

Fehlersuche und Visualisierung der Belegung von Synchronisationsmitteln in nebenläufigen Systemen

Marcel Sobottka

30. April 2020

Erstgutachter: Prof. Dr.-Ing. habil. Herwig Unger

Betreuer: Dipl.-Inform. (Univ.) Marcel Schaible

Inhaltsverzeichnis

Inhaltsverzeichnis	III
Abbildungsverzeichnis	IV
Quellcodeverzeichnis	VI
1 Motivation	7
2 Analyse	8
2.1 Deadlockerkennung allgemein	8
2.2 PEARL	10
2.3 OpenPEARL	13
2.4 MagicLock	14
3 Design	16
3.1 Übersicht	16
3.2 Erzeugung der Trace-Datei	16
3.3 Analysieren der Trace-Datei	18
3.4 Erweiterung: Potenzielle Deadlocks	19
4 Implementierung	21
4.1 Trace Funktion	21
4.2 Analyse Programm	23
4.3 Visualisierung von potenziellen Deadlocks	24
5 Validierung	31
5.1 Trace Funktion	31
5.2 Analyse Programm	35
5.3 Visualisierung von potenziellen Deadlocks	36
6 Ausblick	37
6.1 Offene Punkte	37
6.2 Weiterentwicklung	37
Literatur	38
A OpenPEARL	39
B C++	47
C Python	52

Abbildungsverzeichnis

Zustandsdiagramm einer SEMA Variablen	11
Zustandsdiagramm einer BOLT Variablen	12
UML Klassendiagramm für Trace-Funktionalität	17
Visualisierung der chronologischen Verwendung von <i>SEMA</i> Objekten	19
Visualisierung eines potentiellen Deadlocks	19
Ergebnisse der Laufzeitmessung der Trace-Funktionalität in OpenPEARL .	33
Ergebnisse der Speicherauslastung der Trace-Funktionalität in OpenPEARL	34
Ausgabe der Analyse-Anwendung	35
Vergößerte Darstellung von Abb. 5.2.1	36
Ergebnis der Erkennung von potenziellen Deadlocks aus Listing 5.2.1	36

Quellcodeverzeichnis

Beispiel einer OpenPEARL Anwendung mit einem potenziellen Deadlock . . .	14
Auszug aus LockTraceEntryFormatter.cc: Berechnung des Zeitpunkts	21
LockTracer.cc: Auszug aus der Implementierung des LockTracers	22
Files.common: Auszug aus der Auflistung der zu kompilierenden Dateien . .	23
Semaphore.cc: Auszug aus der Semaphore Implementierung in der Open- PEARL Laufzeitumgebung	23
traceFileReader.py: Auszug aus der Implementierung des Trace-Datei Parsers	23
generateTimeline.py: Auszug aus der Bestimmung der einzelnen Werte für den Graphen	24
generateTimeline.py: Auszug aus der Erzeugung des Graphen	24
magiclockLib/magiclockTypes.py: Repräsentation einer <i>Lock Dependency</i> aus Magiclock[2, S. 3]	25
magiclockLib/magiclockTypes.py: Repräsentation einer <i>Lock Dependency</i> <i>Relation</i> aus Magiclock[2, S. 3]	25
magiclockLib/lockReduction.py: Implementierung des <i>InitClassification(D)</i> Algorithmus aus Magiclock[2, S. 5]	26
magiclockLib/magiclockTypes.py: Datenstruktur der <i>init_Classification(D)</i> Methode	26
magiclockLib/lockReduction.py: Implementierung des <i>LockClassification(D)</i> Algorithmus aus Magiclock[2, S. 5]	27
magiclockLib/lockReduction.py: Implementierung des <i>LockReduction(D)</i> Algorithmus aus Magiclock[2, S. 5]	28
magiclockLib/cycleDetection.py: Implementierung des <i>DisjointComponents-</i> <i>Finder(Cyclic-set)</i> Algorithmus aus Magiclock[2, S. 8]	29
magiclockLib/cycleDetection.py: Implementierung des <i>CycleDetection(dc,</i> <i>D)</i> Algorithmus aus Magiclock[2, S. 8]	29
magiclockLib/cycleDetection.py: Implementierung des <i>DFS_Traverse(i, S,</i> <i>τ)</i> Algorithmus aus Magiclock[2, S. 8]	30
OpenPEARL Anwendung zum Testen der Trace-Funktionalität	31
Trace-Datei die bei aktivierter Trace-Funktionalität aus Listing 5.1.1 erzeugt wird	32
Pythonskipt zur Messung der Laufzeit	32
Pythonskipt zur Messung der Speicherauslastung	33
Beispielhafte Trace-Datei mit einem potenziellen Deadlock	35

Angepasste Semaphore.cc Implementierung der OpenPEARL Laufzeitumgebung	42
Angepasste Files.common der OpenPEARL Laufzeitumgebung	44
OpenPEARL Testanwendung für Performanztests	46
Header-Datei der Repräsentation eines Logeintrags	47
Implementierung der Repräsentation eines Logeintrags	47
Header-Datei des Formatieres für Logeinträge	48
Implementierung des Formatieres für Logeinträge	49
Header-Datei der Enumeration für den Typ eines Logeintrags	49
Header-Datei des Log-Tracers	49
Implementierung des Log-Tracers	51
traceFileReader.py: Implementierung des Trace File Readers	52
generateTimeline.py: Skript zur chronologischen Darstellung der Lock-Ereignisse	53
magiclockLib/magicLockTypes.py: Sammlung von Klassen die von der Magiclock Implementierung verwendet werden	54
magiclockLib/lockReduction.py: Implementierung des Magiclock-Algorithmus zur Reduzierung von Locks	56
magiclockLib/cycleDetection.py: Implementierung des Magiclock-Algorithmus zur Zyklenerkennung	57
magiclockLib/magiclock.py: Implementierung des Magiclock-Algorithmus	58
generateDeadlockGraph.py: Skript zur Erkennung und Darstellung von potentiellen Deadlocks	59
benchmark_cpu.py: Skript zur Messung der CPU-Laufzeit einer OpenPEARL Anwendung	59
benchmark_memory.py: Skript zur Messung der Speicherauslastung einer OpenPEARL Anwendung	59

1 Motivation

Bei der parallelen Programmierung ist Nebenläufigkeit ein bewusst genutztes Mittel. Die Ausführung von solchen Programmen ist nicht deterministisch. Dies führt dazu, dass Zugriffe auf gemeinsam genutzte Ressourcen synchronisiert werden müssen. Bei der Synchronisierung können zur Laufzeit Deadlocks auftreten. Diese werden in Abschnitt 2.1 beschrieben. Für Entwickler stellen potenzielle Deadlocks ein großes Problem dar, da sie oft erst zur Laufzeit auffallen. Während der Entwicklung kann ein Entwickler Nebenläufigkeitsprobleme, die zu Deadlocks führen können, nur sehr schwer erkennen. Gerade in komplexen Anwendungen in denen viele parallele Aufgaben ausgeführt werden, ist es für den Entwickler nicht mehr möglich potenzielle Deadlocks zu erkennen. Automatisierte Tests können das Erkennen solcher Probleme zwar verbessern, durch die nicht deterministische Ausführung bleiben jedoch viele Probleme unerkannt. Für die Echtzeit-Programmiersprache PEARL gibt es derzeit keine Unterstützung für den Entwickler um solche Probleme effektiv zu erkennen. Um den Entwickler besser unterstützen zu können wird ein Verfahren vorgestellt und für OpenPEARL implementiert, welches die chronologische Abfolge von verwendeten Synchronisationsmitteln darstellen und potenzielle Deadlocks erkennen kann.

In Abschnitt 2.1 wird das grundlegende Verfahren zur Identifizierung von potenziellen Deadlocks vorgestellt. Es wird beschrieben was ein Deadlock ist und wie dynamische Verfahren zur Erkennung von Deadlocks funktionieren. In Abschnitt 2.2 wird die Echtzeit-Programmiersprache PEARL beschrieben. Es wird gezeigt welche Synchronisationsmittel in PEARL existieren und wie diese benutzt werden. In Abschnitt 2.3 wird das OpenPEARL Projekt, der Aufbau der OpenPEARL Umgebung und das Zusammenspiel des Compilers und der Laufzeitumgebung dargestellt. Anschließend wird in Abschnitt 2.4 ein Algorithmus zur Erkennung von potenziellen Deadlocks vorgestellt. Anhand eines Quellcode-Beispiels in PEARL wird der Algorithmus Schritt für Schritt durchlaufen und erläutert.

Das Design zur Implementierung des Algorithmus wird in Kapitel 3 beschrieben. In Abschnitt 3.3 wird der in OpenPEARL umzusetzende Anteil definiert, um die benötigten Informationen für den Algorithmus in einer Trace-Datei zusammenzustellen. Zusätzlich werden die Anforderungen bezüglich der Performanz festgelegt. In Abschnitt 3.3 und Abschnitt 3.4 werden die Designs der Programme zur Visualisierung der chronologischen Belegung der Synchronisationsmittel und der Erkennung von potenziellen Deadlocks definiert.

In Abschnitt 4.1 wird die Implementierung zur Erstellung der Trace-Datei in OpenPEARL beschrieben. Anschließend werden die Implementierungen der Programme zur Analyse und Visualisierung der Trace-Datei in Abschnitt 4.2 und Abschnitt 4.3 vorgestellt.

Die Validierung der in Kapitel 4 erstellten Programme und Funktionen wird in Kapitel 5 durchgeführt. Schwerpunkt sind hier die definierten Performanz Anforderungen.

Abschließend werden in Kapitel 6 offene Punkte und Weiterentwicklungen beschrieben.

2 Analyse

2.1	Deadlockerkennung allgemein	8
2.2	PEARL	10
2.3	OpenPEARL	13
2.4	MagicLock	14

2.1 Deadlockerkennung allgemein

Im Gegensatz zu Single-Threaded-Anwendungen sind Multi-Threaded-Anwendungen nicht deterministisch. Dies kann zu *race conditions* führen. Eine *race condition* tritt zum Beispiel dann auf, wenn zwei Threads einen Zähler jeweils um eins erhöhen wollen¹.

Angenommen der Zähler hat zu Beginn den Wert drei. Beide Threads wollen jetzt nahezu gleichzeitig den Zähler um eins erhöhen. Dazu lesen beide Threads den aktuellen Wert des Zählers, in diesem Fall drei, aus. Anschließend addieren beide eins hinzu und schreiben den neuen Wert, in diesem Fall vier, in den Zähler. Erwartet wurde jedoch der Wert fünf, da beide Threads den Zähler um jeweils eins erhöhen sollten. Um solche *race conditions* zu verhindern werden Synchronisationsmechanismen benötigt.

Eine Möglichkeit um den Zugriff auf eine gemeinsame Ressource zu synchronisieren sind Locks. Ein Lock ist ein exklusiver Zugriff auf ein Objekt, ein sogenanntes Lockobjekt. Das bedeutet, dass während ein Thread einen Lock auf ein Objekt besitzt, andere Threads, welche auf dasselbe Objekt zugreifen wollen, warten müssen bis es freigegeben wurde.

Betrachtet man das Beispiel mit dem Zähler erneut, dieses Mal mit Locks als Synchronisationsmittel, kann es zu folgender Ausführung kommen. Der Zähler hat zu Beginn wieder den Wert drei. Die Threads $T1$ und $T2$ wollen erneut den Zähler nahezu gleichzeitig erhöhen. Dieses Mal versuchen beide das Lockobjekt $L1$ in Besitz zu nehmen. Der Thread $T2$ nimmt $L1$ zuerst in Besitz, daraus folgt $T1$ muss warten. $T2$ liest den aktuellen Wert des Zählers aus, erhöht diesen um eins und schreibt den neuen Wert vier in den Zähler. Anschließend gibt $T2$ das Lockobjekt $L1$ frei. Jetzt erhält der Thread $T1$ den Zugriff auf $L1$ und liest ebenfalls den Zähler, jetzt vier, aus, erhöht diesen und schreibt den neuen Wert fünf in den Zähler. Anschließend gibt $T1$ das Lockobjekt $L1$ frei. Jetzt steht der erwartete Wert fünf im Zähler. Durch das Lockobjekt ist die Ausführung weiterhin nicht deterministisch, da die Reihenfolge auch erst $T1$ und dann $T2$ sein kann. Trotzdem ist die korrekte Erhöhung des Zählers sichergestellt.

Die Verwendung von Locks kann in Verbindung mit der nicht deterministischen Ausführung von Multi-Threaded-Anwendungen zu Problemen führen.

Angenommen es existieren zwei Threads $T1$ und $T2$ und zwei Lockobjekte $L1$ und $L2$. Angenommen $T1$ besitzt $L1$ und zu gleichen Zeit erlangt $T2$ das Lockobjekt $L2$. Wenn

¹ Vgl. „Data races“ in [10, S. 70]

jetzt der Thread $T1$ das Lockobjekt $L2$ anfordert und der Thread $T2$ das Lockobjekt $L1$, kommt es zu einem Deadlock.[vgl. 4, S. 70] Die Ausführung des Programms terminiert nicht, da beide Threads auf den jeweils anderen Thread warten und sich gegenseitig blockieren. Solche potenziellen Deadlocks zu erkennen ist die Aufgabe von statischen und dynamischen Methoden zur Deadlockerkennung. Bei der statischen Deadlockerkennung wird der Quellcode direkt analysiert. Dieses Verfahren wird hier nicht näher betrachtet.

Bei der dynamischen Deadlockerkennung wird die Anwendung zur Laufzeit analysiert und läuft in folgenden drei Schritten ab:[vgl. 1, S. 212–213]

1. Erstellung eines *execution traces*
2. Erstellung eines Graphen basierend auf den Informationen aus dem *execution trace*
3. Suche nach potenziellen Deadlocks durch die Identifizierung von Zyklen innerhalb des Graphen

Ein *execution trace* ist eine Abfolge von Events. Ein Event e_i wird durch eine der folgenden Methoden definiert: starten eines Threads, betreten eines Threads, Inbesitznahme eines Lockobjekts und Freigabe eines Lockobjekts.[vgl. 1, S. 212]

Die Positionsangabe der Events durch Zeilennummer wird hier ignoriert, da sie nicht benötigt wird.²

Das Starten eines neuen Threads ist definiert durch ein Thread-Start-Event:

`s(ausführender Thread, Name des neuen Threads)`[vgl. 1, S. 212]

Zum Beispiel bedeutet `s(main,T1)`, dass der Thread *main* den Thread $T1$ gestartet hat. Das Betreten eines Threads ist definiert durch ein Thread-Join-Event:

`j(ausführender Thread, Name des zu betretenden Threads)`[vgl. 1, S. 212]

Zum Beispiel bedeutet `j(T1,T2)`, dass der Thread $T1$ den Thread $T2$ betreten hat. Die Inbesitznahme eines Lockobjekts kann definiert werden durch ein Lock-Event:

`l(ausführender Thread, Name des Lockobjekts)`[vgl. 1, S. 212]

Zum Beispiel bedeutet `l(T1,L3)`, dass der Thread $T1$ das Lockobjekt $L3$ in Besitz genommen hat. Die Freigabe eines Lockobjekts kann definiert werden durch ein Unlock-Event:

`u(ausführender Thread, Name des Lockobjekts)`[vgl. 1, S. 212]

Zum Beispiel bedeutet `u(T1,L3)`, dass der Thread $T1$ das Lockobjekt $L3$ freigegeben hat. Die Abfolge aller während der Laufzeit des Programms aufgetretenen Events definieren einen möglichen *execution trace* des Programms. Programme welche mit mehreren Threads arbeiten, liefern keine deterministische Abfolge. Jede Ausführung eines solchen Programms kann zu unterschiedlichen *execution traces* führen.

Im zweiten Schritt wird aus dem vorher erzeugten *execution trace* ein Lockgraph erstellt.

² Vgl. „line number lno“ in [1, S. 212]

Ein Lockgraph ist definiert durch:

$$LG = (L, R)$$

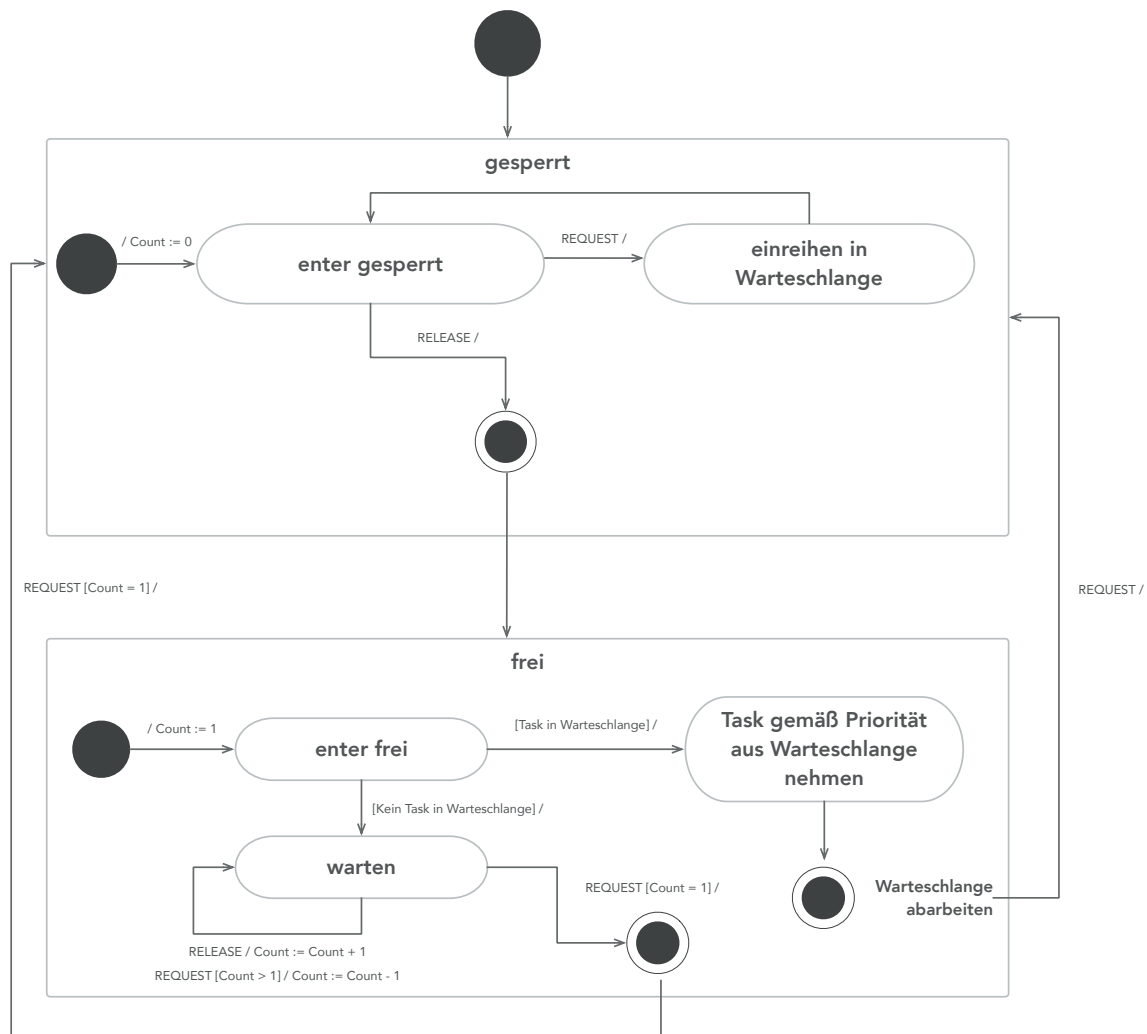
L ist die Menge aller Lockobjekte im *execution trace* und R die Menge aller Lockpaare. Ein Lockpaar ist definiert durch das Tupel $(L1, L2)$ für das gilt: Es existiert ein Thread, welcher das Lockobjekt $L1$ besitzt, während er einen Lock auf $L2$ anfordert.[vgl. 4, S. 72][vgl. 1, S. 213]

2.2 PEARL

Die Programmiersprache PEARL wurde in den 1970er Jahren vom Institut für Regelungstechnik der Universität Hannover entwickelt. PEARL ist eine Abkürzung und steht für „Process and Experiment Automation Realtime Language“. Die Programmiersprache erlaubt eine komfortable, sichere und weitgehend rechnerunabhängige Programmierung von Multitasking- und Echtzeit-Aufgaben.[5] Das Deutsche Institut für Normung standardisierte PEARL mehrmals, unter anderem 1998 in der DIN 66253-2 als PEARL90[7] und zuletzt 2018 als SafePEARL[8]. Nachfolgend werden PEARL und PEARL90 synonym verwendet. In PEARL bezeichnet ein *TASK* eine Aufgabe und wird entweder direkt beim Start des Programms oder durch Signale von anderen Aufgaben gestartet. *TASKs* werden parallel und gemäß ihrer Priorität ausgeführt.[vgl. 6, S. 104] Um mehrere *TASKs* zu synchronisieren gibt es zwei Möglichkeiten: *SEMA* und *BOLT* Variablen.[vgl. 6, S. 120]

Eine *SEMA* Variable ist ein Semaphore und dient als Synchronisationsmittel. Sie kann als Wert nicht negative ganze Zahlen besitzen, wobei null den Zustand „gesperrt“ und positive Zahlen den Zustand „frei“ bedeuten.[vgl. 6, S. 120] Eine *SEMA* Variable hat zu Beginn den Wert null und den Zustand „gesperrt“. Mit dem Befehl *RELEASE* wird eine *SEMA* Variable um den Wert eins erhöht und erhält den Zustand „frei“. Mit dem *REQUEST* Befehl wird der Wert einer *SEMA* Variablen um eins verringert. Ist der Wert einer *SEMA* Variablen null wird der ausführende *TASK* angehalten und in eine Warteschlange eingereiht. Sobald die Variable über den Befehl *RELEASE* wieder freigeben wird, wird der nächste *TASK* in der Warteschlange gemäß seiner Priorität fortgeführt.[vgl. 6, S. 120–121] Das Zustandsdiagramm zur *SEMA* Variable ist in Abb. 2.2.1 dargestellt.

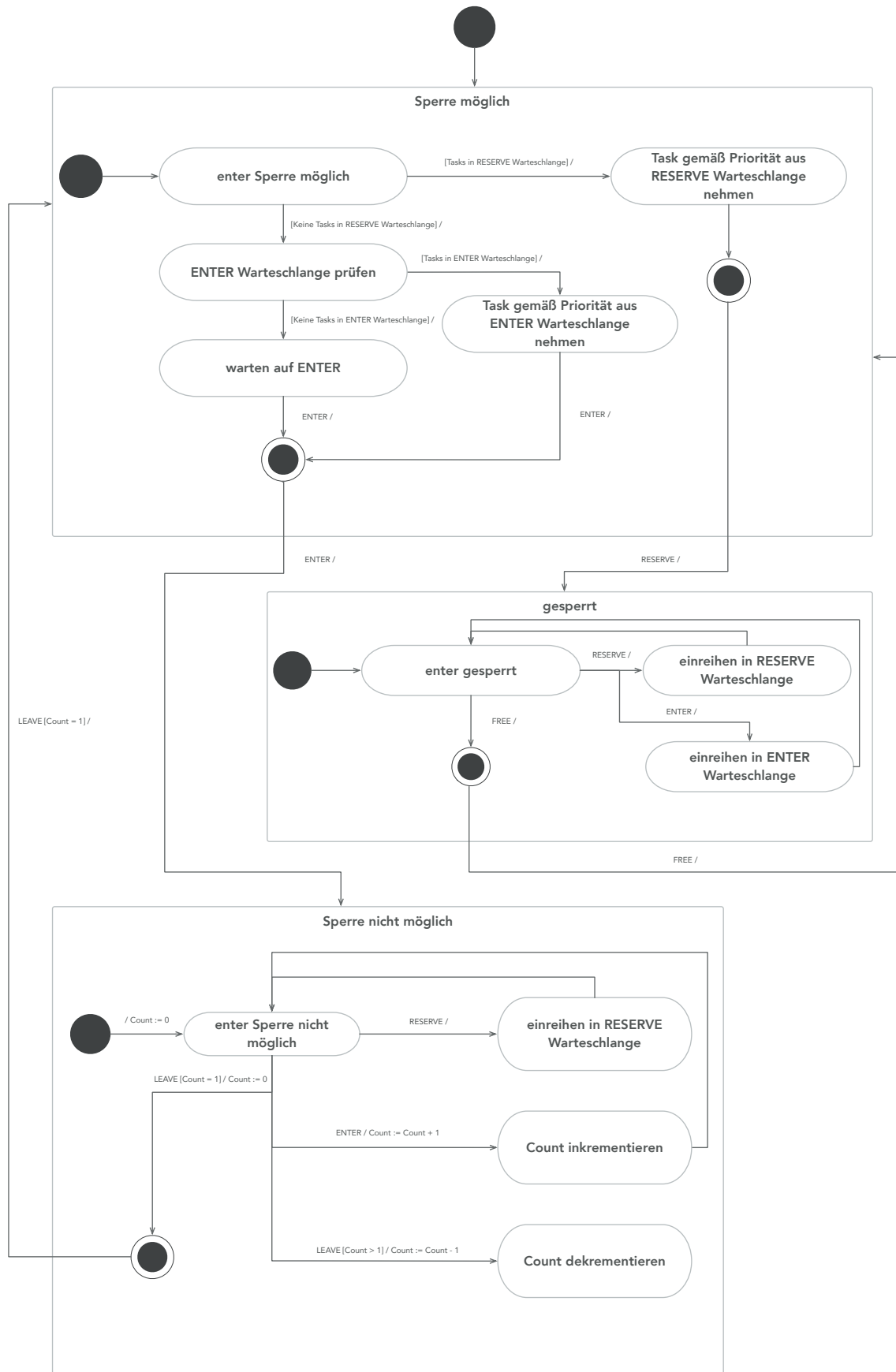
BOLT Variablen haben im Gegensatz zu *SEMA* Variablen drei Zustände: „gesperrt“, „Sperre möglich“ und „Sperre nicht möglich“.[vgl. 6, S. 125] Sie bieten die Möglichkeit exklusive und nicht exklusive Sperren zu ermöglichen. Zum Beispiel können so simultane Lesezugriffe und exklusive Schreibzugriffe realisiert werden. Zu Beginn hat eine *BOLT* Variable den Zustand „Sperre möglich“. Mit dem Befehl *RESERVE* wird ein exklusiver Zugriff auf eine *BOLT* Variable angefordert. Wenn die Variable im Zustand „Sperre möglich“ ist, erhält diese den Zustand „gesperrt“. Ansonsten wird ähnlich zu der *REQUEST* Anweisung für *SEMA* Variablen der ausführende *TASK* angehalten und in eine Warteschlange eingereiht. Mit dem Befehl *FREE* erhält eine *BOLT* Variable den Zustand „Sperre möglich“ und alle *TASKs* in der Warteschlange, welche aufgrund einer *RESERVE* Anweisung warten, werden gemäß ihrer Priorität fortgeführt. Wenn keine *TASKs* in der Warteschlange vorhanden sind, welche auf eine *RESERVE* Anweisung warten, werden die *TASKs* in der Warteschlange gemäß ihrer Priorität fortgeführt, welche aufgrund einer *ENTER* Anweisung



Quelle: Eigene Darstellung

Abbildung 2.2.1: Zustandsdiagramm einer SEMA Variablen

warten. Mit der *ENTER* Anweisung wird ein nicht exklusiver Zugriff angefordert. Wenn die *BOLT* Variable im Zustand „gesperrt“ ist oder ein *TASK* in der Warteschlange existiert, welcher einen exklusiven Zugriff mittels einer *RESERVE* Anweisung angefordert hat, wird der ausführende *TASK* angehalten und in eine Warteschlange eingereiht. Ansonsten erhält die Variable den Zustand „Sperre nicht möglich“, um den exklusiven Zugriff zu verbieten. Zusätzlich wird die Anzahl der benutzenden *TASKs* um eins erhöht. Die *LEAVE* Anweisung verringert die Anzahl der benutzenden *TASKs* um eins, wenn die Anzahl eins entspricht, funktioniert die *LEAVE* Anweisung wie die *FREE* Anweisung.[vgl. 6, S. 125–127] Das Zustandsdiagramm zur *BOLT* Variable ist in Abb. 2.2.2 dargestellt.



Quelle: Eigene Darstellung

Abbildung 2.2.2: Zustandsdiagramm einer BOLT Variablen

2.3 OpenPEARL

Um PEARL Programme auf einem System auszuführen wird ein Compiler benötigt. Das OpenSource Projekt OpenPEARL besteht aus einem Compiler und einer Laufzeitumgebung für PEARL.[11] Unterstützt wird der PEARL90 Standard bis auf einige wenige Unterschiede.[9]

OpenPEARL besteht aus drei wesentlichen Komponenten:[11]

1. Compiler
2. Laufzeitumgebung
3. Inter Module Checker

Der Compiler ist in Java geschrieben und übersetzt PEARL Code in C++ Code. Die Laufzeitumgebung stellt dem Compiler eine API zur Verfügung. Dem Compiler werden durch die API sichere Implementierungen der PEARL Datentypen zur Verfügung gestellt. Zusätzlich enthält die Laufzeitumgebung plattformspezifische Anteile für zum Beispiel die Implementierung für das Scheduling der Tasks. PEARL Anwendungen können aus mehreren Modulen bestehen, welche unabhängig voneinander kompiliert werden. Um Inkonsistenzen bei der Erstellung der Anwendung zu verhindern, prüft der Inter Module Checker die Export- und Importschnittstellen aller Module und deren Kompatibilität.[11]

In Quellcode 2.3.1 ist ein Beispielprogramm in der Programmiersprache PEARL dargestellt. Das Programm startet zwei parallele Aufgaben welche beide eine Zeichenfolge auf der Standardausgabe ausgeben. Der Zugriff auf die Standardausgabe muss dabei synchronisiert erfolgen. In den Zeilen 7 bis 10 werden Variablen definiert, wie zum Beispiel die Ausgabe über die Standardausgabe und die zwei *SEMA* Variablen *L1* und *L2* in den Zeilen 9 und 10. In den Zeilen 12 bis 17 ist ein *TASK* definiert.

Durch die Kennzeichnung *MAIN* wird der *TASK* direkt beim Start des Programms ausgeführt[vgl. 6, S. 28]. Die Befehle *RELEASE* in den Zeilen 13 und 14 erhöhen den Wert der jeweiligen *SEMA* Variable um eins, wodurch der Zustand von „gesperrt“ auf „frei“ gesetzt wird. Anschließend werden in den Zeilen 15 und 16 die *TASKS T2* und *T3* gestartet.

Die *TASKS T2* und *T3* geben in den Zeilen 22 bis 24 und in den Zeilen 32 bis 34 die Zeichenfolge „Hello World T2“ bzw. „Hello World T3“ auf der Standardausgabe aus. Die Synchronisierung des Zugriffs auf die Standardausgabe erfolgt mittels den *SEMA* Variablen *L1* und *L2*. Beide *TASKS* versuchen beide *SEMA* Variablen in Besitz zu nehmen. *T2* versucht in den Zeilen 20 und 21 zuerst *L1* und dann *L2* in Besitz zu nehmen. *T3* versucht in den Zeilen 30 und 31 zuerst *L2* und dann *L1* in Besitz zu nehmen. Da beide *TASKS* parallel laufen, kann es passieren, dass *T2 L1* in Zeile 20 in Besitz nimmt und gleichzeitig *T3* in Zeile 30 *L2* in Besitz nimmt. Beide *SEMA* Variablen haben jetzt den Wert null und den Zustand „gesperrt“. Der *TASK T2* wartet jetzt darauf, dass *L2* freigegeben wird und *T3* wartet darauf, dass *L1* freigegeben wird. Beide *TASKS* warten auf den jeweils anderen und verursachen einen Deadlock.

```

1  MODULE(test);
2
3  SYSTEM;
4      stdout: StdOut;
5
6  PROBLEM;
7      SPC stdout DATION OUT SYSTEM ALPHIC GLOBAL;
8      DCL termout DATION OUT ALPHIC DIM(*,80) FORWARD STREAM CREATED(stdout);
9      DCL L1 SEMA;
10     DCL L2 SEMA;
11
12  T1: TASK MAIN;
13     RELEASE L1;
14     RELEASE L2;
15     ACTIVATE T2;
16     ACTIVATE T3;
17  END;
18
19  T2: TASK;
20     REQUEST L1;
21     REQUEST L2;
22     OPEN termout;
23     PUT 'Hello World T2' TO termout BY A, SKIP;
24     CLOSE termout;
25     RELEASE L2;
26     RELEASE L1;
27  END;
28
29  T3: TASK;
30     REQUEST L2;
31     REQUEST L1;
32     OPEN termout;
33     PUT 'Hello World T3' TO termout BY A, SKIP;
34     CLOSE termout;
35     RELEASE L1;
36     RELEASE L2;
37  END;
38
39  MODEND;

```

Quellcode 2.3.1: Beispiel einer OpenPEARL Anwendung mit einem potenziellen Deadlock

2.4 MagicLock

MagicLock ist ein Algorithmus zur dynamischen Deadlockerkennung. Während der Entwicklung wurde der Fokus auf die Skalierung und Effizienz des Algorithmus gesetzt. Ziel war es mit großen Multithreaded Anwendungen skalieren und diese effizient analysieren zu können.[vgl. 2, S. 1]

MagicLock analysiert einen *execution trace* einer Programmausführung ohne Deadlocks [vgl. 2, S. 4]. Ein möglicher *execution trace* von dem Beispielprogramm aus Quellcode 2.3.1 ist:

$$\sigma = s(\text{main}, T1), u(T1, L1), u(T1, L2), s(T1, T2), s(T1, T3), l(T2, L1), l(T2, L2), \\ u(T2, L2), u(T2, L1), l(T3, L2), l(T3, L1), u(T3, L1), u(T3, L2)$$

Der *execution trace* in MagicLock wird durch eine Lock-Dependency-Relation definiert. Eine Lock-Dependency-Relation D besteht aus einer Sequenz von Lock-Dependencies. Eine Lock-Dependency ist ein Triple $r = (t, m, L)$ in dem t ein Thread ist, m ein Lockobjekt und L eine Menge von Lockobjekten. Das Triple sagt aus, dass der Thread t das Lockobjekt m in Besitz nimmt, während er jedes Lockobjekt in L besitzt.[vgl. 2, S. 3]

Bei einem Thread-Start-Event wird ein neuer Thread-Identifizier und eine leere Menge an

Locks für den neu erzeugten Thread erstellt [vgl. 2, S. 4]. Zum Beispiel wird bei den Event $s(main, T1)$ ein neuer Thread-Identifizier für $T1$ erzeugt und eine leere Menge L_{T1} .

Bei einem Lock-Event $l(T2, L1)$ wird zuerst die Lock-Dependency $(T2, L1, L_{T2})$ an den *execution trace* angehängt und anschließend $L1$ in die Menge der Locks L_{T2} eingefügt. Bei einem Unlock-Event $u(T2, L2)$ wird das Lockobjekt $L2$ aus der Menge L_{T2} entfernt.[vgl. 2, S. 4]

Daraus folgt die Lock-Dependency-Relation:

$$D_\sigma = (T2, L1, \{\}), (T2, L2, \{L1\}), (T3, L2, \{\}), (T3, L1, \{L2\})$$

Anschließend wird ein reduzierter *execution trace* erzeugt. Dazu verwendet MagicLock einen Algorithmus zur Reduzierung von Lockobjekten im *execution trace*. Der Algorithmus entfernt alle Lockobjekte aus der Menge aller Lockobjekte $Locks$ aus D_σ die entweder keine eingehenden $indegree(m) = 0$ oder keine ausgehenden $outdegree(m) = 0$ Kanten im Lockgraph besitzen. Die Annahme ist, dass ein Lockobjekt nur Teil eines Zyklus sein kann, wenn dieses mindestens eine eingehende und mindestens eine ausgehende Kante besitzt. Zusätzlich werden alle Lockobjekte entfernt, welche nur von einem einzigen Thread in Besitz genommen bzw. freigegeben wurden. Wenn nur ein Thread ein Lockobjekt benutzt, kann dieses Lockobjekt nicht Teil eines Deadlocks sein. Mit den reduzierten Lockobjekten wird im nächsten Schritt die Zyklensuche vorbereitet.[vgl. 2, S. 4]

Die noch vorhandenen Lock-Dependencies werden in Partitionen basierend auf ihrer Thread ID unterteilt und anschließend sortiert. Für jeden Thread wird eine Partition erstellt mit allen Lock-Dependencies mit (t_i, m, L) wobei t_i der jeweilige Thread der Partition ist. Zusätzlich werden gleiche Lock-Dependencies in Gruppen eingeteilt. Für gleiche Lock-Dependencies muss dann immer nur ein Element aus der Gruppe geprüft werden. Wenn ein Zyklus gefunden wurde, wurde gleichzeitig ein Zyklus für alle Lock-Dependencies in der Gruppe gefunden. Wenn kein Zyklus gefunden wurde, wird dies gleichzeitig für alle anderen Elemente in der Gruppe angenommen.[vgl. 2, S. 8–9]

Zwei Lock-Dependencies sind gleich wenn folgendes gilt:[vgl. 2, S. 8]

$$\begin{aligned} &\text{Gegeben sind zwei Lock-Dependencies } r_1 = (t_1, m_1, L_1) \text{ und } r_2 = (t_2, m_2, L_2): \\ &r_1 = r_2 \Leftrightarrow t_1 = t_2 \wedge m_1 = m_2 \wedge L_1 = L_2 \end{aligned}$$

Anschließend werden die Partitionen gegeneinander auf Lock-Dependency-Chains geprüft [vgl. 2, S. 8]. Eine Lock-Dependency-Chain ist eine Sequenz von Lock-Dependencies für die gilt:[vgl. 2, S. 3]

$$\begin{aligned} d_{chain} = (r_1, r_2, \dots, r_k) \text{ mit } r_i = (t_i, m_i, L_i), \text{ wenn } m_1 \in L_2 \dots m_{k-1} \in L_k, t_i \neq \\ t_j \text{ und } L_i \cap L_j = \emptyset \text{ für } 1 \leq i, j \leq k (i \neq j) \end{aligned}$$

Eine Zyklische-Lock-Dependency-Chain ist eine Lock-Dependency-Chain für die zusätzlich gilt:[vgl. 2, S. 3]

$$m_k \in L_1$$

Zum Beispiel ist die Lock-Dependency Sequenz $d = (t_1, l_2, \{l_1\}), (t_2, l_1, \{l_2\})$ eine Zyklische-Lock-Dependency-Chain. Jede Zyklische-Lock-Dependency-Chain repräsentiert einen potenziellen Deadlock[vgl. 2, S. 3].

3 Design

3.1 Übersicht	16
3.2 Erzeugung der Trace-Datei	16
3.3 Analysieren der Trace-Datei	18
3.4 Erweiterung: Potenzielle Deadlocks	19

3.1 Übersicht

Die OpenPEARL Laufzeitumgebung wird um eine Trace-Funktionalität für *SEMA* Objekte erweitert. Dazu wird die *SEMA* Implementierung in der OpenPEARL Laufzeitumgebung angepasst. Diese Trace-Funktionalität wird über eine Umgebungsvariable gesteuert. Das Schreiben auf die Festplatte ist sehr zeitintensiv, deswegen werden Lockereignisse zwischengespeichert und erst beim Erreichen eines definierten Werts in die Trace-Datei geschrieben. Dieser Wert wird ebenfalls über eine Umgebungsvariable definiert.

Die erzeugte Trace-Datei dient als Eingabe für die Anwendung zur Generierung und Darstellung der chronologischen Verwendung der *SEMA* Objekte.

Zusätzlich wird die Trace-Datei mit Hilfe des MagicLock¹ Algorithmus nach potentiellen Deadlocks durchsucht. Potentielle Deadlocks werden anschließend als gerichteter Graph dargestellt.

3.2 Erzeugung der Trace-Datei

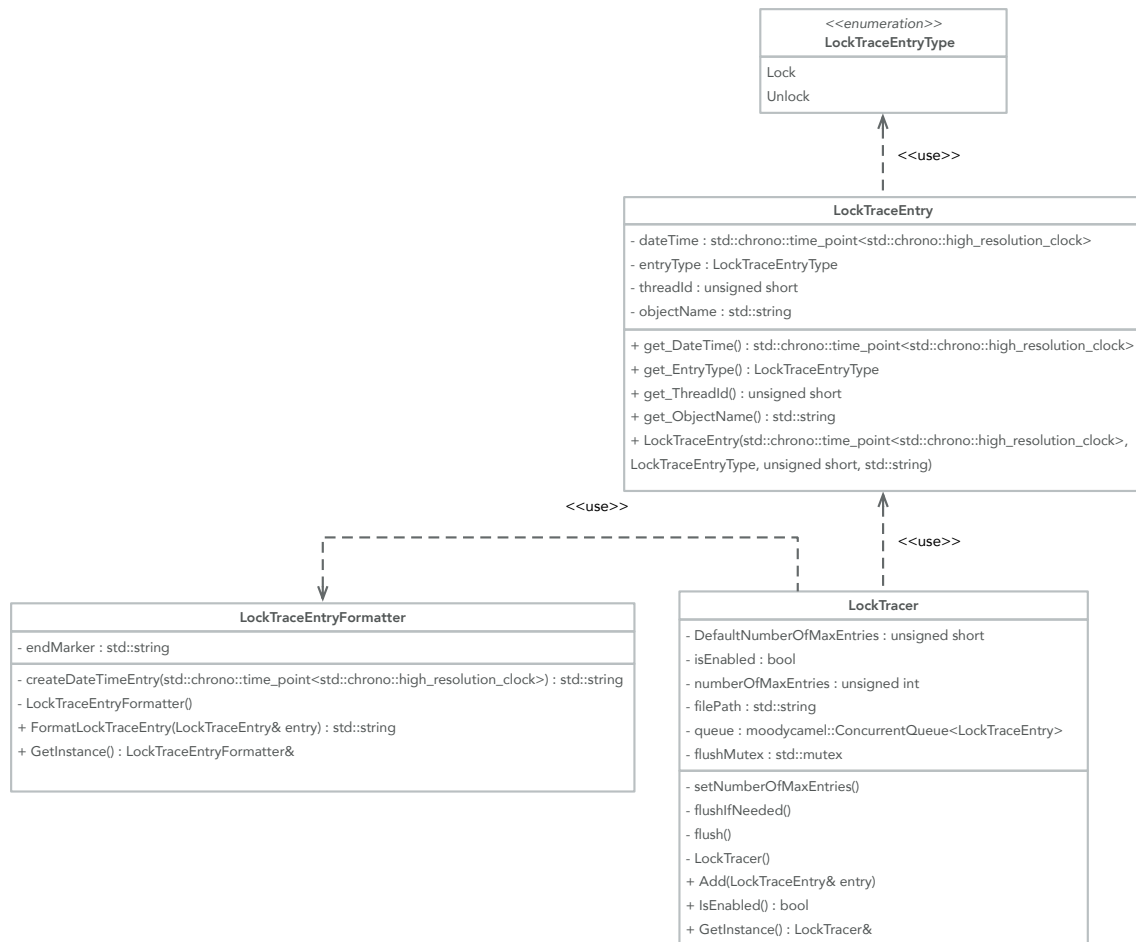
In der Trace-Datei werden alle benötigten Informationen geschrieben, um potentielle Deadlocks zu erkennen:

1. Der genaue Zeitpunkt des Ereignisses
2. Die Art des Ereignisses (Lock, Unlock)
3. Der Name des ausführenden Threads
4. Der Name des verwendeten Lockobjekts

In Abb. 3.2.1 sind die benötigten Klassen dargestellt. Die notwendigen Informationen für einen Trace-Eintrag werden in der Klasse `LockTraceEntry` gehalten. Für den Zeitpunkt wird der Typ `chrono::time_point` vom Typ `chrono::high_resolution_clock` verwendet. Der Typ `chrono::high_resolution_clock` stellt einen Zeitpunkt mit der höchstmöglichen Genauigkeit der jeweiligen Implementierung dar². Für die spätere Visualisierung ist

¹ Siehe Abschnitt 2.4

² C++ ab der Version 11



Quelle: Eigene Darstellung

Abbildung 3.2.1: UML Klassendiagramm für Trace-Funktionalität

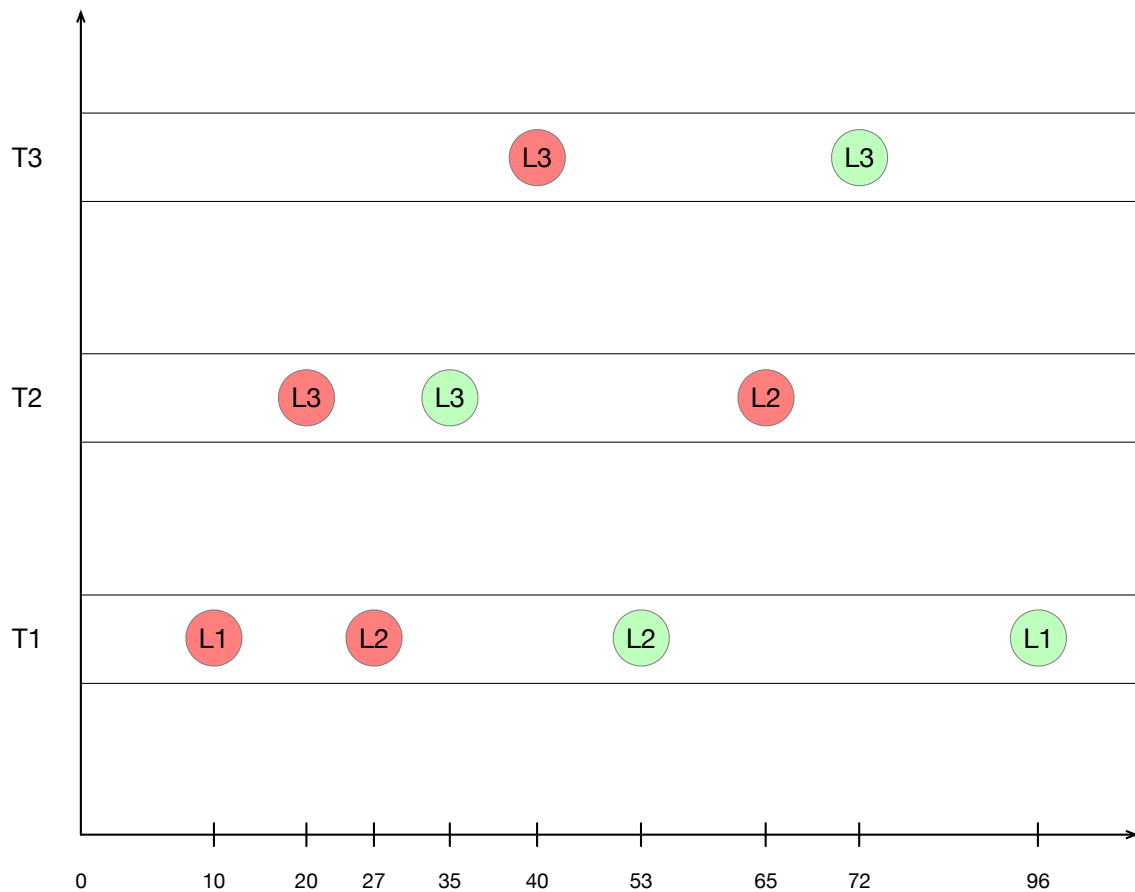
eine hohe Genauigkeit notwendig, um nahezu parallel aufgetretene Lockereignisse chronologisch getrennt visualisieren zu können. Die Klasse **LockTraceEntryFormatter** erstellt mit der Methode **FormatLockTraceEntry** aus einem **LockTraceEntry** eine Zeichenkette, die einer Zeile in der Trace-Datei entspricht. Diese Klasse wird als Singleton implementiert, da zur Laufzeit immer nur genau eine Instanz benötigt wird. Die Klasse **LockTracer** stellt die Methode **Add** zur Verfügung, welche von der OpenPEARL Laufzeitumgebung aufgerufen wird. Mit Hilfe der Methode können Lockereignisse erstellt werden. Die Methode **IsEnabled** gibt den aktuellen Zustand der **LockTracer** Instanz zurück. Mithilfe dieser Methode kann ein Aufrufer prüfen, ob die Trace-Funktionalität aktiviert ist. Ist dies nicht der Fall, muss die **Add** Methode nicht aufgerufen und somit auch kein **LockTraceEntry** Objekt erzeugt werden. Die Klasse wird ebenfalls als Singleton implementiert, damit nur eine Instanz zur Laufzeit verwendet werden kann. Das Speichern der Ereignisse in die Trace-Datei ist kostspielig und soll daher nicht für jeden Eintrag gemacht werden. Die Klasse **LockTracer** reiht dazu die einzelnen Lockereignisse, welche über die **Add** Methode hinzugefügt werden, in eine Warteschlange ein. Sobald eine spezifizierte Anzahl erreicht ist, wird die Warteschlange geleert und in die Trace-Datei geschrieben. Die Anzahl kann über die Umgebungsvariable **OpenPEARL_LockTracer_MaxEntries** spezifiziert werden. Die Umgebungsvariable wird bei der Initialisierung der **LockTracer** Implementierung ausgelesen und in der Variable **numberOfMaxEntries** gespeichert. Das Hinzufügen der Ereignisse in die Warteschlange

kann parallel erfolgen und muss daher Thread sicher implementiert werden. Eine Möglichkeit wäre, die einzelnen Zugriffe über einen Lock zu synchronisieren. Dies würde die Laufzeit der Anwendung stark negativ beeinflussen. Deswegen wird die lock freie Implementierung einer Warteschlange aus [3] verwendet. Die Warteschlange garantiert eine Thread sichere Implementierung, aber keine Sortierung innerhalb der Warteschlange für mehrere Produzenten.³ Da es zur Laufzeit nur eine einzige Instanz der `LockTracer` Klasse gibt, ist die Reihenfolge innerhalb der Warteschlange dennoch garantiert. Der Dateipfad zur Speicherung der Trace-Datei wird über die Umgebungsvariable `OpenPEARL_LockTracer_Path` definiert und bei der Initialisierung der `LockTracer` Implementierung in der Variable `filePath` gespeichert. Die dritte Umgebungsvariable `OpenPEARL_LockTracer_Enabled` wird zur Aktivierung der Trace-Funktionalität verwendet. Wenn die Umgebungsvariable gesetzt ist und den Wert `true` hat, wird die Trace-Funktionalität aktiviert. Ansonsten werden alle Aufrufe zur `Add` Methode direkt über eine `return` Anweisung beendet. Dadurch wird die Laufzeit der Anwendung bei deaktivierter Trace-Funktionalität nicht beeinflusst. In der OpenPEARL Laufzeitumgebung werden die *REQUEST* und *RELEASE* Anweisungen in der Semaphore Implementierung unter `runtime/common/Semaphore.cc` implementiert. Bei einer Erhöhung auf eins oder einer Verringerung auf null eines Semaphors muss ein Lockerereignis erzeugt werden. Bei einer Erhöhung auf eins muss der `LockTraceEntryType Unlock` bei einer Verringerung auf null der `LockTraceEntryType Lock` verwendet werden. Die Klassen für die Implementierung des LockTracers müssen bei der Kompilierung der OpenPEARL Laufzeitumgebung mit einbezogen werden. In der Datei `runtime/common/Files.common` sind alle Dateien aufgeführt, welche bei der Kompilierung einbezogen werden. Dort müssen die Dateien, die aus Abb. 3.2.1 entstehen eingetragen werden.

3.3 Analysieren der Trace-Datei

Als Eingabe dient die in Abschnitt 3.2 erzeugte Trace-Datei. Die chronologische Darstellung wird, wie in Abb. 3.3.1 skizziert, über einen zwei dimensional Graphen realisiert. Die Ordinate bildet die Zeit ab, wobei nur das Delta in Mikrosekunden zwischen den einzelnen Ereignissen dargestellt wird. Für jeden Thread wird ein Eintrag auf der Abszisse gemacht. Es wird zwischen zwei Ereignissen unterschieden. Wird ein *SEMA* Objekt in Besitz genommen wird ein roter und für das Freigeben eines *SEMA* Objekts ein grüner Kreis gezeichnet. Die Beschriftung eines Kreises enthält den Namen des *SEMA* Objekts.

³ Vgl. „Reasons not to use“ in [3]

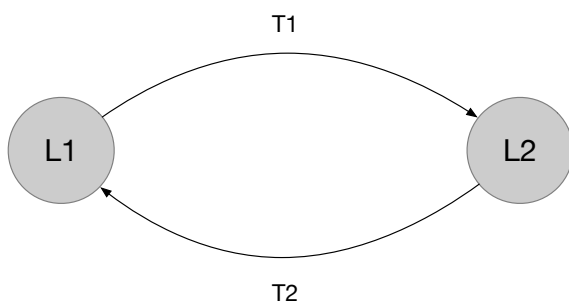


Quelle: Eigene Darstellung

Abbildung 3.3.1: Visualisierung der chronologischen Verwendung von *SEMA* Objekten

3.4 Erweiterung: Potenzielle Deadlocks

Der in Abschnitt 2.4 beschriebene Algorithmus wird dazu verwendet, um potentielle Deadlocks in der aus Abschnitt 3.2 erstellten Trace-Datei zu finden.



Quelle: Eigene Darstellung

Abbildung 3.4.1: Visualisierung eines potentiellen Deadlocks

Potenzielle Deadlocks werden wie in Abb. 3.4.1 skizziert als gerichteter Graph dargestellt. Knoten repräsentieren Lockobjekte, Kanten repräsentieren die Inbesitznahme eines Lockobjekts und die Beschriftung an einer Kante bezeichnet den ausführenden Thread. In dem skizzierten Beispiel existiert ein potentieller Deadlock zwischen den Threads *T1* und *T2*. Der Thread *T1* nimmt das Lockobjekt *L2* in Besitz während dieser bereits *L1* besitzt. Dies ist zu erkennen an der Kante vom Knoten *L1* zum Knoten *L2* mit der Bezeichnung

T1. Der Thread *T2* nimmt in dem Beispiel das Lockobjekt *L1* in Besitz während dieser bereits *L2* besitzt. Dies kann zu einem Deadlock führen, wenn *T1* das Lockobjekt *L1* und *T2* das Lockobjekt *L2* gleichzeitig besitzen.

4 Implementierung

4.1	Trace Funktion	21
4.2	Analyse Programm	23
4.3	Visualisierung von potenziellen Deadlocks	24

4.1 Trace Funktion

Aus dem in Abb. 3.2.1 vorgestellten UML-Diagramm werden drei Klassen und eine Enumeration in C++ implementiert. `LockTraceEntryType` wird als `enum` mit den Werten `Lock` und `Unlock` implementiert. Die Klasse `LockTraceEntry` wird als einfache Datenklasse mit Get-Methoden für jedes Attribut implementiert. Die beiden Klassen `LockTraceEntryFormatter` und `LockTracer` werden als Singleton implementiert. Die Implementierung für die Berechnung des genauen Zeitpunkts ist in Quellcode 4.1.1 dargestellt.

```

31      std::string LockTraceEntryFormatter::createDateTimeEntry(std::chrono::time_point<std::chr_
    ↪      ono::high_resolution_clock> timePoint)
    ↪      {
32          return std::to_string(std::chrono::time_point_cast<std::chrono::microseconds>(timePoi_
    ↪      nt).time_since_epoch().count()) +
    ↪      ":";
33      }

```

Quellcode 4.1.1: Auszug aus `LockTraceEntryFormatter.cc`: Berechnung des Zeitpunkts

Der `chrono::time_point` wird in Mikrosekunden seit dem 01.01.1970 umgerechnet. Mikrosekunden wurden als Genauigkeit gewählt, weil Millisekunden nicht hochauflösend genug sind, um die chronologische Verwendung von Lockobjekten darzustellen. Eine höhere Auflösung als Mikrosekunden würde die Darstellung hingegen zu unübersichtlich machen. In Quellcode 4.1.2 ist ein Auszug aus der Implementierung der Klasse `LockTracer` dargestellt. Die `GetInstance()` Methode stellt sicher, dass nur eine Instanz der Klasse zur Laufzeit existiert¹. Im Konstruktor wird einmalig geprüft, ob die notwendigen Umgebungsvariablen existieren und korrekte Werte aufweisen. Wenn alle Variablen vorhanden sind und die Variable `NameOfEnvironmentVariableEnabled` den Wert `true` hat, wird die Trace-Funktionalität aktiviert. In der Methode `Add(LockTraceEntry& entry)` wird zuerst geprüft, ob die Trace-Funktionalität aktiviert ist. Ist dies nicht der Fall wird die Methode sofort beendet, dadurch wird die Laufzeit der OpenPEARL Anwendung bei deaktivierter Trace-Funktionalität nicht negativ beeinflusst². Es wird sichergestellt, dass beim Beenden der Anwendung alle noch vorhandenen Lockereignisse in die Trace-Datei geschrieben

¹ Ab C++ Version 11

² Der einmalige Aufruf des Konstruktors, der Aufruf der Add Methode und das Prüfen eines booleschen Wertes werden hier ignoriert.

```

11     LockTracer::LockTracer() {
12         isEnabled = false;
13
14         char* envVar = std::getenv(NameOfEnvironmentVariableEnabled);
15         if(envVar != NULL && strcmp(envVar, "true") == 0) {
16             envVar = std::getenv(NameOfEnvironmentVariablePath);
17             if(envVar != NULL && directoryExists(envVar)) {
18                 std::time_t t = std::time(nullptr);
19                 std::tm tm = *std::localtime(&t);
20
21                 std::ostringstream oss;
22                 oss << std::put_time(&tm, "%Y-%m-%d_%H-%M.log");
23                 std::string str = oss.str();
24
25                 filePath = std::string(envVar) + str;
26
27                 setNumberOfMaxEntries();
28                 isEnabled = true;
29             }
30         }
31     }
32
33     LockTracer& LockTracer::GetInstance()
34     {
35         static LockTracer instance;
36         return instance;
37     }
38
39     void LockTracer::Add(LockTraceEntry& entry) {
40         if(isEnabled == false) {
41             return;
42         }
43
44         queue.enqueue(entry);
45         LockTracer::flushIfNeeded();
46     }
47
48     bool LockTracer::IsEnabled() {
49         return isEnabled;
50     }
51
52     LockTracer::~LockTracer() {
53         if(isEnabled == false) {
54             return;
55         }
56
57         LockTracer::flush();
58     }

```

Quellcode 4.1.2: LockTracer.cc: Auszug aus der Implementierung des LockTracers

werden. Dies erfolgt über den Destruktor der Klasse in Zeile 52. Die Integration in die OpenPEARL Laufzeitumgebung erfolgt durch das Hinzufügen der benötigten Dateien in die *Files.common* Datei. Dazu wird die Zeile 86 um Quellcode 4.1.3 erweitert. Im letzten Schritt wird die *Semaphore.cc* Implementierung angepasst. Für jede Erhöhung auf eins eines Semaphore-Objekts wird die Methode *TraceUnlock*, für jede Verringerung auf null die Methode *TraceLock* aus Quellcode 4.1.4 aufgerufen.

```

86     SampleBasicDation.cc \
87     lockTracer/LockTracer.cc lockTracer/LockTraceEntry.cc \
88     lockTracer/LockTraceEntryFormatter.cc

```

Quellcode 4.1.3: Files.common: Auszug aus der Auflistung der zu kompilierenden Dateien

```

93     void TraceLock(char* tName, const char* semaName) {
94         LockTracer& lockTracer = LockTracer::GetInstance();
95         if(lockTracer.IsEnabled() == false) {
96             return;
97         }
98
99         LockTraceEntry entry = LockTraceEntry(std::chrono::high_resolution_clock::now(),
100         ↪ LockTraceEntryType::Lock, tName, semaName);
101         lockTracer.Add(entry);
102     }
103
104     void TraceUnlock(char* tName, const char* semaName) {
105         LockTracer& lockTracer = LockTracer::GetInstance();
106         if(lockTracer.IsEnabled() == false) {
107             return;
108         }
109
110         LockTraceEntry entry = LockTraceEntry(std::chrono::high_resolution_clock::now(),
111         ↪ LockTraceEntryType::Unlock, tName, semaName);
112         lockTracer.Add(entry);
113     }

```

Quellcode 4.1.4: Semaphore.cc: Auszug aus der Semaphore Implementierung in der OpenPEARL Laufzeitumgebung

4.2 Analyse Programm

Die Implementierung für die Analyse und die chronologische Darstellung der Verwendung von *SEMA* Objekten wird in Python³ durchgeführt. Im ersten Schritt wird die Trace-Datei aus Abschnitt 3.2 ausgelesen. Die einzelnen Lockereignisse werden von der Klasse *LockAction* repräsentiert, welche in Quellcode 4.2.1 dargestellt ist.

```

1  class LockActionType:
2      LOCK = 1
3      UNLOCK = 2
4
5  class LockAction(object):
6      def __init__(self, timeStamp, threadName, lockObjectName, actionType):
7          self.timeStamp = timeStamp
8          self.threadName = threadName
9          self.lockObjectName = lockObjectName
10         self.actionType = actionType

```

Quellcode 4.2.1: traceFileReader.py: Auszug aus der Implementierung des Trace-Datei Parsers

Anschließend wird aus den ausgelesenen Lockereignissen ein Graph erstellt. Dazu wird die Python-Bibliothek Matplotlib⁴ verwendet. In Quellcode 4.2.2 ist die Erzeugung des Graphen dargestellt. In den Zeilen 23 bis 25 werden die Threads und Zeitstempel in Hashsets gespeichert und anschließend in den Zeilen 27 und 28 sortiert. In den Zeilen 30

³ Python Version 3.7.3.

⁴ Matplotlib Version 3.1.3.

```

16     threads = set([])
17     times = set([])
18     x = []
19     y = []
20     c = []
21     texts = []
22
23     for lockAction in lockActions:
24         threads.add(lockAction.threadName)
25         times.add(lockAction.timeStamp)
26
27     threads = sorted(list(threads))
28     times = sorted(list(times))
29
30     for lockAction in lockActions:
31         xValue = int(lockAction.timeStamp) - int(times[0])
32         offset = 0
33         if xValue in x:
34             offset = 0.2 * x.count(xValue)
35         x.append(xValue)
36         y.append(threads.index(lockAction.threadName) + offset)
37         c.append(get_Color(lockAction.actionType))
38         texts.append(lockAction.lockObjectName)

```

Quellcode 4.2.2: generateTimeline.py: Auszug aus der Bestimmung der einzelnen Werte für den Graphen

bis 38 werden die Werte für die einzelnen Lockereignisse bestimmt. Um die Threads auf der Ordinate abzubilden wird in Zeile 36 der Index des jeweiligen Threads im sortierten Hashset verwendet. Dadurch entspricht jeder ganzzahlige Wert auf der Ordinate einem Thread. Falls es mehrere Einträge für eine Thread mit dem gleichen Zeitstempel gibt, wird in den Zeilen 33 und 34 ein Offset berechnet und in der Zeile 36 hinzugefügt, um eine Überlappung zu verhindern. Für den Wert auf der Abszisse wird das Delta der Zeitstempel zwischen dem jeweiligen Lockereignis und dem niedrigsten Zeitstempel in der Zeile 31 berechnet. In Zeile 37 wird die Farbe des Lockereignisses und in Zeile 38 die Beschriftung festgelegt. Aus den einzelnen Werten wird in Quellcode 4.2.3 der Graph mit Matplotlib erstellt und kann dann mit dem Befehl `plt.show()` dargestellt werden.

```

42     plt.scatter(x, y, c=c, alpha=0.85, s=100)
43     for i, txt in enumerate(texts):
44         plt.annotate(txt, (x[i], y[i] + 0.1))

```

Quellcode 4.2.3: generateTimeline.py: Auszug aus der Erzeugung des Graphen

4.3 Visualisierung von potenziellen Deadlocks

Im ersten Schritt wird der MagicLock-Algorithmus in Python⁵ implementiert. Das Auslesen der Trace-Datei wurde bereits in Abschnitt 4.2 durchgeführt und wird hier wiederverwendet. Nach dem Auslesen werden die Lockereignisse in eine Lock-Dependency-Relation⁶ überführt. Die Klassen sind in Quellcode 4.3.1 und Quellcode 4.3.2 dargestellt. Eine `LockDependency-Relation` enthält ein Hashset mit allen Lockobjekten, eine Liste aller Threads und eine

⁵ Python Version 3.7.3.

⁶ Siehe Abschnitt 2.4

```

45 class LockDependency(object):
46     def __init__(self, threadName, lockObjectName, currentlyOwnedLockObjectNames):
47         self.threadName = threadName
48         self.lockObjectName = lockObjectName
49         self.currentlyOwnedLockObjectNames = currentlyOwnedLockObjectNames

```

Quellcode 4.3.1: magiclockLib/magiclockTypes.py: Repräsentation einer *Lock Dependency* aus Magiclock[2, S. 3]

```

67 class LockDependencyRelation(object):
68     def __init__(self):
69         self.locks = set()
70         self.threads = []
71         self.lockDependencies = []
72
73     def add(self, lockDependency):
74         self.lockDependencies.append(lockDependency)
75         self.locks.add(lockDependency.lockObjectName)
76         if lockDependency.threadName not in self.threads:
77             self.threads.append(lockDependency.threadName)

```

Quellcode 4.3.2: magiclockLib/magiclockTypes.py: Repräsentation einer *Lock Dependency Relation* aus Magiclock[2, S. 3]

Liste der einzelnen *LockDependency* Objekten. Für die Threads wurde eine Liste gewählt, damit die Ergebnisse deterministisch ausfallen. Die Liste der Threads wird später in einer Schleife durchlaufen. Dies führt bei einem Hashset, aufgrund der zufälligen Sortierung, zu nicht deterministischen Ergebnissen.

Im zweiten Schritt wird die *LockDependencyRelation* reduziert. Dazu werden die einzelnen Lockobjekte klassifiziert durch die Zuweisung in eine der folgenden Mengen.[vgl. 2, S. 4]

1. *Independent-set* = $\{m \mid m \in Locks, indegree(m) = 0 \wedge outdegree(m) = 0\}$
2. *Intermediate-set* = $\{m \mid m \in Locks, (indegree(m) = 0 \vee outdegree(m) = 0) \wedge \neg(indegree(m) = 0 \wedge outdegree(m) = 0)\}$
3. *Inner-set* = $\{m \mid m \in Locks, (\exists(t, m, L) \in D, \forall n \in L, n \in Intermediate-set \cup Inner-set) \vee (\exists(t, n, L) \in D, m \in L \wedge n \in Intermediate-set \cup Inner-set)\}$
4. *Cyclic-set* = $\{m \mid m \in Locks, m \notin Independent-set \cup Intermediate-set \cup Inner-set\}$

Dazu werden die Algorithmen *LockReduction(D)*, *InitClassification(D)* und *LockClassification(D)* implementiert⁷. Als erstes wird in Quellcode 4.3.3 mit *InitClassification(D)* eine Datenstruktur initialisiert. Für jede *LockDependency* werden die eingehenden und ausgehenden Kanten sowie der Mode bestimmt. Der Mode ist entweder der Name des Threads, der als einziger Thread in der *LockDependencyRelation* Zugriffe auf das Lockobjekt ausführt oder -1. Die erstellte Datenstruktur ist in Quellcode 4.3.4 dargestellt. Die Klassifizierung der Lockobjekte erfolgt in Quellcode 4.3.5. Alle Lockobjekte ohne eingehende und ausgehende Kanten werden in den Zeilen 39 und 40 in das *Independent-set* eingefügt. Diese Lockobjekte können ignoriert werden, da ein potenzieller Deadlock mindestens eine ausgehende und mindestens eine eingehende Kante besitzen muss. Lockobjekte ohne eingehende oder ohne ausgehende Kanten werden in den Zeilen 42 bis 44 in das *Intermediate-set*

⁷ Entspricht den Algorithmen 1, 2 und 3 aus MagicLock[2, S. 5]

```

17 def init_Classification(D):
18     initClassification = magiclockTypes.InitClassification()
19     for m in D.locks:
20         initClassification.indegree[m] = 0
21         initClassification.outdegree[m] = 0
22         initClassification.mode[m] = 0
23     for d in D.lockDependencies:
24         if mode(d.lockObjectName, D) != d.threadName:
25             initClassification.mode[d.lockObjectName] = -1
26         else:
27             initClassification.mode[d.lockObjectName] = d.threadName
28     for n in d.currentlyOwnedLockObjectNames:
29         initClassification.indegree[d.lockObjectName] += 1
30         initClassification.outdegree[n] += 1
31         initClassification.edgesFromTo[n][d.lockObjectName] += 1
32     return initClassification

```

Quellcode 4.3.3: magiclockLib/lockReduction.py: Implementierung des *InitClassification(D)* Algorithmus aus Magiclock[2, S. 5]

```

3 class InitClassification(object):
4     def __init__(self):
5         self.indegree = {}
6         self.outdegree = {}
7         self.mode = {}
8         self.edgesFromTo = defaultdict(lambda: defaultdict(int))

```

Quellcode 4.3.4: magiclockLib/magiclockTypes.py: Datenstruktur der *init_Classification(D)* Methode

eingefügt. Zusätzlich werden diese auf den Stack *s* gelegt. In den Zeilen 46 bis 69 wird die Klassifizierung für das *Inner-set* durchgeführt. Das oberste Lockobjekt *m* vom Stack *s* wird entfernt und überprüft. Dies wird wiederholt bis der Stack keine Lockobjekte mehr enthält. Wenn das Lockobjekt *m* keine eingehenden Kanten besitzt, werden alle anderen Lockobjekte mit eingehenden Kanten in den Zeilen 49 bis 58 durchlaufen. Die eingehenden Kanten des Lockobjekts *n* können dann um die Anzahl der Kanten von dem Lockobjekt *m* zu *n* reduziert werden. Wenn das Lockobjekt *n* anschließend selbst keine eingehenden Kanten mehr besitzt, wird es auf den Stack *s* gelegt und in das *Intermediate-set* eingefügt. Anschließend werden in den Zeilen 57 und 58 die Kanten von dem Lockobjekt *m* zu *n* und alle ausgehenden Kanten von *m* auf null gesetzt. Das Gleiche wird für Lockobjekte ohne ausgehende Kanten in den Zeilen 59 bis 69 gemacht. Zuletzt werden in den Zeilen 71 bis 75 alle Lockobjekte, welche in keiner der bisherigen Mengen vorhanden sind, in das *Cyclic-set* eingefügt.

Der Algorithmus zur Reduzierung der *LockDependencyRelation* ist in Quellcode 4.3.6 dargestellt. Als erstes wird die Klassifizierung der Lockobjekte in Zeile 91 durchgeführt. Anschließend wird in den Zeilen 93 bis 103 das *Cyclic-set* durchlaufen und alle Lockobjekte mit einem Mode ungleich -1 aus diesem entfernt. Diese Lockobjekte können nicht Teil eines potenziellen Deadlocks sein, da sie nur von einem einzigen Thread verwendet werden. In der Zeile 105 wird eine neue *LockDependencyRelation* erzeugt, welche nur Lockobjekte aus dem *Cyclic-set* enthält. Wenn sich diese Relation nicht von der ursprünglichen unterscheidet ist die Lock Reduzierung abgeschlossen. Ansonsten wird in Zeile 107 die Lock Reduzierung rekursiv mit der eben erzeugten *LockDependencyRelation* erneut aufgerufen.

```

35 def lock_Classification(D, initClassification):
36     lockClassification = magiclockTypes.LockClassification()
37     s = []
38     for m in D.locks:
39         if initClassification.indegree[m] == 0 and initClassification.outdegree[m] == 0:
40             lockClassification.independentSet.append(m)
41         else:
42             if initClassification.indegree[m] == 0 or initClassification.outdegree[m] == 0:
43                 lockClassification.intermediateSet.append(m)
44                 s.append(m)
45
46     while s:
47         m = s.pop()
48         if initClassification.indegree[m] == 0:
49             for n in D.locks:
50                 if n == m:
51                     continue
52                 if initClassification.indegree[n] != 0:
53                     initClassification.indegree[n] -= initClassification.edgesFromTo[m][n]
54                     if initClassification.indegree[n] == 0:
55                         s.append(n)
56                     lockClassification.innerSet.append(n)
57                 initClassification.outdegree[m] -= initClassification.edgesFromTo[m][n]
58                 initClassification.edgesFromTo[m][n] = 0
59     if initClassification.outdegree[m] == 0:
60         for n in D.locks:
61             if n == m:
62                 continue
63             if initClassification.outdegree[n] != 0:
64                 initClassification.outdegree[n] -= initClassification.edgesFromTo[n][m]
65                 if initClassification.outdegree[n] == 0:
66                     s.append(n)
67                     lockClassification.innerSet.append(n)
68                 initClassification.indegree[m] -= initClassification.edgesFromTo[n][m]
69                 initClassification.edgesFromTo[n][m] = 0
70
71     for m in D.locks:
72         if (m not in lockClassification.independentSet and
73             m not in lockClassification.intermediateSet and
74             m not in lockClassification.innerSet):
75             lockClassification.cyclicSet.append(m)
76
77     return lockClassification

```

Quellcode 4.3.5: magiclockLib/lockReduction.py: Implementierung des *LockClassification(D)* Algorithmus aus Magiclock[2, S. 5]

Nach der Reduzierung der Lockobjekte werden die vorhandenen Lockobjekte aus dem *Cyclic-set* in disjunkte Mengen aufgeteilt. Die disjunkten Mengen haben untereinander keine Kanten und sind daher unabhängige Teilgraphen. Die Suche nach Zyklen erfolgt anschließend für jeden dieser Teilgraphen. In Quellcode 4.3.7 wird die Aufteilung in disjunkte Mengen durchgeführt. In den Zeilen 19 bis 23 werden die einzelnen Lockobjekte aus dem *Cyclic-set* durchlaufen und falls noch nicht betrachtet in der Methode *visit_Edges_From* betrachtet. In der Methode *visit_Edges_From* wird das Lockobjekt, falls noch nicht betrachtet, in die disjunkte Menge übernommen. Zusätzlich werden in den Zeilen 6 bis 8 alle erreichbaren Lockobjekte durchlaufen und ebenfalls in die disjunkte Menge aufgenommen. Im letzten Schritt wird für jede disjunkte Menge eine Zyklensuche durchgeführt, wobei jeder Zyklus einen potenziellen Deadlock repräsentiert. Dazu werden zuerst in Quellcode 4.3.8 in den Zeilen 94 bis 102 Partitionen erstellt. Es wird für jeden Thread *t* eine Partition erstellt, welche die *LockDependency* Objekte enthält, bei denen der ausführende Thread gleich *t* ist. Zusätzlich werden identische *LockDependency* Objekte gruppiert. Ein *LockDependency*

```

90 def lock_Reduction(D, initClassification):
91     lockClassification = lock_Classification(D, initClassification)
92     lockClassification.print()
93     for m in lockClassification.cyclicSet[:]:
94         if initClassification.mode[m] != -1:
95             lockClassification.cyclicSet.remove(m)
96             for n in lockClassification.cyclicSet:
97                 if initClassification.edgesFromTo[m][n] != 0:
98                     initClassification.indegree[n] -= initClassification.edgesFromTo[m][n]
99                     initClassification.edgesFromTo[m][n] = 0
100             for n in lockClassification.cyclicSet:
101                 if initClassification.edgesFromTo[n][m] != 0:
102                     initClassification.outdegree[n] -= initClassification.edgesFromTo[n][m]
103                     initClassification.edgesFromTo[n][m] = 0
104
105     projectedD = get_LockDependencyRelation_For(D, lockClassification.cyclicSet)
106     if projectedD.lockDependencies != D.lockDependencies:
107         return lock_Reduction(projectedD, initClassification)
108
109     return lockClassification, projectedD

```

Quellcode 4.3.6: magiclockLib/lockReduction.py: Implementierung des *LockReduction(D)* Algorithmus aus Magiclock[2, S. 5]

Objekt ist identisch mit einem anderen *LockDependency* Objekt, wenn der ausführende Thread, das betroffene Lockobjekt und die Menge der aktuell in Besitz befindlichen Lockobjekte übereinstimmt. Anschließend wird in den Zeilen 105 bis 108 für jeden Thread die jeweilige Partition durchlaufen. Jede *LockDependency* in der Partition wird mit der Methode *DFS_Traverse* in Quellcode 4.3.9 nach Zyklen durchsucht.

Zuerst wird in Zeile 61 eine Menge *s* erzeugt, die anfangs nur die aktuelle *LockDependency-Relation* enthält. In Zeile 62 werden alle höher sortierten Threads durchlaufen, wobei bereits betrachtete Threads in den Zeilen 63 und 64 ignoriert werden. Anschließend werden in den Zeilen 65 bis 75 die *LockDependency* Objekte in der Partition des höheren Threads durchlaufen. Für jedes dieser Objekte wird in den Zeilen 68 und 69 geprüft, ob die aktuelle Menge *s* zusammen mit dieser *LockDependency* eine Zyklische-Lock-Dependency-Chain⁸ bildet. Ist dies nicht der Fall, wird die Methode *DFS_Traverse* rekursiv für die aktuell in der Iteration befindliche *LockDependency* aufgerufen. Falls es sich um eine Zyklische-Lock-Dependency-Chain handelt wird für den Zyklus und jeden äquivalenten Zyklus ein potenzieller Deadlock in den Zeilen 50 bis 57 gemeldet.

⁸ Siehe Abschnitt 2.4

```

1  def visit_Edges_From(cyclicSet, edgesFromTo, visited, m, dc):
2      if visited[m] == False:
3          if m not in dc:
4              dc.append(m)
5              visited[m] = True
6              for n in cyclicSet:
7                  if edgesFromTo[m][n] != 0:
8                      visit_Edges_From(cyclicSet, edgesFromTo, visited, n, dc)
9
10
11 def disjoint_Components_Finder(cyclicSet, edgesFromTo):
12     dcs = set()
13     dc = []
14     visited = {}
15
16     for m in cyclicSet:
17         visited[m] = False
18
19     for m in cyclicSet:
20         if visited[m] == False:
21             visit_Edges_From(cyclicSet, edgesFromTo, visited, m, dc)
22             dcs.add(tuple(dc))
23             dc = []
24
25     return dcs

```

Quellcode 4.3.7: magiclockLib/cycleDetection.py: Implementierung des *DisjointComponentsFinder(Cyclic-set)* Algorithmus aus Magiclock[2, S. 8]

```

78 def find_Equal_Dependency_Group(Group, D, d):
79     for di in D:
80         if di == d:
81             return Group[di]
82     return []
83
84
85 def cycle_detection(potentialDeadlocks, dc, D):
86     Group = {}
87     isTraversed = {}
88     Di = {}
89
90     for t in D.threads:
91         isTraversed[t] = False
92         Di[t] = []
93
94     for d in D.lockDependencies:
95         if d.lockObjectName in dc and d.currentlyOwnedLockObjectNames:
96             g = find_Equal_Dependency_Group(Group, Di[d.threadName], d)
97             if g:
98                 g.add(d)
99             else:
100                 Di[d.threadName].append(d)
101                 Group[d] = []
102                 Group[d].append(d)
103
104     s = []
105     for t in D.threads:
106         for d in Di[t]:
107             isTraversed[t] = True
108             DFS_Traverse(potentialDeadlocks, t, s, d, D.threads, isTraversed, Di, Group)

```

Quellcode 4.3.8: magiclockLib/cycleDetection.py: Implementierung des *CycleDetection(dc, D)* Algorithmus aus Magiclock[2, S. 8]

```

50 def reportCycle(potentialDeadlocks, o, size, equCycle, Group):
51     if size == len(o):
52         potentialDeadlocks.append(equCycle.copy())
53     else:
54         for d in Group[o[size]]:
55             equCycle.append(d)
56             reportCycle(potentialDeadlocks, o, size + 1, equCycle, Group)
57             equCycle.remove(d)
58
59
60 def DFS_Traverse(potentialDeadlocks, i, s, d, k, isTraversed, Di, Group):
61     s.append(d)
62     for j in k[k.index(i) + 1:]:
63         if isTraversed[j] == True:
64             continue
65         for di in Di[j]:
66             o = s.copy()
67             o.append(di)
68             if is_Lock_Dependency_Chain(o):
69                 if lock_Dependency_Chain_Is_Cyclic_Lock_Dependency_Chain(o):
70                     equCycle = []
71                     reportCycle(potentialDeadlocks, o, 0, equCycle, Group)
72             else:
73                 isTraversed[j] = True
74                 DFS_Traverse(potentialDeadlocks, i, s, di, k, isTraversed, Di, Group)
75                 isTraversed[j] = False

```

Quellcode 4.3.9: magiclockLib/cycleDetection.py: Implementierung des *DFS_Traverse*(*i*, *S*, τ) Algorithmus aus Magiclock[2, S. 8]

5 Validierung

5.1	Trace Funktion	31
5.2	Analyse Programm	35
5.3	Visualisierung von potenziellen Deadlocks	36

5.1 Trace Funktion

Um die Trace-Funktionalität in OpenPEARL zu prüfen werden folgende Umgebungsvariablen verwendet:

- *OpenPEARL_LockTracer_Enabled* = false
- *OpenPEARL_LockTracer_Path* = /tmp/LockTracer/
- *OpenPEARL_LockTracer_MaxEntries* = 1

Zum Testen wird die OpenPEARL Anwendung aus Quellcode 5.1.1 verwendet. Es wird ein

```

1  MODULE(test);
2
3  SYSTEM;
4      stdout: StdOut;
5
6  PROBLEM;
7      SPC stdout DATION OUT SYSTEM ALPHIC GLOBAL;
8      DCL termout DATION OUT ALPHIC DIM(*,80) FORWARD STREAM CREATED(stdout);
9      DCL test_sema SEMA;
10
11  T1: TASK PRIORITY 1 MAIN;
12      FOR i TO 9
13          REPEAT
14              RELEASE test_sema;
15              REQUEST test_sema;
16          END;
17      END;
18
19  MODEND;

```

Quellcode 5.1.1: OpenPEARL Anwendung zum Testen der Trace-Funktionalität

Thread T1 erzeugt, welcher die *SEMA* Variable `test_sema` insgesamt neun Mal freigibt und in Besitz nimmt. Wird die Anwendung ausgeführt wird keine Trace-Datei angelegt. Wird die Umgebungsvariable *OpenPEARL_LockTracer_Enabled* auf `true` gesetzt, wird die Trace-Datei aus Quellcode 5.1.2 erzeugt.

Die Tests zur Messung der Laufzeit und der Speicherauslastung werden in einer virtuellen Maschine mit Debian 9 Betriebssystem, 4 CPU Kernen und 2 GB Arbeitsspeicher durchgeführt. Das Host System läuft mit dem Betriebssystem macOS 10.15.4 und verfügt über einen Intel Core i7 mit 3,1 GHz, 16 GB Arbeitsspeicher und einer 512 GB PCIe SSD.

1	1585474510537398:u(_T1,_test_sema)	10	1585474510538073:l(_T1,_test_sema)
2	1585474510537584:l(_T1,_test_sema)	11	1585474510538113:u(_T1,_test_sema)
3	1585474510537659:u(_T1,_test_sema)	12	1585474510538207:l(_T1,_test_sema)
4	1585474510537737:l(_T1,_test_sema)	13	1585474510538252:u(_T1,_test_sema)
5	1585474510537778:u(_T1,_test_sema)	14	1585474510538324:l(_T1,_test_sema)
6	1585474510537849:l(_T1,_test_sema)	15	1585474510538365:u(_T1,_test_sema)
7	1585474510537890:u(_T1,_test_sema)	16	1585474510538530:l(_T1,_test_sema)
8	1585474510537961:l(_T1,_test_sema)	17	1585474510538574:u(_T1,_test_sema)
9	1585474510538002:u(_T1,_test_sema)	18	1585474510538646:l(_T1,_test_sema)

Quellcode 5.1.2: Trace-Datei die bei aktivierter Trace-Funktionalität aus Quellcode 5.1.1 erzeugt wird

Zur Messung der Laufzeit wird das Pythonskript aus Quellcode 5.1.3 und zur Messung der Speicherauslastung das Pythonskript aus Quellcode 5.1.4 verwendet. Für die Tests

```

1  import sys
2  import time
3  import subprocess
4
5  times = 0
6  for x in range(1, 4):
7      timeStarted = time.time()
8      process = subprocess.check_call(['prl', '-r', sys.argv[1]])
9      timeEnd = time.time()
10
11     times += timeEnd - timeStarted
12  print("Finished process in " + str(times / 3) + " seconds.")

```

Quellcode 5.1.3: Pythonskript zur Messung der Laufzeit

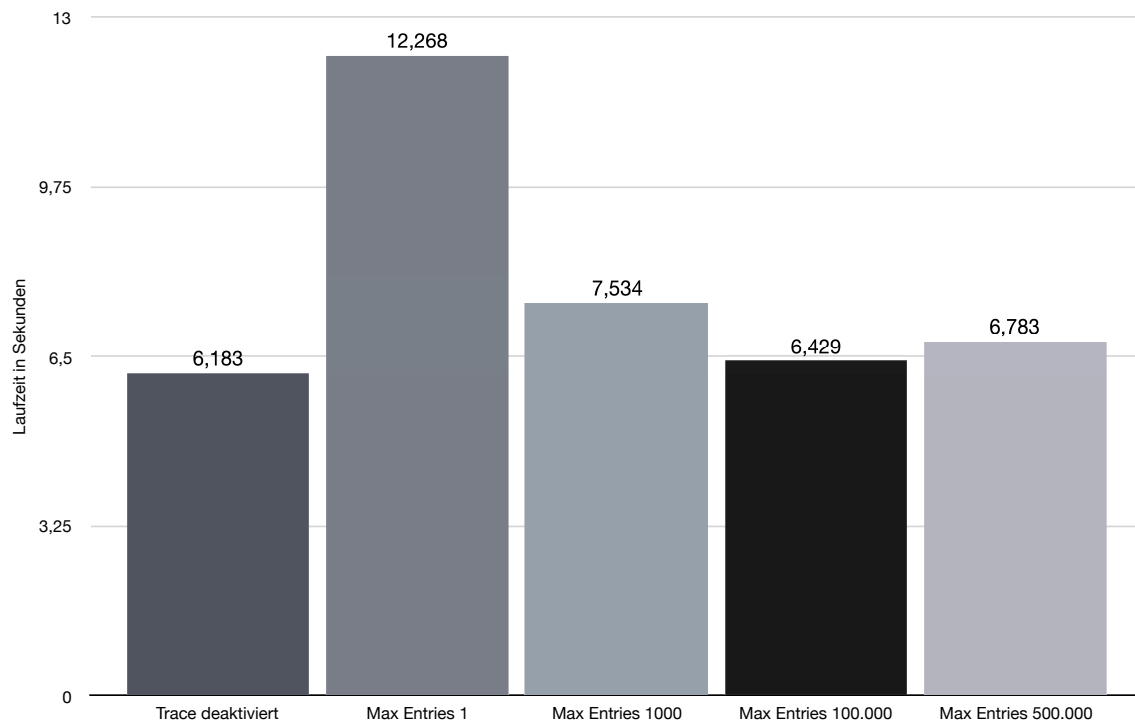
wird eine OpenPEARL Anwendung verwendet, welche zehn Threads erzeugt, die jeweils nacheinander zwei *SEMA* Objekte in Besitz nehmen und wieder freigeben. Dabei verwendet der Thread *T1* die Objekte *L01* und *L02*, der Thread *T2* die Objekte *L02* und *L03*, bis zum letzten Thread *T10*, welcher die Objekte *L10* und *L01* verwendet. Insgesamt werden 400.000 Einträge erzeugt. Das Ergebnis der Laufzeitmessung ist in Abb. 5.1.1 dargestellt. Eine höhere Anzahl an gepufferten Objekten führt bis zu einem bestimmten Punkt zu einer besseren Laufzeit. Bei einer Puffergröße von 100.000 gibt es nur noch einen sehr geringen Unterschied zwischen aktivierter und deaktivierter Trace-Funktionalität. Wird die Puffergröße auf 500.000 gesetzt, passen alle Einträge in den Puffer und werden beim Beenden der Anwendung in die Trace-Datei geschrieben. Die Laufzeit verbessert sich bei dieser Größe jedoch nicht weiter, sondern verschlechtert sich leicht. Bereits der Unterschied zwischen den Puffergrößen 1.000 und 100.000 zeigt, dass die Puffergröße nicht beliebig hoch gesetzt werden sollte, um eine optimale Laufzeit zu erhalten. Die Puffergröße muss individuell für das Zielsystem ermittelt werden. Das Ergebnis der Messung der Speicherauslastung ist in Abb. 5.1.2 dargestellt. Je größer der verwendete Puffer ist, desto höher ist auch die Speicherauslastung. Für das verwendete Testsystem ist eine Puffergröße von 1.000 ein guter Kompromiss zwischen Laufzeit und Speicherauslastung. Ist die Laufzeit wichtiger und die Speicherauslastung kein Problem, sollte eine Puffergröße von 100.000 gewählt werden.

```

1  import sys
2  import psutil
3  import time
4  import subprocess
5  import os
6
7  maxMemoryUsed = 0
8  for x in range(1, 4):
9      process = subprocess.Popen(['prl', '-r', sys.argv[1]])
10
11     maxMemory = 0
12     while process.poll() == None:
13         parent = psutil.Process(os.getpid())
14         memory = parent.memory_info().rss / 1024 / 1024
15
16         for child in parent.children(recursive=True):
17             memory += child.memory_info().rss / 1024 / 1024
18
19         if memory > maxMemory:
20             maxMemory = memory
21         time.sleep(.01)
22
23     process.wait()
24     maxMemoryUsed += maxMemory
25     print("Process used " + str(maxMemoryUsed / 3) + " MB.")

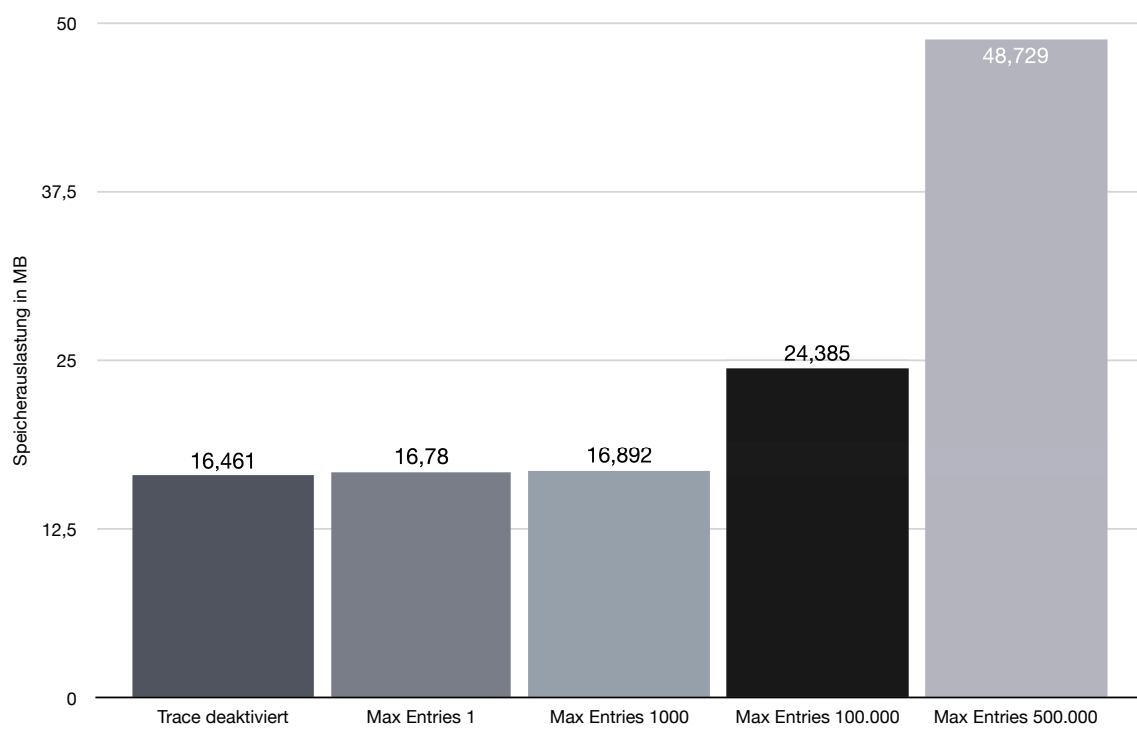
```

Quellcode 5.1.4: Pythonskript zur Messung der Speicherauslastung



Quelle: Eigene Darstellung

Abbildung 5.1.1: Ergebnisse der Laufzeitmessung der Trace-Funktionalität in OpenPEARL



Quelle: Eigene Darstellung

Abbildung 5.1.2: Ergebnisse der Speicherauslastung der Trace-Funktionalität in OpenPEARL

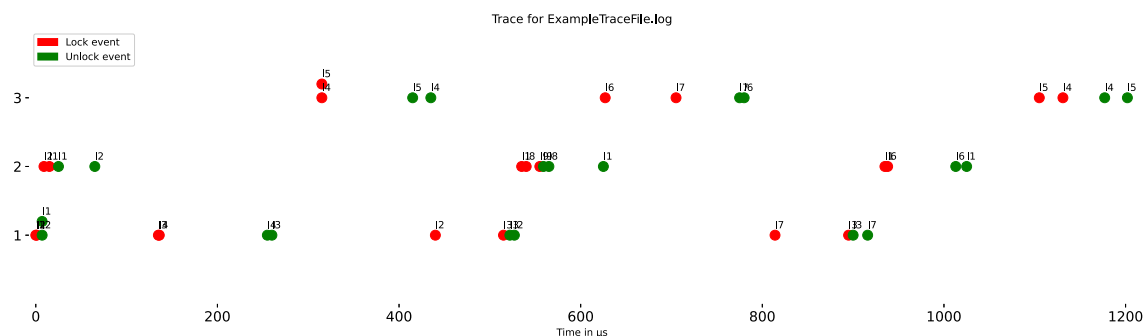
5.2 Analyse Programm

Für die chronologische Darstellung der Lockobjekte wird die Trace-Datei aus Quellcode 5.2.1 mit drei Threads und neun Lockobjekten verwendet. In dem Beispiel gibt es zusätzlich

1	44450944785:l(1,11)	15	44450945200:u(3,15)	29	44450945560:u(3,17)
2	44450944786:l(1,12)	16	44450945220:u(3,14)	30	44450945565:u(3,16)
3	44450944792:u(1,12)	17	44450945225:l(1,12)	31	44450945599:l(1,17)
4	44450944792:u(1,11)	18	44450945300:l(1,13)	32	44450945680:l(1,13)
5	44450944794:l(2,12)	19	44450945307:u(1,13)	33	44450945685:u(1,13)
6	44450944800:l(2,11)	20	44450945312:u(1,12)	34	44450945701:u(1,17)
7	44450944810:u(2,11)	21	44450945320:l(2,11)	35	44450945720:l(2,11)
8	44450944850:u(2,12)	22	44450945325:l(2,18)	36	44450945723:l(2,16)
9	44450944920:l(1,13)	23	44450945340:l(2,19)	37	44450945798:u(2,16)
10	44450944921:l(1,14)	24	44450945344:u(2,19)	38	44450945810:u(2,11)
11	44450945040:u(1,14)	25	44450945350:u(2,18)	39	44450945890:l(3,15)
12	44450945045:u(1,13)	26	44450945410:u(2,11)	40	44450945916:l(3,14)
13	44450945100:l(3,14)	27	44450945412:l(3,16)	41	44450945962:u(3,14)
14	44450945100:l(3,15)	28	44450945490:l(3,17)	42	44450945987:u(3,15)

Quellcode 5.2.1: Beispielhafte Trace-Datei mit einem potenziellen Deadlock

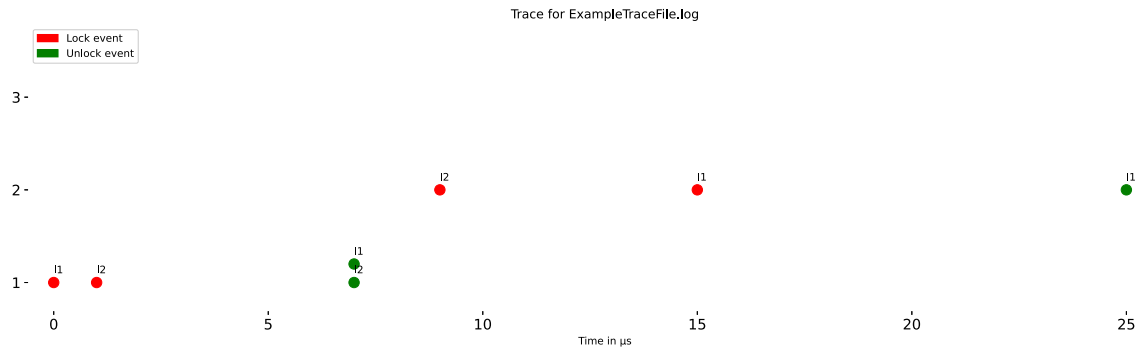
zwei Einträge mit dem gleichen Zeitstempel in den Zeilen 3 und 4 sowie in den Zeilen 13 und 14. Die Ausgabe der Anwendung aus Abschnitt 4.2 ist in Abb. 5.2.1 dargestellt.



Quelle: Eigene Darstellung

Abbildung 5.2.1: Ausgabe der Analyse-Anwendung

Die überlappenden Logeinträge können auseinander gezogen werden in dem in den Graphen hineingezoomt wird. Die Vergrößerung auf 70 Mikrosekunden ist in Abb. 5.2.2 dargestellt. Die einzelnen Logeinträge sind sichtbar und können auseinander gehalten werden. Eine Überlappung wird trotz gleichen Zeitstempel verhindert, indem die Logeinträge mit gleichen Zeitstempel bei sieben Mikrosekunden vertikal versetzt dargestellt werden.



Quelle: Eigene Darstellung

Abbildung 5.2.2: Vergrößerte Darstellung von Abb. 5.2.1

5.3 Visualisierung von potenziellen Deadlocks

Für die Visualisierung von potenziellen Deadlocks wird erneut die Trace-Datei aus Quellcode 5.2.1 verwendet. In dem Beispiel gibt es genau einen potenziellen Deadlock zwischen den Threads 1 und 2. In den Zeilen 1 und 2 nimmt der Thread 1 die Lockobjekte *l1* und *l2* nacheinander in Besitz. In den Zeilen 5 und 6 nimmt der Thread 2 die Lockobjekte *l2* und *l1* nacheinander in Besitz. Der potenzielle Deadlock entsteht, da der Thread 1 zuerst das Lockobjekt *l1* in Besitz nehmen kann und bevor dieser das Lockobjekt *l2* in Besitz nehmen kann, kann der Thread 2 das Lockobjekt *l2* bereits in seinen Besitz genommen haben. Dadurch blockieren sich beide Threads gegenseitig und ein Deadlock entsteht. Die Ausgabe der Anwendung zur Erkennung und Visualisierung von potenziellen Deadlocks ist in Abb. 5.3.1 dargestellt. Die Beschriftung der Kanten erfolgt immer zum Ende der

Potential Deadlocks found in: ExampleTraceFile.log



Quelle: Eigene Darstellung

Abbildung 5.3.1: Ergebnis der Erkennung von potenziellen Deadlocks aus Quellcode 5.2.1

Kante hin, zum Beispiel gehört die Beschriftung 1 zu der Kante von *l1* zu *l2*. Zusätzlich zur grafischen Darstellung werden die Ergebnisse als Zyklische-Lock-Dependency-Chain auf der Konsole ausgegeben. Für die verwendete Trace-Datei wird „(1,l2,l1) (2,l1,l2)“ als potenzieller Deadlock auf der Konsole ausgegeben.

6 Ausblick

6.1 Offene Punkte	37
6.2 Weiterentwicklung	37

6.1 Offene Punkte

In OpenPEARL gibt es neben den *SEMA* Variablen noch *BOLT* Variablen, die von der aktuellen Implementierung nicht beachtet werden. Um alle Synchronisierungsprobleme in OpenPEARL Anwendungen aufzeigen zu können, muss die Unterstützung für *BOLT* Variablen hinzugefügt werden.

Die derzeitige Integration in die OpenPEARL Laufzeitumgebung erfüllt noch nicht alle Kriterien in Bezug auf Softwarequalität, gerade im Bereich Robustheit. Aus diesem Grund ist die Integration noch kein Bestandteil des offiziellen OpenPEARL Repository.

6.2 Weiterentwicklung

Der nächste Entwicklungsschritt ist die Erkennung von potenziellen Deadlocks in Echtzeit während der Ausführung einer OpenPEARL Anwendung. Dazu muss die *LockTracer*-Implementierung aus Quellcode 4.1.2 erweitert werden. Anstatt die Ereignisse in eine Trace-Datei zu schreiben, müssen diese direkt über Interprozesskommunikation von der OpenPEARL Laufzeitumgebung an einen zweiten Prozess übertragen werden. Dieser kann dann die Python-Implementierungen aus Abschnitt 4.2 und Abschnitt 4.3 aufrufen und die Analysen direkt durchführen. Die Visualisierung muss dann in regelmäßigen Intervallen aktualisiert werden, um dem Benutzer die Änderungen anzuzeigen.

Literatur

- [1] Saddek Bensalem und Klaus Havelund. „Dynamic deadlock analysis of multi-threaded programs“. In: *Haifa Verification Conference*. Springer. 2005, S. 208–223 (siehe S. 9, 10).
- [2] Yan Cai und W. K. Chan. „Magiclock: Scalable Detection of Potential Deadlocks in Large-Scale Multithreaded Programs“. In: *IEEE Transactions on Software Engineering (TSE)* (2014) (siehe S. V, 14, 15, 25–30).
- [3] cameron314. *A fast multi-producer, multi-consumer lock-free concurrent queue for C++11*. 2020. URL: <https://github.com/cameron314/concurrentqueue> (besucht am 15. März 2020) (siehe S. 18).
- [4] Edward G Coffman, Melanie Elphick und Arie Shoshani. „System deadlocks“. In: *ACM Computing Surveys (CSUR)* 3.2 (1971), S. 67–78 (siehe S. 9, 10).
- [5] IEP Ingenieurbüro für Echtzeitprogrammierung GmbH. *Die Echtzeit Programmiersprache PEARL*. 2014. URL: <http://www.pearl90.de/pearlein.htm> (besucht am 12. Dez. 2019) (siehe S. 10).
- [6] PEARL’ GI-Fachgruppe 4.4.2 ’Echtzeitprogrammierung. *PEARL 90 Sprachreport*. 1. Jan. 1995. URL: <https://www.real-time.de/misc/PEARL90-Sprachreport-V2.0-GI-1995-de.pdf> (besucht am 13. Nov. 2019) (siehe S. 10, 11, 13).
- [7] *Informationstechnik - Programmiersprache PEARL - PEARL 90*. Norm. Apr. 1998 (siehe S. 10).
- [8] *Informationsverarbeitung - Programmiersprache PEARL - SafePEARL*. Norm. März 2018 (siehe S. 10).
- [9] Rainer Müller. *Differences between PEARL90 and OpenPEARL*. 2020. URL: <https://sourceforge.net/p/openpearl/wiki/Differences%20between%20OpenPEARL%20and%20PEARL90/> (besucht am 7. Apr. 2020) (siehe S. 13).
- [10] Robert HB Netzer und Barton P Miller. „What are race conditions? Some issues and formalizations“. In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.1 (1992), S. 74–88 (siehe S. 8).
- [11] Marcel Schaible u. a. *Structure of the OpenPEARL System*. 2020. URL: <https://sourceforge.net/p/openpearl/wiki/Home/#structure-of-the-openpearl-system> (besucht am 7. Apr. 2020) (siehe S. 13).

A OpenPEARL

```

/*
[The "BSD license"]
Copyright (c) 2012-2013 Rainer Mueller
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.
3. The name of the author may not be used to endorse or promote products
derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

/**
\file

\brief semaphore implementation for posix threads using simultaneous
locking pattern

*/

#define __STDC_LIMIT_MACROS // enable UINT32_MAX-macro
// must be set before stdint.h
#include <stdint.h>
#include "TaskCommon.h"
#include "Semaphore.h"
#include "Log.h"
#include "lockTracer/LockTracer.h"
#include "lockTracer/LockTraceEntry.h"

namespace pearlrt {

    PriorityQueue Semaphore::waiters;

    Semaphore::Semaphore(uint32_t preset, const char * n) {
        value = preset;
        name = n;
        Log::debug("Sema %s created with preset %u", n, (int)preset);
    }

    const char * Semaphore::getName(void) {
        return name;
    }

    uint32_t Semaphore::getValue(void) {
        return value;
    }
}

```

```

void Semaphore::decrement(void) {
    value --;
}

void Semaphore::increment(void) {
    if (value == UINT32_MAX) {
        throw theSemaOverflowSignal;
    }

    value ++;
}

int Semaphore::check(BlockData::BlockReasons::BlockSema *bd) {
    int wouldBlock = 0;
    int i;

    for (i = 0; i < bd->nsemas; i++) {
        if (bd->semas[i]->getValue() == 0) {
            wouldBlock = 1;
        }

        Log::debug("    check::sema: %s is %d",
            bd->semas[i]->getName(), (int)bd->semas[i]->getValue());
    }

    return wouldBlock;
}

void TraceLock(char* tName, const char* semaName) {
    LockTracer& lockTracer = LockTracer::GetInstance();
    if(lockTracer.IsEnabled() == false) {
        return;
    }

    LockTraceEntry entry = LockTraceEntry(std::chrono::high_resolution_clock::now(),
        ↳ LockTraceEntryType::Lock, tName, semaName);
    lockTracer.Add(entry);
}

void TraceUnlock(char* tName, const char* semaName) {
    LockTracer& lockTracer = LockTracer::GetInstance();
    if(lockTracer.IsEnabled() == false) {
        return;
    }

    LockTraceEntry entry = LockTraceEntry(std::chrono::high_resolution_clock::now(),
        ↳ LockTraceEntryType::Unlock, tName, semaName);
    lockTracer.Add(entry);
}

void Semaphore::request(TaskCommon* me,
    int nbrOfSemas,
    Semaphore** semas) {
    int i;
    int wouldBlock = 0;
    BlockData bd;

    bd.reason = REQUEST;
    bd.u.sema.nsemas = nbrOfSemas;
    bd.u.sema.semas = semas;

    TaskCommon::mutexLock();
    Log::info("request from task %s for %d semaphores", me->getName(),
        nbrOfSemas);

    wouldBlock = check(&(bd.u.sema));

    if (! wouldBlock) {
        for (i = 0; i < nbrOfSemas; i++) {
            semas[i]->decrement();
            if (semas[i]->getValue() == 0) {
                TraceLock(me->getName(), semas[i]->getName());
            }
        }
    }
}

```



```

        // critical region end
        TaskCommon::mutexUnlock();
    } else {
        Log::info("    task: %s going to blocked", me->getName());
        waiters.insert(me);
        // critical region ends in block()
        me->block(&bd);
        me->scheduleCallback();
    }
}

void Semaphore::release(TaskCommon* me,
                       int nbrOfSemas,
                       Semaphore** semas) {
    BlockData bd;
    int i;
    int wouldBlock;

    // start critical region - end after doing all possible releases
    TaskCommon::mutexLock();
    Log::debug("release from task %s for %d semaphores", me->getName(),
              nbrOfSemas);

    try {
        for (i = 0; i < nbrOfSemas; i++) {
            semas[i]->increment();
            if (semas[i]->getValue() == 1) {
                TraceUnlock(me->getName(), semas[i]->getName());
            }
            Log::debug("    sema: %s is now %u",
                      semas[i]->getName(), (int)semas[i]->getValue());
        }
    } catch (SemaOverflowSignal x) {
        Log::error("SemaOverflowSignal for %s",
                  semas[i]->getName());
        TaskCommon::mutexUnlock();
        throw;
    }

    TaskCommon * t = waiters.getHead();

    while (t != 0) {
        t->getBlockingRequest(&bd);
        wouldBlock = check(&(bd.u.sema));

        if (!wouldBlock) {
            for (i = 0; i < bd.u.sema.nsemas; i++) {
                bd.u.sema.semas[i]->decrement();
                if (bd.u.sema.semas[i]->getValue() == 0) {
                    TraceLock(me->getName(), bd.u.sema.semas[i]->getName());
                }
            }

            waiters.remove(t);
            t->unblock();
            Log::info("    unblocking: %s", t->getName());
        } else {
            Log::debug("    task %s still blocked", t->getName());
        }

        t = waiters.getNext(t);
    }

    TaskCommon::mutexUnlock();
}

BitString<1> Semaphore::dotry(TaskCommon* me, int nbrOfSemas, Semaphore** semas) {
    int i;
    int wouldBlock = 0;
    BlockData bd;
    BitString<1> result(1); // true

    bd.reason = REQUEST;
    bd.u.sema.nsemas = nbrOfSemas;
    bd.u.sema.semas = semas;

```

```

    // start critical region
    TaskCommon::mutexLock();
    Log::debug("try from task %s for %d semaphores", me->getName(),
        nbrOfSemas);
    wouldBlock = check(&(bd.u.sema));

    if (! wouldBlock) {
        for (i = 0; i < nbrOfSemas; i++) {
            semas[i]->decrement();
            if (semas[i]->getValue() == 0) {
                TraceLock(me->getName(), semas[i]->getName());
            }
        }
    }

    TaskCommon::mutexUnlock();

    if (wouldBlock) {
        result.x = 0;    // false
    }

    //return !wouldBlock;
    return result;
}

void Semaphore::removeFromWaitQueue(TaskCommon * t) {
    waiters.remove(t);
}

void Semaphore::addToWaitQueue(TaskCommon * t) {
    BlockData bd;
    int wouldBlock;

    t->getBlockingRequest(&bd);
    wouldBlock = check(&(bd.u.sema));

    if (!wouldBlock) {
        for (int i = 0; i < bd.u.sema.nsemas; i++) {
            bd.u.sema.semas[i]->decrement();
            if (bd.u.sema.semas[i]->getValue() == 0) {
                TraceLock(t->getName(), bd.u.sema.semas[i]->getName());
            }
        }

        waiters.remove(t);
        t->unblock();
        Log::debug("  unblocking: %s", t->getName());
    } else {
        waiters.insert(t);
    }
}

void Semaphore::updateWaitQueue(TaskCommon * t) {
    if (waiters.remove(t)) {
        waiters.insert(t);
    }
}
}

```

Quellcode A.0.1: Angepasste Semaphore.cc Implementierung der OpenPEARL Laufzeitumgebung

```

/**
# [A "BSD license"]
# Copyright (c) 2012-2017 Rainer Mueller
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
# are met:

```

```

#
# 1. Redistributions of source code must retain the above copyright
#    notice, this list of conditions and the following disclaimer.
# 2. Redistributions in binary form must reproduce the above copyright
#    notice, this list of conditions and the following disclaimer in the
#    documentation aor other materials provided with the distribution.
# 3. The name of the author may not be used to endorse or promote products
#    derived from this software without specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
# IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
# OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
# IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
# INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
# NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
# DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
# THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
# (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
# THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

# pearl runtime system portable components

# list of files which are independend of the target platform
#
# give the path name relative to this directory. The target specific
# makefile will add a prefix with the relatove path corresponding to
# the target specific makefile

CXX_COMMON = \
    Array.cc \
    Bolt.cc \
    LogFile.cc \
    Log.cc \
    Clock.cc \
    Duration.cc \
    PutDuration.cc GetDuration.cc \
    Signals.cc \
    ScheduleSignalAction.cc \
    Fixed63.cc \
    Fixed.cc \
    Prio.cc \
    CharSlice.cc \
    Character.cc RefChar.cc \
    compare.cc \
    Interrupt.cc \
    Source.cc \
    RefCharSink.cc RefCharSource.cc \
    SystemDationNBSource.cc SystemDationNBSink.cc \
    PutClock.cc GetClock.cc \
    GetBitString.cc \
    PutBitString.cc \
    GetHelper.cc PutHelper.cc \
    TaskWhenLinks.cc \
    TaskCommon.cc \
    TaskList.cc \
    TaskMonitor.cc \
    TaskTimerCommon.cc \
    MutexCommon.cc CSemaCommon.cc \
    Semaphore.cc PriorityQueue.cc \
    Rst.cc \
    SystemDation.cc \
    SystemDationNB.cc \
    DationRW.cc \
    IOFormats.cc \
    IOJob.cc \
    DationPG.cc \
    StringDationConvert.cc \
    DationTS.cc \
    UserDationNB.cc UserDation.cc \
    TFUBuffer.cc \
    DationDim.cc DationDim1.cc \
    DationDim2.cc DationDim3.cc \
    FloatHelper.cc \
    SoftInt.cc \
    Control.cc \

```

```

    ConsoleCommon.cc \
    FullDuplexDationAbortNB.cc\
    SampleBasicDation.cc \
    lockTracer/LockTracer.cc lockTracer/LockTraceEntry.cc \
    lockTracer/LockTraceEntryFormatter.cc

#CONFIG_HAC_I2C must be set manually by the makefile of
# the platform, which is build now
ifeq ($(CONFIG_HAS_I2C),y)
    HDR_COMMON += I2CProvider.h
    CXX_COMMON += LM75.cc
    XML_COMMON += LM75.xml
    CXX_COMMON += ADS1015SE.cc
    XML_COMMON += ADS1015SE.xml
    CXX_COMMON += PCF8574Pool.cc PCF8574In.cc PCF8574Out.cc
    XML_COMMON += PCF8574In.xml PCF8574Out.xml
    CXX_COMMON += BME280.cc bosch/bme280.c
    XML_COMMON += BME280.xml
    CXX_COMMON += PCA9685.cc PCA9685Channel.cc
    XML_COMMON += PCA9685.xml PCA9685Channel.xml
endif

XML_COMMON += Log.xml LogFile.xml SampleBasicDation.xml SoftInt.xml

HDR_COMMON = $(CXX_COMMON:%.cc=%.h) \
    BitString.h \
    BitSlice.h \
    GetBitString.h \
    Sink.h RefCharSink.h \
    PutFixed.h GetFixed.h \
    Dation.h \
    Device.h UserDation.h \
    PutCharacter.h \
    Float.h \
    Ref.h

ifeq ($(CONFIG_CANSUPPORT),y)
    HDR_COMMON += Can2AMessage.h
endif

```

Quellcode A.0.2: Angepasste Files.common der OpenPEARL Laufzeitumgebung

```

MODULE(test);

SYSTEM;
    stdout: StdOut;

PROBLEM;
    SPC stdout DATION OUT SYSTEM ALPHIC GLOBAL;
    DCL termout DATION OUT ALPHIC DIM(*,80) FORWARD STREAM CREATED(stdout);
    DCL 101 SEMA;
    DCL 102 SEMA;
    DCL 103 SEMA;
    DCL 104 SEMA;
    DCL 105 SEMA;
    DCL 106 SEMA;
    DCL 107 SEMA;
    DCL 108 SEMA;
    DCL 109 SEMA;
    DCL 110 SEMA;

TO: TASK PRIORITY 1 MAIN;
    RELEASE 101;
    RELEASE 102;
    RELEASE 103;
    RELEASE 104;
    RELEASE 105;
    RELEASE 106;
    RELEASE 107;
    RELEASE 108;
    RELEASE 109;
    RELEASE 110;

```

```
END;

T1: TASK PRIORITY 1 MAIN;
  FOR i TO 10000
    REPEAT
      REQUEST 101;
      RELEASE 101;
      REQUEST 102;
      RELEASE 102;
    END;
  END;

T2: TASK PRIORITY 1 MAIN;
  FOR i TO 10000
    REPEAT
      REQUEST 102;
      RELEASE 102;
      REQUEST 103;
      RELEASE 103;
    END;
  END;

T3: TASK PRIORITY 1 MAIN;
  FOR i TO 10000
    REPEAT
      REQUEST 103;
      RELEASE 103;
      REQUEST 104;
      RELEASE 104;
    END;
  END;

T4: TASK PRIORITY 1 MAIN;
  FOR i TO 10000
    REPEAT
      REQUEST 104;
      RELEASE 104;
      REQUEST 105;
      RELEASE 105;
    END;
  END;

T5: TASK PRIORITY 1 MAIN;
  FOR i TO 10000
    REPEAT
      REQUEST 105;
      RELEASE 105;
      REQUEST 106;
      RELEASE 106;
    END;
  END;

T6: TASK PRIORITY 1 MAIN;
  FOR i TO 10000
    REPEAT
      REQUEST 106;
      RELEASE 106;
      REQUEST 107;
      RELEASE 107;
    END;
  END;

T7: TASK PRIORITY 1 MAIN;
  FOR i TO 10000
    REPEAT
      REQUEST 107;
      RELEASE 107;
      REQUEST 108;
      RELEASE 108;
    END;
  END;

T8: TASK PRIORITY 1 MAIN;
  FOR i TO 10000
    REPEAT
```

```
        REQUEST 108;
        RELEASE 108;
        REQUEST 109;
        RELEASE 109;
    END;
END;

T9: TASK PRIORITY 1 MAIN;
    FOR i TO 10000
        REPEAT
            REQUEST 109;
            RELEASE 109;
            REQUEST 110;
            RELEASE 110;
        END;
    END;
END;

T10: TASK PRIORITY 1 MAIN;
    FOR i TO 10000
        REPEAT
            REQUEST 110;
            RELEASE 110;
            REQUEST 101;
            RELEASE 101;
        END;
    END;
END;

MODEND;
```

Quellcode A.0.3: OpenPEARL Testanwendung für Performanztests

B C++

```
#pragma once

#include <chrono>
#include <string>
#include "LockTraceEntryType.h"

namespace pearlrt {
    class LockTraceEntry
    {
    private:
        std::chrono::time_point<std::chrono::high_resolution_clock> dateTime;
        LockTraceEntryType entryType;
        std::string threadName;
        std::string objectName;
    public:
        std::chrono::time_point<std::chrono::high_resolution_clock> get_DateTime();
        LockTraceEntryType get_EntryType();
        std::string get_ThreadName();
        std::string get_ObjectName();
        LockTraceEntry();
        LockTraceEntry(std::chrono::time_point<std::chrono::high_resolution_clock> dt,
            ↪ LockTraceEntryType et, std::string tn, std::string on);
    };
}
```

Quellcode B.0.1: Header-Datei der Repräsentation eines Logeintrags

```
#include "LockTraceEntry.h"

namespace pearlrt {

    LockTraceEntry::LockTraceEntry() {
    }

    LockTraceEntry::LockTraceEntry(std::chrono::time_point<std::chrono::high_resolution_clock>
    ↪ > dt, LockTraceEntryType et, std::string tn, std::string
    ↪ on){
        dateTime = dt;
        entryType = et;
        threadName = tn;
        objectName = on;
    }

    std::chrono::time_point<std::chrono::high_resolution_clock>
    ↪ LockTraceEntry::get_DateTime() {
        return dateTime;
    }

    LockTraceEntryType LockTraceEntry::get_EntryType() {
        return entryType;
    }

    std::string LockTraceEntry::get_ThreadName() {
        return threadName;
    }

    std::string LockTraceEntry::get_ObjectName() {
        return objectName;
    }
}
```

Quellcode B.0.2: Implementierung der Repräsentation eines Logeintrags

```
#pragma once

#include <string>
#include "LockTraceEntry.h"

namespace pearlrt {
    class LockTraceEntryFormatter {
    private:
        const std::string endMarker = "\r\n";
        const std::string emptyReturnValue = "";
        std::string createDateTimeEntry(std::chrono::time_point<std::chrono::high_resolution_clock>
        ↪   timePoint);
        LockTraceEntryFormatter();
    public:
        static LockTraceEntryFormatter& GetInstance();
        std::string FormatLockTraceEntry(LockTraceEntry& logTraceEntry);
        LockTraceEntryFormatter(const LockTraceEntryFormatter&) = delete;
        LockTraceEntryFormatter(LockTraceEntryFormatter&&) = delete;
        LockTraceEntryFormatter& operator=(const LockTraceEntryFormatter&) = delete;
        LockTraceEntryFormatter& operator=(LockTraceEntryFormatter&&) = delete;
    };
}
```

Quellcode B.0.3: Header-Datei des Formatieres für Logeinträge

```
#include "LockTraceEntryFormatter.h"

namespace pearlrt {

    using namespace std::chrono;
    using namespace std;

    LockTraceEntryFormatter::LockTraceEntryFormatter() {}

    std::string LockTraceEntryFormatter::FormatLockTraceEntry(LockTraceEntry& logTraceEntry) {

        std::string prefix = "";
        switch (logTraceEntry.get_EntryType())
        {
            case LockTraceEntryType::Lock:
                prefix = "l";
                break;
            case LockTraceEntryType::Unlock:
                prefix = "u";
                break;
            default:
                return emptyReturnValue;
        }

        return LockTraceEntryFormatter::createDateTimeEntry(logTraceEntry.get_DateTime())
        + prefix + "(" + logTraceEntry.get_ThreadName() + ", " +
        ↪   logTraceEntry.get_ObjectName() + ")"
        + endMarker;
    }

    std::string LockTraceEntryFormatter::createDateTimeEntry(std::chrono::time_point<std::chrono::high_resolution_clock> timePoint)
    ↪   {
        return std::to_string(std::chrono::time_point_cast<std::chrono::microseconds>(timePoint).time_since_epoch().count()) +
        ↪   ":";
    }

    LockTraceEntryFormatter& LockTraceEntryFormatter::GetInstance()
    {
        static LockTraceEntryFormatter instance;
        return instance;
    }
}
```


Quellcode B.0.4: Implementierung des Formatierers für Logeinträge

```
#pragma once

namespace pearlrt {
    enum class LockTraceEntryType {
        Lock,
        Unlock
    };
}
```

Quellcode B.0.5: Header-Datei der Enumeration für den Typ eines Logeintrags

```
#pragma once

#include <mutex>
#include "LockTraceEntry.h"
#include "LockTraceEntryFormatter.h"
#include "concurrentqueue.h"

namespace pearlrt {
    class LockTracer {
    private:
        const char* NameOfEnvironmentVariableEnabled = "OpenPEARL_LockTracer_Enabled";
        const char* NameOfEnvironmentVariablePath = "OpenPEARL_LockTracer_Path";
        const char* NameOfEnvironmentVariableNumberOfMaxEntries =
            "OpenPEARL_LockTracer_MaxEntries";
        const unsigned short DefaultNumberOfMaxEntries = 20;
        bool isEnabled;
        unsigned int numberOfMaxEntries = 20;
        std::string filePath;
        moodycamel::ConcurrentQueue<LockTraceEntry> queue;
        std::mutex flushMutex;

        LockTracer();
        bool directoryExists(const char *fileName);
        void setNumberOfMaxEntries();
        void flushIfNeeded();
        void flush();
    public:
        static LockTracer& GetInstance();
        void Add(LockTraceEntry& entry);
        bool IsEnabled();
        LockTracer(const LockTracer&) = delete;
        LockTracer(LockTracer&&) = delete;
        LockTracer& operator=(const LockTracer&) = delete;
        LockTracer& operator=(LockTracer&&) = delete;
        ~LockTracer();
    };
}
```

Quellcode B.0.6: Header-Datei des Log-Tracers

```
#include <iostream>
#include <fstream>
#include <ctime>
#include <sstream>
#include <string.h>
#include <iomanip>
#include "LockTracer.h"

namespace pearlrt {

    LockTracer::LockTracer() {
        isEnabled = false;

        char* envVar = std::getenv(NameOfEnvironmentVariableEnabled);
        if(envVar != NULL && strcmp(envVar, "true") == 0) {
            envVar = std::getenv(NameOfEnvironmentVariablePath);
        }
    }
}
```

```

        if(envVar != NULL && directoryExists(envVar)) {
            std::time_t t = std::time(nullptr);
            std::tm tm = *std::localtime(&t);

            std::ostringstream oss;
            oss << std::put_time(&tm, "%Y-%m-%d_%H-%M.log");
            std::string str = oss.str();

            filePath = std::string(envVar) + str;

            setNumberOfMaxEntries();
            isEnabled = true;
        }
    }

    LockTracer& LockTracer::GetInstance()
    {
        static LockTracer instance;
        return instance;
    }

    void LockTracer::Add(LockTraceEntry& entry) {
        if(isEnabled == false) {
            return;
        }

        queue.enqueue(entry);
        LockTracer::flushIfNeeded();
    }

    bool LockTracer::IsEnabled() {
        return isEnabled;
    }

    LockTracer::~LockTracer() {
        if(isEnabled == false) {
            return;
        }

        LockTracer::flush();
    }

    bool LockTracer::directoryExists(const char *fileName)
    {
        std::ifstream infile(fileName);
        return infile.good();
    }

    void LockTracer::setNumberOfMaxEntries() {
        char* envVar = std::getenv(NameOfEnvironmentVariableNumberOfMaxEntries);
        if(envVar != NULL) {
            try
            {
                numberOfMaxEntries = std::stoi(envVar);
            }
            catch(const std::exception& e)
            {
                numberOfMaxEntries = DefaultNumberOfMaxEntries;
            }
        }
        else {
            numberOfMaxEntries = DefaultNumberOfMaxEntries;
        }
    }

    void LockTracer::flushIfNeeded() {
        if(queue.size_approx() >= numberOfMaxEntries) {
            LockTracer::flush();
        }
    }

    void LockTracer::flush() {
        std::lock_guard<std::mutex> lock(flushMutex);
        try

```

```
{
    std::ofstream fileStream;
    fileStream.open(filePath, std::ios_base::out | std::ios_base::app);
    for (uint i = 0; i < numberOfMaxEntries; i++)
    {
        LockTraceEntry entry;
        if(queue.try_dequeue(entry)) {
            fileStream <<
                ↳ LockTraceEntryFormatter::GetInstance().FormatLockTraceEntry(entry);
            }
        }
        fileStream.close();
    }
    catch(const std::exception& e)
    {
    }
}
```

Quellcode B.0.7: Implementierung des Log-Tracers

C Python

```

class LockActionType:
    LOCK = 1
    UNLOCK = 2

class LockAction(object):
    def __init__(self, timeStamp, threadName, lockObjectName, actionType):
        self.timeStamp = timeStamp
        self.threadName = threadName
        self.lockObjectName = lockObjectName
        self.actionType = actionType

def lockAction_sort(lockAction):
    return lockAction.timeStamp

def read_Trace_File_Lines(traceFilename):
    file = open(traceFilename, 'r')
    lines = file.readlines()

    lockActions = []
    for line in lines:
        lineValues = line.strip().split(':')
        threadAndObjectName = lineValues[1][2:-1].split(',')
        if lineValues[1][0] == "l":
            lockActions.append(LockAction(lineValues[0], threadAndObjectName[0],
            ↪ threadAndObjectName[1], LockActionType.LOCK))
        elif lineValues[1][0] == "u":
            lockActions.append(LockAction(lineValues[0], threadAndObjectName[0],
            ↪ threadAndObjectName[1], LockActionType.UNLOCK))

    lockActions.sort(key=lockAction_sort)
    return lockActions

```

Quellcode C.0.1: traceFileReader.py: Implementierung des Trace File Readers

```

import sys
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import ntpath

import traceFileReader

def get_Color(value):
    switcher = {
        traceFileReader.LockActionType.LOCK: "red",
        traceFileReader.LockActionType.UNLOCK: "green"
    }
    return switcher.get(value, "")

def create_Graph(lockActions, title):
    threads = set([])
    times = set([])
    x = []
    y = []
    c = []
    texts = []

    for lockAction in lockActions:
        threads.add(lockAction.threadName)
        times.add(lockAction.timeStamp)

    threads = sorted(list(threads))
    times = sorted(list(times))

```

```

for lockAction in lockActions:
    xValue = int(lockAction.timeStamp) - int(times[0])
    offset = 0
    if xValue in x:
        offset = 0.2 * x.count(xValue)
    x.append(xValue)
    y.append(threads.index(lockAction.threadName) + offset)
    c.append(get_Color(lockAction.actionType))
    texts.append(lockAction.lockObjectName)

plt.figure(figsize=(20, len(threads) + 2))
plt.title(title)
plt.scatter(x, y, c=c, alpha=0.85, s=100)
for i, txt in enumerate(texts):
    plt.annotate(txt, (x[i], y[i] + 0.1))
plt.xlabel("Time in  $\mu$ s")

red_patch = mpatches.Patch(color='red', label='Lock event')
green_patch = mpatches.Patch(color='green', label='Unlock event')
plt.legend(handles=[red_patch, green_patch], loc="upper left")

ax = plt.subplot(111)

# Styling
ax.spines["top"].set_visible(False)
ax.spines["bottom"].set_visible(False)
ax.spines["right"].set_visible(False)
ax.spines["left"].set_visible(False)

ax.get_xaxis().tick_bottom()
ax.get_yaxis().tick_left()
# Styling End

plt.ylim((-1, len(threads)))
plt.xlim((-1, int(times[len(times) - 1]) - int(times[0]) + 1))

plt.yticks(range(len(threads)), threads, fontsize=14)
plt.xticks(fontsize=14)

return plt

traceFilename = sys.argv[1]
lockActions = traceFileReader.read_Trace_File_Lines(traceFilename)
plt = create_Graph(lockActions, "Trace for " + ntpath.basename(traceFilename))
plt.show()

```

Quellcode C.0.2: generateTimeline.py: Skript zur chronologischen Darstellung der Lock-Ereignisse

```

from collections import defaultdict

class InitClassification(object):
    def __init__(self):
        self.indegree = {}
        self.outdegree = {}
        self.mode = {}
        self.edgesFromTo = defaultdict(lambda: defaultdict(int))

    def print(self):
        print("\nInit Classification:")
        print("Indegrees:")
        print(self.indegree)
        print("Oudegrees:")
        print(self.outdegree)
        print("Modes:")
        print(self.mode)
        print("edgesFromTo:")
        for (f, t) in self.edgesFromTo.items():
            print(f, end=": ")
            for x in t:
                print(x, end=",")
            print("")

```

```

class LockClassification(object):
    def __init__(self):
        self.independentSet = []
        self.intermediateSet = []
        self.innerSet = []
        self.cyclicSet = []

    def print(self):
        print("\nLock Classification:")
        print("IndependentSet:")
        print(self.independentSet)
        print("IntermediateSet:")
        print(self.intermediateSet)
        print("InnerSet:")
        print(self.innerSet)
        print("CyclicSet:")
        print(self.cyclicSet)

class LockDependency(object):
    def __init__(self, threadName, lockObjectName, currentlyOwnedLockObjectNames):
        self.threadName = threadName
        self.lockObjectName = lockObjectName
        self.currentlyOwnedLockObjectNames = currentlyOwnedLockObjectNames

    def clone(self):
        lst = []
        for x in self.currentlyOwnedLockObjectNames:
            lst.append(x)
        return LockDependency(self.threadName, self.lockObjectName, lst)

    def print(self):
        print("(", end="")
        print(str(self.threadName) + ", " + str(self.lockObjectName) + ", {", end="")
        for x in self.currentlyOwnedLockObjectNames:
            print(str(x), end="")
            if x != self.currentlyOwnedLockObjectNames[-1]:
                print(", ", end="")
        print("}")", end="")

class LockDependencyRelation(object):
    def __init__(self):
        self.locks = set()
        self.threads = []
        self.lockDependencies = []

    def add(self, lockDependency):
        self.lockDependencies.append(lockDependency)
        self.locks.add(lockDependency.lockObjectName)
        if lockDependency.threadName not in self.threads:
            self.threads.append(lockDependency.threadName)

    def print(self):
        print("\nLockDependencyRelation:")
        for d in self.lockDependencies:
            d.print()
            if d != self.lockDependencies[-1]:
                print(", ", end="")
        print("\nLocks:")
        print(self.locks)

```

Quellcode C.0.3: magiclockLib/magicLockTypes.py: Sammlung von Klassen die von der Magiclock Implementierung verwendet werden

```

import magiclockLib.magiclockTypes as magiclockTypes

def mode(m, D):
    thread = None
    for d in D.lockDependencies:
        if d.lockObjectName != m:

```

```

        continue
    if thread == None:
        thread = d.threadName
    elif d.threadName == thread:
        continue
    else:
        return None
return thread

def init_Classification(D):
    initClassification = magiclockTypes.InitClassification()
    for m in D.locks:
        initClassification.indegree[m] = 0
        initClassification.outdegree[m] = 0
        initClassification.mode[m] = 0
    for d in D.lockDependencies:
        if mode(d.lockObjectName, D) != d.threadName:
            initClassification.mode[d.lockObjectName] = -1
        else:
            initClassification.mode[d.lockObjectName] = d.threadName
    for n in d.currentlyOwnedLockObjectNames:
        initClassification.indegree[d.lockObjectName] += 1
        initClassification.outdegree[n] += 1
        initClassification.edgesFromTo[n][d.lockObjectName] += 1
    return initClassification

def lock_Classification(D, initClassification):
    lockClassification = magiclockTypes.LockClassification()
    s = []
    for m in D.locks:
        if initClassification.indegree[m] == 0 and initClassification.outdegree[m] == 0:
            lockClassification.independentSet.append(m)
        else:
            if initClassification.indegree[m] == 0 or initClassification.outdegree[m] == 0:
                lockClassification.intermediateSet.append(m)
                s.append(m)

    while s:
        m = s.pop()
        if initClassification.indegree[m] == 0:
            for n in D.locks:
                if n == m:
                    continue
                if initClassification.indegree[n] != 0:
                    initClassification.indegree[n] -= initClassification.edgesFromTo[m][n]
                    if initClassification.indegree[n] == 0:
                        s.append(n)
                        lockClassification.innerSet.append(n)
                initClassification.outdegree[m] -= initClassification.edgesFromTo[m][n]
                initClassification.edgesFromTo[m][n] = 0
        if initClassification.outdegree[m] == 0:
            for n in D.locks:
                if n == m:
                    continue
                if initClassification.outdegree[n] != 0:
                    initClassification.outdegree[n] -= initClassification.edgesFromTo[n][m]
                    if initClassification.outdegree[n] == 0:
                        s.append(n)
                        lockClassification.innerSet.append(n)
                initClassification.indegree[m] -= initClassification.edgesFromTo[n][m]
                initClassification.edgesFromTo[n][m] = 0

    for m in D.locks:
        if (m not in lockClassification.independentSet and
            m not in lockClassification.intermediateSet and
            m not in lockClassification.innerSet):
            lockClassification.cyclicSet.append(m)

    return lockClassification

def get_LockDependencyRelation_For(D, cyclicSet):
    lockDependencyRelation = magiclockTypes.LockDependencyRelation()

```

```

    for d in D.lockDependencies:
        if d.lockObjectName in cyclicSet:
            lockDependencyRelation.add(d)

    return lockDependencyRelation

def lock_Reduction(D, initClassification):
    lockClassification = lock_Classification(D, initClassification)
    lockClassification.print()
    for m in lockClassification.cyclicSet[:]:
        if initClassification.mode[m] != -1:
            lockClassification.cyclicSet.remove(m)
            for n in lockClassification.cyclicSet:
                if initClassification.edgesFromTo[m][n] != 0:
                    initClassification.indegree[n] -= initClassification.edgesFromTo[m][n]
                    initClassification.edgesFromTo[m][n] = 0
            for n in lockClassification.cyclicSet:
                if initClassification.edgesFromTo[n][m] != 0:
                    initClassification.outdegree[n] -= initClassification.edgesFromTo[n][m]
                    initClassification.edgesFromTo[n][m] = 0

    projectedD = get_LockDependencyRelation_For(D, lockClassification.cyclicSet)
    if projectedD.lockDependencies != D.lockDependencies:
        return lock_Reduction(projectedD, initClassification)

    return lockClassification, projectedD

```

Quellcode C.0.4: magiclockLib/lockReduction.py: Implementierung des Magiclock-Algorithmus zur Reduzierung von Locks

```

def visit_Edges_From(cyclicSet, edgesFromTo, visited, m, dc):
    if visited[m] == False:
        if m not in dc:
            dc.append(m)
        visited[m] = True
        for n in cyclicSet:
            if edgesFromTo[m][n] != 0:
                visit_Edges_From(cyclicSet, edgesFromTo, visited, n, dc)

def disjoint_Components_Finder(cyclicSet, edgesFromTo):
    dcs = set()
    dc = []
    visited = {}

    for m in cyclicSet:
        visited[m] = False

    for m in cyclicSet:
        if visited[m] == False:
            visit_Edges_From(cyclicSet, edgesFromTo, visited, m, dc)
            dcs.add(tuple(dc))
            dc = []

    return dcs

def is_Lock_Dependency_Chain(d):
    if len(d) <= 1:
        return False

    for i in range(len(d) - 1):
        if d[i].lockObjectName not in d[i + 1].currentlyOwnedLockObjectNames:
            return False
        for j in range(len(d)):
            if d[i].threadName == d[j].threadName:
                continue
            if list(set(d[i].currentlyOwnedLockObjectNames) &
                    set(d[j].currentlyOwnedLockObjectNames)):
                return False

```



```

    return True

def lock_Dependency_Chain_Is_Cyclic_Lock_Dependency_Chain(d):
    if d[-1].lockObjectName in d[0].currentlyOwnedLockObjectNames:
        return True
    return False

def reportCycle(potentialDeadlocks, o, size, equCycle, Group):
    if size == len(o):
        potentialDeadlocks.append(equCycle.copy())
    else:
        for d in Group[o[size]]:
            equCycle.append(d)
            reportCycle(potentialDeadlocks, o, size + 1, equCycle, Group)
            equCycle.remove(d)

def DFS_Traverse(potentialDeadlocks, i, s, d, k, isTraversed, Di, Group):
    s.append(d)
    for j in k[k.index(i) + 1:]:
        if isTraversed[j] == True:
            continue
        for di in Di[j]:
            o = s.copy()
            o.append(di)
            if is_Lock_Dependency_Chain(o):
                if lock_Dependency_Chain_Is_Cyclic_Lock_Dependency_Chain(o):
                    equCycle = []
                    reportCycle(potentialDeadlocks, o, 0, equCycle, Group)
                else:
                    isTraversed[j] = True
                    DFS_Traverse(potentialDeadlocks, i, s, di, k, isTraversed, Di, Group)
                    isTraversed[j] = False

def find_Equal_Dependency_Group(Group, D, d):
    for di in D:
        if di == d:
            return Group[di]
    return []

def cycle_detection(potentialDeadlocks, dc, D):
    Group = {}
    isTraversed = {}
    Di = {}

    for t in D.threads:
        isTraversed[t] = False
        Di[t] = []

    for d in D.lockDependencies:
        if d.lockObjectName in dc and d.currentlyOwnedLockObjectNames:
            g = find_Equal_Dependency_Group(Group, Di[d.threadName], d)
            if g:
                g.add(d)
            else:
                Di[d.threadName].append(d)
                Group[d] = []
                Group[d].append(d)

    s = []
    for t in D.threads:
        for d in Di[t]:
            isTraversed[t] = True
            DFS_Traverse(potentialDeadlocks, t, s, d, D.threads, isTraversed, Di, Group)

```

Quellcode C.0.5: magiclockLib/cycleDetection.py: Implementierung des
Magiclock-Algorithmus zur Zyklenerkennung

```
import traceFileReader
```

```

import magiclockLib.magiclockTypes as magiclockTypes
import magiclockLib.lockReduction as lockReduction
import magiclockLib.cycleDetection as cycleDetection

def get_OwnedLockObjects(threadName, ownedLockObjectsByThread):
    if threadName not in ownedLockObjectsByThread:
        ownedLockObjectsByThread[threadName] = []
    return ownedLockObjectsByThread[threadName]

def remove_From_List(x, lst):
    if x in lst:
        lst.remove(x)

def create_LockDependencyRelation(lockActions):
    lockDependencyRelation = magiclockTypes.LockDependencyRelation()
    ownedLockObjectsByThread = {}
    for lockAction in lockActions:
        ownedLockObjects = get_OwnedLockObjects(lockAction.threadName,
        ↪ ownedLockObjectsByThread)
        if lockAction.actionType == traceFileReader.LockActionType.LOCK:
            lockDependencyRelation.add(magiclockTypes.LockDependency(lockAction.threadName,
            ↪ lockAction.lockObjectName, ownedLockObjects.copy()))
            ownedLockObjects.append(lockAction.lockObjectName)
        elif lockAction.actionType == traceFileReader.LockActionType.UNLOCK:
            remove_From_List(lockAction.lockObjectName, ownedLockObjects)
    return lockDependencyRelation

def find_potential_Deadlocks(traceFilename):
    lockActions = traceFileReader.read_Trace_File_Lines(traceFilename)

    lockDependencyRelation = create_LockDependencyRelation(lockActions)
    lockDependencyRelation.print()

    initClassification = lockReduction.init_Classification(lockDependencyRelation)
    initClassification.print()

    lockClassification, lockDependencyRelation =
    ↪ lockReduction.lock_Reduction(lockDependencyRelation, initClassification)
    print("\nLock reduction Result:")
    print("Cyclic-set:")
    print(lockClassification.cyclicSet)
    lockDependencyRelation.print()

    disjointComponents =
    ↪ cycleDetection.disjoint_Components_Finder(lockClassification.cyclicSet,
    ↪ initClassification.edgesFromTo)
    print("\nDisjoint Components:")
    print(disjointComponents)

    potentialDeadlocks = []
    for dc in disjointComponents:
        cycleDetection.cycle_detection(potentialDeadlocks, dc, lockDependencyRelation)

    print("\nPotential Deadlocks:")
    for potentialDeadlock in potentialDeadlocks:
        for d in potentialDeadlock:
            d.print()
            print(end=" ")
        print()

    return potentialDeadlocks

```

Quellcode C.0.6: magiclockLib/magiclock.py: Implementierung des Magiclock-Algorithmus

```

import sys
import networkx as nx
import matplotlib.pyplot as plt
import ntpath

import magiclock_main as magiclock

```

```

traceFilename = sys.argv[1]
deadlockGraph = magiclock.get_potential_Deadlock_Nodes(traceFilename)

DG = nx.DiGraph()
for e in deadlockGraph.edges:
    DG.add_edge(e.fromNode, e.toNode, label=e.text)

plt.subplot(121)
plt.title("Potential Deadlocks found in: " + ntpath.basename(traceFilename))

pos = nx.spring_layout(DG)
nx.draw(DG, pos, with_labels=True, font_weight='bold', connectionstyle='arc3, rad = 0.1',
↪ node_size=1000)

edge_labels = nx.get_edge_attributes(DG, 'label')
nx.draw_networkx_edge_labels(DG, pos, edge_labels, label_pos=0.3)

plt.show()

```

Quellcode C.0.7: generateDeadlockGraph.py: Skript zur Erkennung und Darstellung von potentiellen Deadlocks

```

import sys
import time
import subprocess

times = 0
for x in range(1, 4):
    timeStarted = time.time()
    process = subprocess.check_call(['prl', '-r', sys.argv[1]])
    timeEnd = time.time()

    times += timeEnd - timeStarted
print("Finished process in " + str(times / 3) + " seconds.")

```

Quellcode C.0.8: benchmark_cpu.py: Skript zur Messung der CPU-Laufzeit einer OpenPEARL Anwendung

```

import sys
import psutil
import time
import subprocess
import os

maxMemoryUsed = 0
for x in range(1, 4):
    process = subprocess.Popen(['prl', '-r', sys.argv[1]])

    maxMemory = 0
    while process.poll() == None:
        parent = psutil.Process(os.getpid())
        memory = parent.memory_info().rss / 1024 / 1024

        for child in parent.children(recursive=True):
            memory += child.memory_info().rss / 1024 / 1024

        if memory > maxMemory:
            maxMemory = memory
        time.sleep(.01)

    process.wait()
    maxMemoryUsed += maxMemory
print("Process used " + str(maxMemoryUsed / 3) + " MB.")

```

Quellcode C.0.9: benchmark_memory.py: Skript zur Messung der Speicherauslastung einer OpenPEARL Anwendung