' \; \neg \exists G_j \in \mathcal{G_W} : \text{other rule can be applied to } G_j}{!dead(G_i)} \tag{4.20}$$

NoReference describes situations, where no other routines has a reference to any of the *blocking* elements. This is the situation shown in code 4.3a. Since no other routine has a reference to *channel c* and can therefore wake up the *blocking send* in line 5, the routine $G1$ is *dead* and therefore part of a (partial) deadlock.

DeadReference is the situation, where all other *active* routines, that hold a reference to a *blocking* element are *dead*. We can see this in code 4.3c. Lets assume, G_1 has acquiring *user3* the routine and then terminated. It is therefore not an *active* routine anymore and can be ignored for further analysis. G_2 now blocks on the *lock*. There is an *active* reference in G_3, but the routine would first need to run the *send* on the *channel*. By NoReference we can conclude, that G_3 is *dead*, since no other *active* routine contains a reference to the channel. Given this, we apply DeadReference to conclude, that G_2 is also *dead*, since all *active* routines holding a reference to the *mutex* (not including G_2 itself) are *dead*.

NonDeadReference stats, that if that if a *blocking* element in a *waiting* routine G_i has a reference in another routine G_j that is not known to be *dead*, we cannot know, wether it is *dead* and therefore say it is *suspect*. This rule can be split into two parts, one where the referenced routine is *running* and one, where we know it is *suspect*.

In the first case, we know that, if a *blocking* element in a *waiting* routine G_i has a reference in another routine G_j that is still *running*, G_i is not *dead*, since G_j could release the *block*in G_i. An example for this is in code 4.3b. Here the send on the *channel c* blocks routine G_1. If we run the analysis while G_2 is in the *doSomething* function, G_2 is in *running* and in Ref_i. This means, it can wake up the *blocked* routines, by calling the *receive* of *c*. While we cannot conclude from such a situation, that a *waiting* routine is in a *deadlock*, we can also not

guarantee, that it is not. This is especially the case, if the functions on the *blocked* elements are not compatible with the *blocked* operations. If the *receive* would be a *send*, the routine would be *dead*. Since we cannot distinguish between the two situations based on the given information, we cannot determine the routine to be *dead* or not *dead*, meaning it is *suspect*.

The second case can be seen as a transitive rule. To determine if a *waiting* routine could at some point be woken up, we need to determine if there is a routine that could wake it up, meaning a non *dead* routine that holds a reference to *blocking* element. This is the case, if a routine is either *running* or if it is currently *waiting*, but could be woken up in the future. This is marked by a routine being *suspect*. For this reason we can conclude, that a routine where the *blocked* element is referenced in a routine that is marked *suspect* must also be *suspect*. An example for this can be seen in code 4.3d. Here, we assume, that G_1 is *blocked* by the *send* on c, G_2 is *blocked* by the *send* on d and G_3 is in the *doSomething* function. We want to determine the status of G_1. The *blocking* element in G_1 is also referenced in G_2. Since G_3 is *running*, we can use AliveReference to determine, that G_2 is *suspect*. This means it could be *dead*, but could also be woken up again at some point. If it is woken up again, this would also release G_1, meaning we cannot determine that G_1 is *dead*. But if G_2 would be *dead*, or if the operations in G_2 would not be compatible with the *send* in G_2 (we only know that there is a reference, but not if it is the required *receive* or a *send* or), G_1 would be *dead*. We therefore classify the routine as *suspect*.

NoOtherRule is a default rule. If we get to a point, where there are still routines G_i with $waiting'(G_i)$, but we cannot apply any other rule to any of those routines, we can conclude that all routines that have not been classified as either *dead* or *suspect* must be *dead*. This mainly covers the case for cyclic *deadlock*. If we use code 4.2 as an example we can see why this is required. We cannot use NoReference, since for each *blocking* element, the other routine contains a reference. We cannot apply AliveReference since non of the routines containing references to the *blocked* elements are *running*. Since NoReference and AliveReference cannot be used, we cannot classify any of the routines as *dead* or *suspect*. It is therefore not possible to apply DeadReference or SuspectReference. Since there is therefore no other rule we can apply, we apply NoOtherRule to determine, that both routines are *dead*, meaning they form a deadlock. Since for all not-*dead* routines we must have an unbroken chain of *suspect* routines going back to a *running*routine. This means, we must

have been able to call NonDeadReference on each of those routines at some point during the analysis. By doing so, we removed them from $\mathcal{G}_W{}'$ which means NoOtherRule cannot be applied. This meaning, this rule cannot lead to a non-*dead* routine being labeled !*dead*.

NoReference and DeadReference are not necessarily required, since all routines that are determined to be *dead* by these routines, would also be labeled *dead* by NoOtherRule. These rules are nevertheless retained, to promote clarity and aid in the comprehension of both the underlying ideas and the system properties following from those rules.

4.2.2 Implementation

We implement the detection of *partial deadlock*s during runtime inside the patch Go *runtime*[1]. We run a background routine, that runs the analysis once per second and when the program terminates.

The analysis contains three phases. First we determine all currently *waiting* routines and for each of them, which element is *blocking*. Then we use the GC to determine all routines that have a reference to any of the *blocking* elements. At last, we use the rules stated above, to determine if there are any *dead* routines. If we find a *dead* routine, we report it as a *deadlock*.

Before we can determine if a routine are *dead* or not, we first need to find all routines in $\mathcal{G}_W$. Fortunately, the Go runtime allows us the easily detect such routines. Each routine is internally represented by an object called g. From this g, we can read the current state of the routine, e.g. *running*, *runnable* or waiting. A routine, that is *blocked* by a concurrency element is always in the *waiting* state. If the routine is in a *waiting* state, we can also query an additional field in g called *waitreason*, witch tells us, why the routine is *waiting*. If this vale is *waitReasonChanSend*, *waitReasonSyncMutexLock*, or similar, we known that the routine is currently *blocking* on one of the relevant concurrency operations and therefore *waiting*.

To detect if there are any *dead* routines in $\mathcal{G}_W$, we need to determine, if the *blocking* element could be released in the future. For this we need to determine, on which operations the routine blocks and if another routine could release it. While g does not directly contain a field pointing to the element it is *waiting* on, it is easy to add such an element for all relevant operations and always setting it,

[1] goPatch/src/runtime/advocate_partial_deadlock.go

```
1  var c = make(chan int)
2
3  func main() {
4    go func() { // G1
5      c <- 1
6    }()
7
8    go func() { // G2
9      doSomething()
10   }()
11 }
```

(a) NoReference

```
1  var c = make(chan int)
2
3  func main() {
4    go func() { // G1
5      c <- 1
6    }()
7
8    go func() { // G2
9      doSomething()
10     <-c
11   }()
12 }
```

(b) NonDeadReference (alive Reference)

```
1  var c = make(chan int)
2  var m = sync.Mutex{}
3
4  func main() {
5    go func() {  // G1
6      m.Lock()
7    }
8
9    go func() {  // G2
10     m.Lock()
11   }
12
13   go func() {  // G3
14     c <- 1
15     m.Unlock()
16   }()
17 }
```

(c) DeadReference

```
1  var c = make(chan int)
2  var d = make(chan int)
3
4  func main() {
5    go func() { // G1
6      c <- 1
7    }()
8
9    go func() { // G2
10     d <- 1
11     <-c
12   }
13
14   go func() { // G3
15     doSomething()
16     <-d
17   }
18 }
```

(d) NonDeadReference (suspect reference)

Code 4.3: Examples for rules

before the relevant *gopark* command is run, which causes the routine to switch to the *waiting* state.

Using the internal $forEachG$ function provided by the Go *runtime*, we can iterate over the g structures of all *active* routines. Using the internal state of the routine represented by g and the *waitreason*, we can filter out the set $\mathcal{G}_W$

of routines that are currently *waiting* and for each of these routines the set of *waiting* elements $\mathcal{E}_{Wi}$.

In the second phase we adapt and run the Go GC [7]. The GC finds all "roots", in our case, meaning variables accessible from the stack and global variables, and marks every object that can be reached from these roots. By propagating this process through references, like pointers, slices, maps or channels, it marks the accessible memory. We can use this functionality to our advantage. By slightly modifying the implementation, we can determine for each marking, from which routine the propagation was started and therefore, which routines has a reference to an object at a given memory address. When marking a cell, we check if the memory address is equal to the address of one of the elements at least one routine is *waiting* and, if so, store which routine has marked a reference to the element[2]. Doing this, we can for each *blocking* element find all active routines, that have a reference to the same element.

Allocating memory while the GC is running can lead to problems. For this reason, we need to avoid this by allocating all required memory before the GC is started. This is mainly required when storing which routines reference each *blocked* element.

In the third phase, we iterate over all routines in $\mathcal{G}_W$ and try to apply any of the rules outlined in eq. (4.17)-(4.19). We repeat this, until none of the rules can be applied to any of the *waiting* routines and then report all *waiting* routines, that are marked as *dead* as well as all *waiting* routines that have not been marked *suspect*, in accordance with NoOtherRule (eq. (4.20)).

4.2.3 Soundness and Completeness

The soundness of the analysis follows directly from the rules as described in section 4.2.1. This means that eq. (4.10) is *true* for $P = dead$. While for our implementation soundess is not as critical as in Golf, since we only report the detected result and do not attempt to reclaim stuck routines (cf. section 4.2.4), we still wand to guarantee, that no false positives can be detected, since it would needlessly complicate the analysis, if we would be required to manually confirm every reported situation.

The approach is not complete, meaning there are situations, in which a *dead* routine cannot be determined to be *dead*. For each *blocking* element, we only

[2] goPatch/src/runtime/mgcmark.go, line 1431–1437

determine which other routines contain a reference to this element. We do not determine if there is an operation on the element and if there is, if it is compatible or reachable.

The first problem lies in global elements, as shown in code 4.4a. A global variable is accessible from each routine. While the implemented modification to the GC is able to distinguish between global and local variables, it is for global variables not able to decide, in which routines it is used. This means, if a routine blocks on a global variable, the analysis will act as if all *active* routines have a reference to the *blocking* routine. This means, it is only possible to detect a *deadlock* on a global variable, if it is a *total deadlock*.

An example for a false negative with a non-compatible partner can be seen in code 4.4b. Here the *send* on *channel* c in G_1 has no possible partner and can therefore never be released. It is therefore part of a *deadlock*. Lets assume, we call the analysis while G_1 is *waiting* on the *channel* and G_2 is running the *doSomething* function. This means, we get $\mathit{waiting}'(G_1)$, $!\mathit{running}'(G_2)$ and $G_2 \in \mathit{Ref}_1$. By eq. (4.19) (NonDeadReference) we classify G_1 as $!\mathit{suspect}(G_1)$ and therefore not as *dead*. Deadlocks of this forms can often be detected by running the analysis again at a later point in time. As soon as G_2 reaches the *send* operation, we cannot apply (NonDeadReference) anymore, and the two routines are detected as *dead* by eq. (4.20) (NoOtherRule).

A third scenario that can cause a false positive, is if the operation that could wake up a operation is not executed or reachable. Examples for this can be seen in code 4.4c and d. In both examples, there is a *send* on *channel* c in G_1. Additionally, both examples G_2 has a *receive* on c, which could un-*block* G_1. The analysis will therefor not report a deadlock in either of the cases, event through both examples contain a deadlock. In code 4.4c the receive cannot un-*block* G_1, since G_2 returns before reaching the *receive*. In situations like this, we can often detect the deadlock later, by running the analysis again after the routine has returned. This is not the case for d. Here, the *receive* is preceded by an infinite *for* loop. This means that the *receive* can never be reached, meaning the *blocking send* can never be released. The routine is also never terminated (except if the *main* routine terminates). For this reason it is impossible for us to detect such deadlocks even if we run the analysis again at a later time.

```
1  var c = make(chan int)
2
3  func main() {
4    go func() {
5      c <- 1
6    }()
7  }
```

(a) Not compatible

```
1  c := make(chan int)
2
3  go func() {  // G1
4    c <- 1
5  }()
6
7  go func() {  // G2
8    doSomething()
9    c <- 1
10 }
```

(b) Not compatible

```
1  c := make(chan int)
2
3  go func() { // G1
4    c <- 1
5  }()
6
7  go func() {  // G2
8    if true {
9      return
10   }
11
12   <- c
13 }
```

(c) Not reachable 1

```
1  c := make(chan int)
2
3  go func() {  // G1
4    c <- 1
5  }()
6
7  go func() {  // G2
8    for {}
9
10   <- c
11 }
```

(d) Not reachable 2

Code 4.4: Examples for false negatives

4.2.4 Comparison with Golf

how is our implementation different from [6]

Compare Overhead with Golf

Write why we do not implement reclame