h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

# Hochschule Darmstadt

– Fachbereich Informatik–

# Möglichkeiten zur Erkennung von Thread Safety Problemen und deren Lösung

Wissenschaftliche Arbeit zu einem aktuellen Thema der Informatik


vorgelegt von

**Tobias Renner**

Matrikelnummer: 1113581


Referent : Jens Karas

# ERKLÄRUNG

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

*Darmstadt, 20. März 2023*

Tobias Renner

## ABSTRACT

Short summary of the contents in English. Approximately one page...

BTW: A great guide by Kent Beck how to write good abstracts can be found here:

https://plg.uwaterloo.ca/~migod/research/beckOOPSLA.html

# ZUSAMMENFASSUNG

# INHALTSVERZEICHNIS

# ABBILDUNGSVERZEICHNIS

# TABELLENVERZEICHNIS

# LISTINGS

# ABKÜRZUNGSVERZEICHNIS

| | |
|---|---|
| TSVD | Thread Safety Violation Detection |
| CLI | Command Line Interface |
| API | Application Programming Interface |
| HB | Happens Before |

# Teil I

# WISSENSCHAFTLICHE ARBEIT

# 1 EINLEITUNG

## 1.1 PROBLEMSTELLUNG

Threads findet man in allen Programmen heutzutage. Auch schon die Ausführung eines einfachen Java Programms ohne Multithreading erstellt nicht nur einen Thread für das Main Programm, sondern auch ein weiterer für die Garbage Collection. Auch Servelets, wie zum Beispiel Spring Boot, erstellen für jede Anfrage eines HTTP Clients einen eigenen Thread, so dass mehrere Nutzer gleichzeitig die gleiche Schnittstelle aufrufen können [vgl. 4, S. 8].

Multithreading bietet durch die Parallelisierung der Aufgaben Vorteile, beispielsweise das komplexe Anwendungen bessere Leistung haben [vgl. 4, S. 3].

Die durch Multithreading erlangte Leistung bringt aber auch einen großen Nachteil mit sich. Durch das parallele Aufrufen des selben Programms können mehrere Threads gleichzeitig auf die gleiche Ressource zugreifen. Wenn alle Operationen der Threads auf dieselbe Variable nur Leseoperationen sind besteht zunächst kein Problem, dies wäre zum Beispiel in Java bei einer `final` Variable der Fall, die read only ist. Probleme entstehen dann, wenn eine der Operationen schreiben ist. Bei schlechtem Timing kann es zum Beispiel zu dem Problem kommen, dass eine Ressource von zwei Threads gleichzeitig beschrieben wird, wodurch eine Inkonsistenz in den Daten entsteht [vgl. 4, S. 11–15].

## 1.2 ZIELSETZUNG

Das Ziel dieser Arbeit ist es, den Lesern aufzuzeigen was Thread Safety ist und warum man es braucht. Zudem möchte die Arbeit Möglichkeiten aufzeigen, wie man Programme auf Thread Safety Probleme durch verschiedene Methoden überprüfen kann und dadurch vorher unbekannte Bugs erkennen kann. Zuletzt wird die Arbeit dann darauf eingehen, wie man die zuvor gefundenen Probleme durch Synchronisation beheben kann.

# 2 STAND DER FORSCHUNG

Zu dem ausgewählten Thema "**Thread Safety**" liegt bereits einige Literatur von beispielsweise Li u. a. [5] und Erickson u. a. [2] vor. Das Thema "**Thread Safety Probleme überprüfen**" wurde dabei in der Literatur in Dynamische und Statische Race Detection Programme aufgeteilt. Mit Dynamischer Erkennung beschäftigten sich Li u. a. [5], Erickson u. a. [2] und Savage u. a. [7], wohingegen sich Voung, Jhala und Lerner [8] mit Statischer Erkennung beschäftigt. Li u. a. [5] und Erickson u. a. [2] gehen hierbei auf Dynamische Race Detection durch die Happens Before (HB) Beziehung ein, wohingegen Savage u. a. [7] auf den Lockset Algorithmus eingeht.

Mit dem Thema "**wie man diese Probleme nach dem Erkennen dann löst**" beschäftigt sich Göetz u. a. [4] und Fekete [3]. Beide gehen dazu beispielhaft auf die Programmiersprache Java ein. Fekete [3] geht jedoch aus der Sicht lehrender Professoren der University of Sydney an das Thema heran und behandelt hierbei wie sie Studierenden das Schreiben von Thread sicheren Klassen beibringen kann.

# 3 THEORETISCHE GRUNDLAGEN

Dieses Kapitel beschäftigt damit, was ein Thread ist und wie daraus Multithreading entseht. Die darauf folgende Sektion beschäftigt sich damit, was Thread Safety ist und welche Probleme durch Multithreading entstehen können.

## 3.1 THREADS UND MULTITHREADING

Ein Thread in einem Programm ist eine Reihe unabhängiger Befehle. Die meisten Programme starten dabei mit einem Thread. In Multiprozessor Systemen können diese Threads parallel ausgeführt werden. Programme erstellen im Laufe ihrer Ausführung weitere Threads oder beenden die von ihnen zuvor kreierte Threads. Dieses Konzept wird Multithreading genannt [vgl. 1, S. 70].

Threads, die zu einem gleichen Prozess gehören, greifen zudem auf denselben Adressraum zu und haben somit die gleichen Variablen und erzeugte Objekte, welche auf dem Heap gespeichert sind. Hier befindet sich das Problem der Thread Safety, da verschiedene Threads zur gleichen Zeit auf die gleichen Ressourcen zugreifen können [vgl. 4, S. 2].

## 3.2 THREAD SAFETY

"A class is thread-safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code" [4, S. 12].

```
1 public class UnsafeSequenz {
2         private int value;
3
4         public int getNext() {
5                 return value++;
6         }
7 }
```

Listing 3.1: Non-thread-safe Sequence Generator [4]

*Data Race*

Data Races entstehen, wenn zwei Threads gleichzeitig auf eine Ressource im Shared Memory zugreifen und mindestens eine der Operation beim Zugriff eine Schreiboperation ist [vgl. 1, S. 72].

Als Beispiel hierfür dient der Quellcode 3.1. Wenn zwei Threads, durch schlechtes Timing, parallel auf die Funktion `getNext()` zugreifen, kann in beiden Fällen die gleiche Zahl zurück gegeben werden. Dies ist möglich, da es sich bei Operation `value++` nicht eine, sondern um drei Unterschiedliche handelt. Zunächst holt man sich den momentanen Integer, der in der Variable `value` steht. Danach addiert man eins auf `value` und schreibt den neuen Wert für `value` zurück. Das Problem ist, das bei schlechtem Timing am Anfang dieselbe Zahl benutzt werden kann und somit am Ende dieselbe Zahl zurückgegeben wird [vgl. 4, S. 5].

# 4

# THREAD SAFETY PROBELEME ÜBERPRÜFEN

Race Detection Programme zur Überprüfung von Thread Safety Problemen werden in der Literatur in zwei Kategorien aufgeteilt: Dynamische und Statische Race Detection. Statische Race Detection analysiert dabei den Quell-, beziehungsweise Byte Code des Programms, ohne es auszuführen. Die Dynamische Methode hingegen führt das Programm aus und analysiert dieses während der Laufzeit. Unterscheiden wird bei der Dynamischen Race Detection zwischen der Erkennung durch eine HB Beziehung und den Lockset Algorithmus [vgl. 2, S. 4]. "Despite significant advances in static race detection, state-of the-art race detection tools are still predominantly dynamic"[6, S. 308].

## 4.1 DYNAMISCHE RACE DETECTION DURCH DIE HAPPENS BEFORE BEZIEHUNG

In dieser Sektion wird Dynamische Race Detection durch die HB Beziehung behandelt. Eine HB Beziehung liegt vor, wenn bei zwei Aktionen A und B eine Verzögerung in A eine Verzögerung in B verursacht. Wenn keine HB Beziehung vorliegt, kann man daraus ein Thread Safety Problem folgern [vgl. 5, S. 163]. Um die Wahrscheinlichkeit für das Vorkommen dieser Beziehung zu erhöhen, verzögern beide Algorithmen das Programm.

### *Data Collider*

Data Collider ist einer der Algorithmen, welcher die HB Beziehung ausnutzt, um Data Races zu finden. Dazu benutzt Data Collider zunächst einen Sampling Algorithmus um einen kleinen Teil an Speicher Zugriffen nimmt und diese speichert. Jene Speicherzugriffe werden daraufhin an die Konflikt Erkennung weiter gegeben, um Data Races zu finden. Zuletzt verwendet Data Collider mehrere Heuristiken, um gutartige Data Races zu entfernen [vgl. 2, S. 6].

### *Sampling Algorithmus*

Data Collider nimmt den zu Analysierenden Programm Code als Binärdatei und schreibt alle Stellen an denen der Speicher angesprochen wird in ein Sampling Set. Aus dem Sampling Set entfernt werden dabei Anweisungen die nur den Threadlokale Stack Speicherplätze ansprechen und Synchronisierungsanweisungen. Data Collider nimmt daraufhin proben aus dem Sampling Set in dem es Breakpoints einfügt, die dann genutzt werden um Konflikte zu erkennen [vgl. 2, S. 6].

*Konflikterkennung*

Konflikterkennung entsteht bei Data Collider entweder durch Data Breakpoints oder durch Reapeated Reads [vgl. 2, S. 7].

Data Breakpoints funktionieren über die vier von einem CPU der x86 Architektur gegebenen Data Breakpoint Register. Wenn ein Zugriff auf den Speicher eine Schreiboperation ist, sagt Data Collider dem Prozessor, dass dieser eine Falle aufstellen soll für diese Speicherstelle. Die Ausführung dieses Threads wird dann verzögert. Bei einer intialen Leseoperation wird die Falle nur ausgelöst, wenn auf die Speicherstelle mit einer Schreiboperation zugegriffen wird [vgl. 2, S. 7–8].

Sobald Data Collider keine Breakpoint Register mehr hat, greift der Algorithmus auf Repreated Reads zurück. Falls ein Zugriff auf den Speicher gemacht wird, wird der Thread wieder verzögert. Währenddessen wird die ganze Zeit überprüft, ob sich die Speicherstelle verändert. Wenn diese verändert wird, liegt ein Data Race vor. Jedoch ist es schwierig herauszufinden, durch welche Stelle das Data Race verursacht wurde [vgl. 2, S. 7–8].

*Umgang mit gutartigen Datenabweichungen*

"Research on data-race detection has amply noted the fact that not all data races are erroneous"[2, S. 8]. Diese Data Races müssen aus der Ausgabe der gefundenen Bugs entfernt werden. Dazu benutzt Data Collider drei Muster, um dies zu erkennen: Statische Zähler, Safe Flag Updates und Spezielle Variablen [vgl. 2, S. 8].

Statische Zähler sind da, um diverse Statistiken über das Programm zu machen. Statische Zähler welche Write Only sind werden als gutartig markiert [vgl. 2, S. 8].

Safe Flag Updates besteht darin, dass ein Thread ein Flag Bit in einem Speicherplatz liest, während ein anderer Thread ein anderes Bit in demselben Speicherplatz aktualisiert. Jedoch gehen bei Schreib-Schreib Konflikten hierbei Informationen verloren, somit werden diese weiterhin beachtet [vgl. 2, S. 8].

In Windows Kernel, für den Data Collider gemacht wurde, gibt es Spezielle Variablen, bei denen Data Races erwartet werden und kein Problem darstellen. Diese Stellen werden durch eine Datenbank an Speziellen Variablen von Data Collider nicht beachtet [vgl. 2, S. 8].

*TSVD*

Thread Safety Violation Detection (TSVD) benutzt Near-Miss Tracking, um potentielle Stellen Paare für Data Races zu finden. Zudem wird HB Tracking genutzt, welches durch Verzögerungen die Wahrscheinlichkeit für ein Data Race erhöht [vgl. 5, S. 163].

*Near-Miss Tracking*

In einem Programm nennt TSVD ein Paar an Programmstellen gefährlich, wenn zwei verschiedene Threads auf das selbe Objekt zugreifen und eine der Operationen, die ausgeführt wird, und die Zeit zwischen den Zugriffen unter einem bestimmten Schwellenwert ist [vgl. 5, S. 168].

*Wo wird Verzögert?*

TSVD benutzt ein Trap Set, um gefährliche Paare zu speichern. Der Algorithmus will Near-Misses zu wahren Konflikten machen, also werden Near-Misses in das Trap Set hinzugefügt. Aus dem Trap Set werden Paare entfernt, die entweder einen wahren Konflikt ausgelöst haben oder eine HB Bezeihung zueinander haben [vgl. 5, S. 167].

*Wann wird Verzögert?*

Die Planung und Injektion sind bei TSVD in einem Durchlauf des Programms. Zudem werden mehrere Testdurchläufe gemacht, da die Möglichkeit besteht, dass gefährliche Paare nie in der Nähe von einander aufgerufen werden. Das Trap Set, welches TSVD speichert, wird in einen Trap File geschrieben und beim nächsten Durchlauf als initial Wert für das Trap Set genutzt [vgl. 5, S. 169].

*Algorithmus*

Listing 4.1: TSVD Trap Mechanism

```
1 OnCall (thread_id, obj_id, op_id) {
2      check_for_trap(thread_id , obj_id , op_id)
3      if (should_delay(op_id)) {
4          set_trap(thread_id, obj_id, op_id)
5          delay()
6          clear_trap(thread_id, obj_id, op_id)
7     }
8 }
```

In 4.1 dargestellt ist der Algorithmus, der verwendet wird, um Thread Safety Verstöße in TSVD zu finden [Figure 5, 5, S. 166].

Das Prinzip des Algorithmus ist eine Falle für Thread Safety Verstöße zu stellen und zu warten, ob ein anderer Thread die Falle auslöst.

Die Funktion `OnCall` nimmt als Parameter die Id des Threads, von dem diese ausgelöst wird, die Id vom Objekt, auf welches zugegriffen wird, und die Operation Id, also welche Operation auf dem Objekt ausgeführt wird.

Sobald die Methode `OnCall` von einem Thread auf ein Objekt aufgerufen wird, wird in Zeile 3 überprüft, ob eine Falle gesetzt werden soll. Wenn diese gesetzt werden soll, so wird dies umgesetzt und eine bestimmte Zeit gewartet bis die Falle aufgelöst wird. Wenn nun ein anderer Thread `OnCall` auf das selbe Objekt aufruft und einer der Operationen eine Schreib Operation

war, gilt die Falle als ausgelöst und vom Programm wird dieser Bug zurück gegeben [vgl. 5, S. 166].

*Implementation*

Die Implementation dieses Algorithmus wurde für .NET Applikationen gemacht (`https://github.com/microsoft/TSVD`). Dabei wurde das Programm aufgeteilt in den TSVD Instrumenter und TSVD Runtime.

Der Instrumenter ist dabei ein Command Line Interface (CLI) Tool oder in Visual Studio ein Post-Build Step. Als Eingabe nimmt das Programm dabei die Binärdatei eines Programms und eine Liste an Thread unsicheren Application Programming Interface (API)s. Der Instumenter ersetzt dann die Aufrufe auf die Thread unsicheren APIs mit Proxy Aufrufen. Wie diese Proxy Aufrufe funktionieren, ist in 4.2 dargestellt. [vgl. 5, S. 170].

TSVD Runtime hingegen implementiert die `OnCall` Methode und protokolliert den Kontext, wenn ein Bug gefunden wird [vgl. 5, S. 170–171].

```
1  // (a) Original code
2  List <int> listObject = new List <int>();
3  listObject.Add(15);
4
5  // (b) Instrumented code
6  List <int> listObject = new List <int>();
7  int op_id = GetOpId();
8  Proxy_123(listObject , 15, op_id);
9
10 // (c) Proxy method
11 void Proxy_123 ( Object obj , int x ,int op_id ) {
12     var thread_id = GetCurrentThreadId ();
13     var obj_id = obj.GetHashCode();
14     OnCall(thread_id , obj_id , op_id);
15     obj.Add(x);
16 }
```

Listing 4.2: TSVD Proxy Aufrufe

*Vergleich TSVD mit Data Collider*

In dieser Sektion wird TSVD mit Data Collider verglichen.
TSVD hat in der ersten Runde 42 Bugs und nach zwei Runden 11 weitere Bugs gefunden, während Data Collider signifikant weniger Bugs gefunden hat, was in 4.1 erkannt werden kann.

## 4.2 DYNAMISCHE RACE DETECTION DURCH LOCKSETS

In dieser Sektion wird Dynamische Race Detection durch Locksets betrachtet und anhand von Eraser nähergebracht. Auf einem hohen Level überprüft

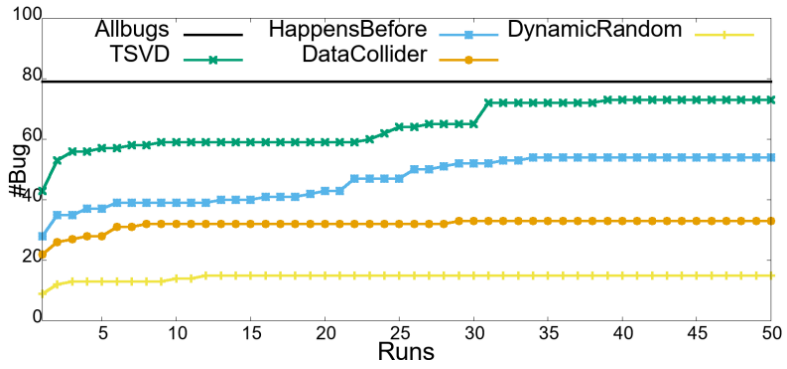


Abbildung 4.1: Vergleich TSVD mit Data Collider [5, S. 173]

Eraser, ob geteilter Speicherzugriff einem konsistentem Lock Mechanismus geschützt ist. Auf Locks wird in Kapitel 5 weiter eingegangen [vgl. 7, S. 392].

*Eraser*

*Lockset Refinement*

Erasers Lock Mechanismus ist, ob jeder Zugriff auf eine geteilte Variable durch ein Schloss gesichert ist. Dieser Mechanismus wird überprüft, indem Eraser alle Zugriffe auf den Speichert überwacht. Das Problem dabei ist, dass Eraser nicht weiß, welches Lock für welche Variable ist [vgl. 7, S. 396].

Um dies herauszufinden, benutzt Eraser Lockset Refinement. Für jede geteilte Variable `v` hat Eraser ein Set `C(v)` an möglichen Locks für `v`. Bei jedem Zugriff auf `v` von einem Thread wird die Schnittmenge zwischen die aktuellen Locks `locks_held` und `C(v)` gebildet und zurück in `C(v)` geschrieben. Wenn `C(v)` nun leer ist liegt ein Data Race vor. Dargestellt ist dieser Mechanismus in 4.1 [vgl. 7, S. 396–397].

| PROGRAM | LOCKS_HELD | C(V) |
|---|---|---|
| lock(mu1); | { } | { mu1, mu2 } |
| v := v + 1; | { mu1 } | { mu1, mu2 } |
| unlock(mu1); | { } | { mu1 } |
| lock(mu2); | { } | { mu1 } |
| v := v + 1; | { mu2 } | { mu1 } |
| unlock(mu1); | { } | { } |

Tabelle 4.1: Lockset Refinment [7, S. 397]

*Verbesserungen von Lockset Refinement*

Der Lock Mechanismus ist jedoch zu restriktiv. Um diesen zu Verbessern nimmt Eraser drei Szenarien aus dem Lock Mechanismus, die ein Data Race

verursachen würden, aber keines sind. Diese drei Szenarien sind: Die Initialisierung einer geteilten Variable, Read-Shared Data, also Varaiblen die nach einmaligen initialisieren nur noch gelesen werden, und Read-Write Locks, welches Variablen sind, auf die nur ein Thread mit Schreiboperationen zugreift [vgl. 7, S. 396–397].

In 4.2 dargestellt sind die Zustände, welche eine geteilte Variable bei der verbesserten Version des Lockset Refinements haben kann. Bei der Initialisierung wird der Zustand der Variable auf den `Virgin` Zustand gesetzt. Sobald ein Thread auf die Variable zugreift, ändert sich der Zustand zu `Exclusive` und bleibt solange in diesem Zustand bis ein neuer Thread auf die Variable zugreift. Wenn eine Lese Operation von dem neuen Thread ausgeht, ist der neue Zustand `Shared`. Wird von dem neuen Thread jedoch eine Schreib Operation ausgeführt, geht die Variable in den `Shared-Modified` Zustand. Um die oben genannten Fälle zu lösen, wird ein Data Race nur im `Shared-Modified` Zustand berichtet [vgl. 7, S. 397–399].

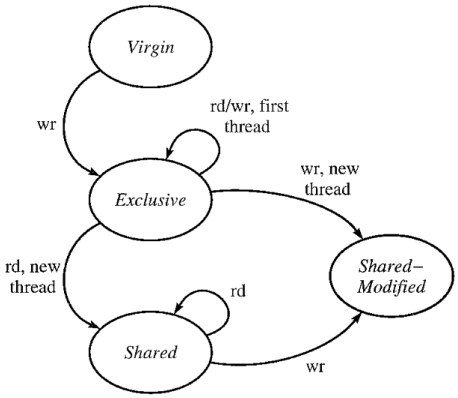


Abbildung 4.2: Eraser States [7, S. 389]

*Leistung*

Savage u. a. [vgl. 7, S. 400] beschreiben, dass Leistung nicht das primäre Ziel bei Eraser war und dementsprechend noch viele Stellen Verbesserungsmöglichkeiten.

Programme, auf die Eraser angewendet wird, sind um einen Faktor von 10 bis 30 langsamer als ohne. Zudem kann Thread Scheduling einen Einfluss auf das Ergebnis von Eraser haben kann [vgl. 7, S. 400].

## 4.3 STATISCHE RACE DETECTION

Diese Sektion wird sich mit Relay beschäftigen, welches ein Programm für Statische Race Detection ist, der auf dem Lockset Algorithmus basiert, welcher zuvor auch in der Dynmamischen Race Detection verwendet wurde.

*Relay*

Relay benutzt eine Bottom-Up Analyse mit drei Komponenten. Zunächst die symbolische Ausführung, dann die Lockset Analyse und eine Analyse, welche Guarded Accesses berechnet. Nach der Analyse werden aus dem Ergebnis Warnungen generiert, welche auf Data Races hinweisen [vgl. 8, S. 208].

*Symbolische Ausführung*

*Lockset Analys*

*Guarded Access*

*Generation von Warnungen*

# 5 LÖSEN VON THREAD SAFETY PROBLEMEN

Diese Sektion beschäftigt sich damit, wie die zuvor gefundenen Thread Safety Verstöße behoben werden können. Die Erklärung wird Beispielhaft in Java gemacht. Für andere Programmiersprachen gibt es aber ähnliche Methoden, welche man benutzen kann um diese Fehler zu beheben.

## 5.1 SYNCHRONIZED

Java bietet mehrere Optionen zum Lösen der Verstöße. Zum einen bietet Java seit Java 5 die `java.util.concurrent` Bibliothek, mit der sich der nächste Teil beschäftigt. Zum andern wird das `synchronized` keyword gestellt, mit dem sich diese Sektion beschäftigt [vgl. 3, S. 121].

Um das `synchronized` keyword zu erläutern wird das Beispiel, aus den Theoretische Grundlagen, 3.1. In 5.1 wird eine Klasse beschrieben, die dieselbe Funktionalität wie die Klasse in 3.1 hat. Der Unterschied zwischen den Klassen ist, dass 5.1 Thread sicher ist und somit von mehreren Threads gleichzeitig aufgerufen werden kann, ohne das Problem entstehen [vgl. 4, S. 5–6].

```
1 public class Sequenz {
2         @GuardedBy("this") private int value;
3
4         public synchronized int getNext() {
5                 return value++;
6         }
7 }
```

Listing 5.1: Thread-safe Sequence Generator [4]

Das `synchroized` keyword setzt dabei eine Art Schloss, auch Lock genannt, auf das Objekt, sodass nur ein Thread gleichzeitig darauf zugreifen kann. Wenn also die Methode `getNext()` von einem Objekt der Klasse `Sequenz` vom Thread A aufgerufen wird und der Thread B dieselbe Methode des gleichen Objekts aufrufen will, muss Thread B solange warten bis A fertig ist und das Schloss auflöst [vgl. 4, S. 17].

## 5.2 LIVENESS HAZARDS

Ein Problem, dass durch schlechte Synchronisation auftreten kann, sind Liveness Hazards. Diese entstehen, wenn beispielsweise Thread A darauf wartet, dass Thread B eine Ressource, aufhört zu locken, aber Thread B dies nie tut. Diese Art von Liveness Hazard nennt man livelock [vgl. 4, S. 5–6].

```
1  class BankAccountA {
2      private int balance;
3      private Bank myBank;
4
5      public synchronized void transfer(BankAccountA target, int amount)
           throws DifferentBankException {
6          if (myBank != target.myBank)
7              throw new DifferentBankException();
8
9          balance -= amount;
10         synchronized(target) {
11             target.balance += amount;
12         }
13     }
14 }
```

Listing 5.2: Deadlock [3]

Ein weiterer Liveness Hazard ist der Deadlock [vgl. 4, S. 6]. Dieser entsteht, wenn verschiedene Threads sich gegenseitig blockieren und dadurch kein Thread weitere Aufgabe ausführen kann. Ein Beispiel hierfür ist 5.2. Problem hierbei ist, dass wenn das `target` gleich dem auszuführenden BankAccountA ist, das `target` nicht gelocked werden kann in Zeile 10, da es bereits durch den Funktionsaufruf gelocked ist [vgl. 3, S. 122].

```
1  class BankAccountB {
2      private int balance;
3
4      private Bank myBank;
5
6      public void transfer(BankAccountB target, int amount) throws
           DifferentBankException {
7          if (myBank != target.myBank)
8              throw new DifferentBankException();
9
10         synchronized(myBank) {
11             balance -= amount;
12             target.balance += amount;
13         }
14     }
15 }
```

Listing 5.3: Deadlock Lösung [3]

In 5.3 ist dieselbe Funktion dargestellt, mit dem Unterschied, dass diese keinen Deadlock erzeugt. Der Unterschied in der Klasse `BackAccountB` ist, dass am Anfang der Funktion nicht das Objekt gelocked wird und dadurch das vorher beschriebene Problem nicht mehr auftreten kann [vgl. 3, S. 122].

# 6 ZUSAMMENFASSUNG

## 6.1 FAZIT

Ziel der Arbeit war es, den Lesenden zu zeigen, was Thread Safety ist und warum man es braucht. Des Weiteren sollte die Arbeit erläutern, wie man Verstöße erkennt und erkannte Verstöße löst.

Was Thread Safety ist und wie ein Data Race entsteht, wurde in den Theoretischen Grundlagen erklärt.

Anschließend wurde darauf eingegangen, wie man durch Dynamische bzw. Statische Race Detection Data Races in einem Programm erkennen kann. Dabei wurde die Dynamische Race Detection zwischen der HB Beziehung und dem Lockset Algorithmus unterschieden.

Zuletzt wurde beispielhaft an dem Synchronized Keyword in Java erklärt, wie man die gefunden Data Races beheben kann und welche Probleme, also Deadlocks oder Livelocks, entstehen können, wenn man schlechte Synchronization verwendet.

## 6.2 AUSBLICK

In der Zukunft könnte man weitere Vergleiche zwischen Statischer und Dynamischer Race Detection erstellen. Zudem kann man weitere Algorithmen zur Statischen, beziehungsweise Dynamischen Data Race Detection betrachten, die eine höhere Anzahl an Bugs finden und schneller bei dem Suchprozess agieren.

# LITERATUR


[1] Utpal Banerjee, Brian Bliss, Zhiqiang Ma und Paul Petersen. "A theory of data race detection". en. In: *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*. Portland Maine USA: ACM, Juli 2006, S. 69–78. ISBN: 978-1-59593-414-7. DOI: 10.1145/1147403.1147416. URL: https://dl.acm.org/doi/10.1145/1147403.1147416 (besucht am 06. 02. 2023).

[2] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt und Kirk Olynyk. "Effective Data-Race Detection for the Kernel". In: (Okt. 2010). URL: https://www.usenix.org/conference/osdi10/effective-data-race-detection-kernel.

[3] Alan D Fekete. "Teaching students to develop thread-safe java classes". en. In: (Sep. 2008). URL: https://dl.acm.org/doi/pdf/10.1145/1597849.1384304.

[4] Brian Göetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes und Doug Lea. *Java Concurrency In Practice*. Mai 2006. ISBN: 978-0-321-34960-6. URL: https://d1wqtxts1xzle7.cloudfront.net/53777814/Java_Concurrency_In_Practice-libre.pdf.

[5] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath und Rohan Padhye. "Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing". en. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. Huntsville Ontario Canada: ACM, Okt. 2019, S. 162–180. ISBN: 978-1-4503-6873-5. DOI: 10.1145/3341301.3359638. URL: https://dl.acm.org/doi/10.1145/3341301.3359638.

[6] Mayur Naik, Alex Aiken und John Whaley. "Effective Static Race Detection for Java". en. In: (Juni 2006). URL: https://dl.acm.org/doi/pdf/10.1145/1133981.1134018.

[7] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro und Thomas Anderson. "Eraser: A Dynamic Data Race Detector for Multithreaded Programs". en. In: *ACM Transactions on Computer Systems* 15 (Nov. 1997). URL: https://dl.acm.org/doi/pdf/10.1145/265924.265927.

[8] Jan Wen Voung, Ranjit Jhala und Sorin Lerner. "RELAY: Static Race Detection on Millions of Lines of Code". In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC-FSE '07. Dubrovnik, Croatia: Association for Computing Machinery, 2007, S. 205–214. ISBN: 9781595938114. DOI: 10.1145/1287624.1287654. URL: http://progsys.ucsd.edu/~rjhala/papers/relay.pdf.