

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

ABSTRACT

Multithreading wird in allen modernen Computer Programmen verwendet, da diese den Vorteil bringen, dass Programme schneller ausgeführt werden können. Jedoch kann es durch Paralleles zugreifen, von verschiedenen Threads, auf die gleiche Variable innerhalb eines Programms zu Thread Safety Problemen führen. Diese Arbeit beschäftigt sich mit dynamischen und statischen Methoden zur Erkennung von Thread Safety Problemen, sowie das Lösen solcher Probleme mittels verschiedener Lösungsansätzen.

INHALTSVERZEICHNIS

I Wissenschaftliche Arbeit

1	Einleitung	2
1.1	Problemstellung	2
1.2	Zielsetzung	2
1.3	Gliederung	2
2	Stand der Forschung	4
3	Theoretische Grundlagen	5
3.1	Threads und Multithreading	5
3.2	Thread Safety	5
4	Thread Safety Probleme überprüfen	7
4.1	Dynamische Race Detection durch die Happens Before Beziehung	7
4.1.1	Data Collider	7
4.1.2	TSVD	8
4.2	Dynamische Race Detection durch Lockset Analyse	11
4.3	Statische Race Detection	13
5	Lösen von Thread Safety Problemen	16
5.1	Synchronized	16
5.2	Liveness Hazards	16
6	Zusammenfassung	18
6.1	Fazit	18
6.2	Ausblick	18
	Literatur	19

ABBILDUNGSVERZEICHNIS

Abbildung 4.1	Vergleich TSVD mit Data Collider [7, S. 173]	11
Abbildung 4.2	Eraser States [9, S. 389]	12

TABELLENVERZEICHNIS

Tabelle 4.1	Lockset Refinement	11
-------------	------------------------------	----

LISTINGS

Listing 3.1	Non-thread-safe Sequence Generator [6]	5
Listing 4.1	TSVD Trap Mechanism	9
Listing 4.2	TSVD Proxy Aufrufe	10
Listing 4.3	Ownership Domäne [2]	14
Listing 5.1	Thread-safe Sequence Generator [6]	16
Listing 5.2	Deadlock [4]	17
Listing 5.3	Deadlock Lösung [4]	17

ABKÜRZUNGSVERZEICHNIS

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

Teil I

WISSENSCHAFTLICHE ARBEIT

EINLEITUNG

1.1 PROBLEMSTELLUNG

Threads findet man heutzutage in allen Programmen. Bereits bei der Ausführung eines einfachen Java Programms, erstellt Java nicht nur einen Thread für das Main Programm, sondern auch ein weiterer für die Garbage Collection. Auch Servlets, wie zum Beispiel Spring Boot, erstellen für jede Anfrage eines HTTP Clients einen eigenen Thread, sodass mehrere Nutzer gleichzeitig die gleiche Schnittstelle aufrufen können [vgl. 6, S. 8].

Multithreading bietet durch die Parallelisierung der Aufgaben Vorteile, beispielsweise das komplexe Anwendungen bessere Leistung haben [vgl. 6, 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 parallel 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 mindestens eine der Operationen schreiben ist. Kommt es dazu, dass zwei Threads zum gleichen Zeitpunkt die Ressource beschreiben führt dies zu Dateninkonsistenzen [vgl. 6, S. 11–15].

1.2 ZIELSETZUNG

Das Ziel dieser Arbeit ist es, den Lesern aufzuzeigen was Thread Safety ist und warum es so ein relevantes Thema in der Entwicklung von Software ist. Zudem soll so den Lesern eine Möglichkeit aufgezeigt werden, wie man Programme auf Thread Safety Probleme überprüfen kann. Dabei werden verschiedene Methoden beleuchtet, die dafür eingesetzt werden können, um potenzielle Bugs sichtbar zu machen. Zuletzt wird darauf eingegangen, wie man die gefundenen Probleme mittels Synchronisation behebt.

1.3 GLIEDERUNG

Zunächst wird die Arbeit sich mit dem aktuellen Stand der Forschung beschäftigen. Daraufhin werden die Theoretischen Grundlagen zum Thema Thread Safety erklärt. Dazu zählen auch die Grundlagen zum Thema Threads und Multithreading. Als Nächstes werden verschiedene Methoden zum Überprüfen von Thread Safety Problemen vorgestellt. Anschließend wird beispielhaft in Java gezeigt, wie durch Synchronisation Thread Safety Probleme lö-

sen kann. Zudem werden in diesem Kapitel mögliche Probleme aufgezeigt, die durch diese entstehen.

STAND DER FORSCHUNG

Zu dem ausgewählten Thema **“Thread Safety”** liegt bereits einige Literatur von beispielsweise Li u. a. [7] und Erickson u. a. [3] 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. [7], Erickson u. a. [3] und Savage u. a. [9], wohingegen sich Voung, Jhala und Lerner [10] mit Statischer Erkennung beschäftigt. Li u. a. [7] und Erickson u. a. [3] gehen hierbei auf Dynamische Race Detection durch die Happens Before (HB) Beziehung ein, wohingegen Savage u. a. [9] auf den Lockset Algorithmus eingeht.

Mit dem Thema **“wie man diese Probleme nach dem Erkennen dann löst”** beschäftigt sich Göetz u. a. [6] und Fekete [4]. Beide gehen dazu beispielhaft auf die Programmiersprache Java ein. Fekete [4] 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.

THEORETISCHE GRUNDLAGEN

Im folgenden Kapitel wird auf die Thematik von Threads eingegangen und was Multithreading ist. Im Anschluss daran wird das Thema Thread Safety behandelt und wie durch Multithreading Probleme entstehen können.

3.1 THREADS UND MULTITHREADING

Ein Thread ist ein Programm, welches aus einer Reihe von unabhängigen Befehlen besteht. Die meisten Programme starten 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 erstellten 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. Hier befindet sich das Problem der Thread Safety, da verschiedene Threads zur gleichen Zeit auf die gleichen Ressourcen zugreifen können [vgl. 6, S. 2].

3.2 THREAD SAFETY

Eine Klasse gilt als Thread sicher, wenn diese von mehreren Threads gleichzeitig aufgerufen werden kann, ohne zusätzliche Synchronisation oder Koordination vom Code, der die Klasse aufruft [vgl. 6, S. 12].

Data Race

```
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 [6]

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, zum gleichen Zeitpunkt 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. 6, S. 5].

THREAD SAFETY PROBLEME ÜBERPRÜFEN

Dieses Kapitel beschäftigt sich damit, wie man Programme auf Data Races überprüft.

Race Detection Programme zur Überprüfung von Thread Safety Problemen werden in der Literatur in zwei Kategorien unterteilt: 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. 3, S. 4]. "Despite significant advances in static race detection, state-of the-art race detection tools are still predominantly dynamic"[8, 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. 7, S. 163]. Um die Wahrscheinlichkeit für das Vorkommen dieser Beziehung zu erhöhen, verzögern beide Algorithmen das Programm.

4.1.1 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. 3, 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 Synchro-

nisierungsanweisungen. Data Collider nimmt daraufhin Proben aus dem Sampling Set in dem es Breakpoints einfügt, die dann genutzt werden um Konflikte zu erkennen [vgl. 3, S. 6].

Konflikterkennung

Konflikterkennung entsteht bei Data Collider entweder durch Data Breakpoints oder durch Repeated Reads [vgl. 3, 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 initialen Leseoperation wird die Falle nur ausgelöst, wenn auf die Speicherstelle mit einer Schreiboperation zugegriffen wird [vgl. 3, S. 7–8].

Sobald Data Collider keine Breakpoint Register mehr hat, greift der Algorithmus auf Repeated 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. 3, S. 7–8].

Umgang mit gutartigen Datenabweichungen

"Research on data-race detection has amply noted the fact that not all data races are erroneous"[3, 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. 3, 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. 3, 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. 3, 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. 3, S. 8].

4.1.2 TSVD

Thread Safety Violation Detection (TSVD) benutzt Near-Miss Tracking, um potentielle Stellenpaare für Data Races zu finden. Zudem wird HB Tracking

genutzt, welches durch Verzögerungen die Wahrscheinlichkeit für ein Data Race erhöht [vgl. 7, 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. 7, S. 168].

Wo wird Verzögert?

TSVD benutzt ein Trap Set, um gefährliche Paare zu speichern, bei denen potentiell Data Races entstehen können. 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** Beziehung zueinander haben [vgl. 7, 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. 7, S. 169].

Algorithmus

```
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 }
```

Listing 4.1: **TSVD** Trap Mechanism

In 4.1 dargestellt ist der Algorithmus, der verwendet wird, um Thread Safety Verstöße in **TSVD** zu finden [Figure 5, 7, 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 aufgelöst und vom Programm wird dieser Bug zurück gegeben [vgl. 7, 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 Instrumenter ersetzt dann die Aufrufe auf die Thread unsicheren APIs mit Proxy Aufrufen. Wie diese Proxy Aufrufe funktionieren, ist in 4.2 dargestellt. [vgl. 7, S. 170]. `TSVD Runtime` hingegen implementiert die `OnCall` Methode und protokolliert den Kontext, wenn ein Bug gefunden wird [vgl. 7, 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.

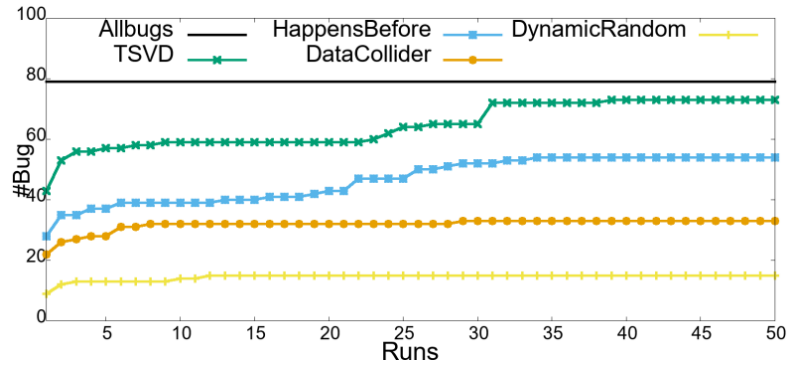


Abbildung 4.1: Vergleich TSVD mit Data Collider [7, S. 173]

4.2 DYNAMISCHE RACE DETECTION DURCH LOCKSET ANALYSE

In dieser Sektion wird Dynamische Race Detection durch Lockset Analyse betrachtet und anhand von Eraser nähergebracht. Auf einem hohen Level überprüft Eraser, ob geteilter Speicherzugriff einem konsistentem Lock Mechanismus geschützt ist. Auf Locks wird in Kapitel 5 weiter eingegangen [vgl. 9, S. 392].

Eraser

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 Speicherteil überwacht. Das Problem dabei ist, dass Eraser nicht weiß, welches Lock für welche Variable ist [vgl. 9, 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. 9, 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 Refinement [9, 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 Variablen die nach einmaligen initialisieren nur noch gelesen werden, und Read-Write Locks, welches Variablen sind, auf die nur ein Thread mit Schreiboperationen zugreift [vgl. 9, 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. 9, S. 397–399].

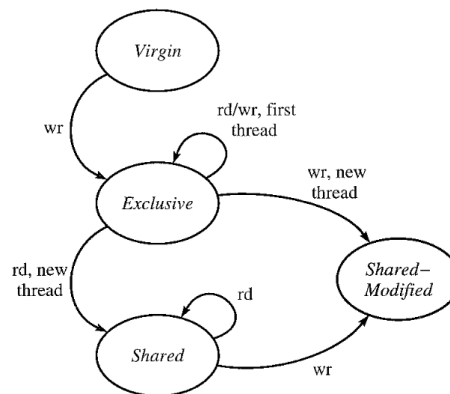


Abbildung 4.2: Eraser States [9, S. 389]

Leistung

Savage u. a. [vgl. 9, 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. 9, S. 400].

4.3 STATISCHE RACE DETECTION

Diese Sektion beschäftigt sich mit Statischer Race Detection. Diese funktioniert in den meisten Algorithmen, wie zum Beispiel Relay [vgl. 10, S. 208] oder RacerD [vgl. 5, S. 57], mittels der Analyse von Locks in einem gegebenen Programm. Diese Analyse ist ähnlich, wie die Dynamische Race Detection durch Lockset Analyse. Die Sektion wird sich auf den von Facebook genutzten RacerD Algorithmus [vgl. 2, S. 2] fokussieren und diesen erklären.

RacerD

RacerD analysiert alle nicht private Methoden von Klassen eines gegebenen Programms. Dabei müssen zwei Fragen beantwortet werden, für ein Paar Zugriffe einer Klasse [vgl. 2, S. 7]:

1. Sind die Zugriff auf die selbe Adresse?
2. Sind die Zugriffe gleichzeitig?

Um diese Fragen zu beantworten gibt es in RacerD die Locks, Threads, Ownership und Access Snapshots Domänen [vgl. 2, S. 7–8].

Locks Domäne

Die Locks Domäne erkennt, dass kein Data Race entsteht, dadurch, dass das Objekt durch das selbe Lock geschützt wird und deswegen nur ein Thread gleichzeitig drauf zugreifen kann [vgl. 2, S. 8]. RacerD hat als Ziel eine kleine Menge an Data Races zu finden, die mit hohem Vertrauen auch Races sind. Deshalb fokussiert sich RacerD darauf, unsicher Stellen zu finden, die durch kein Lock geschützt sind. Dabei versucht RacerD keine Races zu finden, die entstehen, da das Objekt durch das falsche Lock geschützt sind [vgl. 2, S. 9].

Threads Domäne

Die Threads Domäne erkennt, dass kein Data Race vorliegt, indem es erfasst, ob ein Stelle nur von einem Thread aufgerufen werden kann [vgl. 2, S. 8]. Um zu festzustellen, ob eine Stelle im Programm mehreren Thread aufgerufen wird "concurrent context inference" [2, S. 9] genutzt. Dafür gibt es drei abstrakte Zustände. Gestartet wird im Stadium "NoThread". Dieser bedeutet, dass diese Stelle nur auf einem Thread ausgeführt wird [vgl. 2, S. 9–10].

Werden Annotationen, wie @ThreadSafe oder @GuardedBy genutzt, wird in den Zustand "AnyThread" übergegangen. In diesem Zustand können Programm Code Stellen, die auf einem Thread laufen, der mit jedem anderen Thread verschachtelt werden kann. Dasselbe gilt auch bei Verwendung des synchronized keyword oder anderen Locks [vgl. 2, S. 10].

Wenn die Annotation `@UiThread` oder die Methode `assertMainThread()` benutzt wird ändert sich der Zustand in "AnyThreadButMain". Diese Prozeduren laufen auf dem Main Thread und können parallel zu Code in Hintergrund Threads laufen [vgl. 2, S. 10].

Ownership Domäne

Die Ownership Domäne bezieht sich auf Speicher, welcher nur von einem Thread angesprochen werden kann und dadurch zu keinem Race führen kann. 4.3 gilt als Beispiel hierfür. Die Methode `ownedAccess()` kann ohne Data Races von mehreren Threads ausgeführt werden. Dies liegt daran, dass die Threads zwar das gleiche ausführen, aber jeder Thread eine neue Instanz `x` der Klasse `C` sich erzeugt. Es macht sich die Tatsache zunutze, dass ein neu initialisiertes Objekt über den Befehl `new` zugewiesen wird [vgl. 2, S. 10].

```

1 void ownedAccess() {
2     C x = new C();
3     c.x = 42;
4 }
```

Listing 4.3: Ownership Domäne [2]

Access Snapshots Domänen

Die Access Snapshots Domäne nimmt relevante Informationen in einem Access Snapshot und speichert sie, sobald auf Dynamischen Speicher zugegriffen wird [vgl. 2, S. 11]. Gespeichert werden [vgl. 2, S. 8]:

- Die Stelle im Speicher, welche aufgerufen wird.
- Die Art des Zugriffs, also ob gelesen oder geschrieben wird.
- Wie viele Locks gehalten werden.
- Ob die Stelle von mehreren Threads ausgeführt wird (Thread Domäne).
- Ob die Stelle von nur einem Thread angesprochen werden kann (Ownership Domäne).

Data Races erkennen

Um Data Races mit RacerD zu erkennen werden für jedes Paar Access Snapshots mit einander verglichen. Um ein Data Race zu erzeugen müssen beide Access Snapshots auf die selbe Speicherstelle zugreifen und eine Operation darauf muss eine Schreiboperation sein. Zudem muss mindestens einer der Zugriffe ohne Lock passieren. Außerdem darf keiner der beiden Access Snapshots nur von einem Thread angesprochen werden. Zuletzt muss der Wert für die Thread Domäne "AnyThread" haben, also das mindestens einer

der Threads nicht der Main Thread ist [vgl. 2, S. 8]. Wenn ein Race gefunden wurde, wird ein Stacktrace zurückgegeben, sodass man den Bug analysieren kann [vgl. 2, S. 15].

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. 4, 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. 6, 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 [6]

Das `synchronized` 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. 6, 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. 6, 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 [4]

Ein weiterer Liveness Hazard ist der Deadlock [vgl. 6, 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. 4, 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 [4]

In 5.3 ist dieselbe Funktion dargestellt, mit dem Unterschied, dass diese keinen Deadlock erzeugt. Der Unterschied in der Klasse BankAccountB ist, dass am Anfang der Funktion nicht das Objekt gelocked wird und dadurch das vorher beschriebene Problem nicht mehr auftreten kann [vgl. 4, S. 122].

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.

Thread Safety ist gegeben, wenn diese von mehreren Threads gleichzeitig aufgerufen werden kann, ohne zusätzliche Synchronisation oder Koordination vom Code, der die Klasse aufruft. Wenn zwei Threads zur gleichen Zeit Parallel auf eine Ressource zugreifen, nennt man das Data Race und es gibt ein Thread Safety Problem.

Um Data Races zu erkennen gibt es zwei verschiedene Möglichkeiten. Die eine Möglichkeit sind dynamische Race Detection Algorithmen, welche in die HB-Beziehung und den Lockset Algorithmus aufgeteilt sind. Diese Methode Analysiert ein Programm während der Laufzeit. Die andere Möglichkeit ist es statische Race Detection Algorithmen zu verwenden, welche den Quellcode ohne diesen Auszuführen analysieren.

Die gefundenen Data Races kann man durch Synchronisierung beheben. In Java gibt es dafür das synchronized Keyword, welches ein Schloss auf das Objekt setzt, sodass nur ein Thread auf das Objekt zugreifen kann. Jedoch muss man dabei vorsichtig sein, da durch schlechte Synchronisierung Liveness Hazards, wie Deadlocks auftreten können.

6.2 AUSBLICK

Aufbauend auf diese Arbeit kann man weitere Vergleiche zwischen Statischer und Dynamischer Race Detection erstellen. Des Weiteren kann man weitere Race Detection erforschen, die eine höhere Anzahl an Bugs finden und schneller bei dem Suchprozess agieren. Zudem kann man Algorithmen betrachten mit denen es möglich ist, auch Liveness Hazards erkennen zu können, sodass man auch diese Art Bugs erkennen kann.

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](https://doi.org/10.1145/1147403.1147416). URL: <https://dl.acm.org/doi/10.1145/1147403.1147416> (besucht am 06.02.2023).
- [2] Sam Blackshear, Nikos Gorogiannis, Peter W. O’Hearn und Ilya Sergey. “RacerD: Compositional Static Race Detection”. In: Bd. 2. OOPSLA. New York, NY, USA: Association for Computing Machinery, 2018. DOI: [10.1145/3276514](https://doi.org/10.1145/3276514). URL: <https://dl.acm.org/doi/pdf/10.1145/3290370>.
- [3] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt und Kirk Olynyk. “Effective Data-Race Detection for the Kernel”. In: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. Vancouver, BC: USENIX Association, Okt. 2010. URL: <https://www.usenix.org/conference/osdi10/effective-data-race-detection-kernel>.
- [4] Alan D. Fekete. “Teaching Students to Develop Thread-Safe Java Classes”. In: *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’08. Madrid, Spain: Association for Computing Machinery, 2008, 119–123. ISBN: 9781605580784. DOI: [10.1145/1384271.1384304](https://doi.org/10.1145/1384271.1384304). URL: <https://dl.acm.org/doi/pdf/10.1145/1597849.1384304>.
- [5] Nikos Gorogiannis, Peter W. O’Hearn und Ilya Sergey. “A True Positives Theorem for a Static Race Detector”. In: Bd. 3. POPL. New York, NY, USA: Association for Computing Machinery, 2019. DOI: [10.1145/3290370](https://doi.org/10.1145/3290370). URL: <https://doi.org/10.1145/3290370>.
- [6] 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://dlwqtxtslxzle7.cloudfront.net/53777814/Java_Concurrency_In_Practice-libre.pdf.
- [7] 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](https://doi.org/10.1145/3341301.3359638). URL: <https://dl.acm.org/doi/10.1145/3341301.3359638>.

- [8] Mayur Naik, Alex Aiken und John Whaley. "Effective Static Race Detection for Java". In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '06. Ottawa, Ontario, Canada: Association for Computing Machinery, 2006, 308–319. ISBN: 1595933204. DOI: [10.1145/1133981.1134018](https://doi.org/10.1145/1133981.1134018). URL: <https://dl.acm.org/doi/pdf/10.1145/1133981.1134018>.
- [9] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro und Thomas Anderson. "Eraser: A Dynamic Data Race Detector for Multi-threaded Programs". In: Bd. 15. 4. New York, NY, USA: Association for Computing Machinery, 1997, 391–411. DOI: [10.1145/265924.265927](https://doi.org/10.1145/265924.265927). URL: <https://dl.acm.org/doi/pdf/10.1145/265924.265927>.
- [10] Jan Wen Voun, 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](https://doi.org/10.1145/1287624.1287654). URL: <http://progsys.ucsd.edu/~rjhala/papers/relay.pdf>.