Fehlersuche und Visualisierung der Belegung von Synchronisationsmitteln in nebenläufigen Systemen

Marcel Sobottka

30. April 2020

Erstgutachter: Prof. Dr.-Ing. habil. Herwig Unger
Zweitgutachter: Dipl.-Inform. (Univ.) Marcel Schaible

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

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 PEARL 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 dargestellt was ein Deadlock ist und wie dynamische Verfahren zur Erkennung von Deadlocks funktionieren. In Abschnitt 2.2 wird die Echtzeit-Programmiersprache PEARL [3] beschrieben. Es wird gezeigt welche Synchronisationsmittel in PEARL existieren und wie diese benutzt werden. In Abschnitt 2.3 wird das Open-PEARL Projekt und der Aufbau der OpenPEARL Umgebung sowie das Zusammenspiel des Compilers und der Laufzeitumgebung aufgezeigt. Anschließend wird ein Algorithmus zur Erkennung von potenziellen Deadlocks in Abschnitt 2.4 beschrieben. 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 3.2 wird der in PEARL 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 3.2 wird das Design des Programms zur Visualisierung der chronologischen Belegung der Synchronisationsmittel basierend auf den Informationen der erstellten Trace-Datei definiert. In Abschnitt 3.3 wird eine Erweiterung des Analyse-Programms vorgestellt. Zusätzlich zu der chronologischen Belegung der Synchronisationsmittel werden potenzielle Deadlocks erkannt. Es wird beschrieben wie die Visualisierung aussehen und wie der in Abschnitt 2.4 vorgestellte Algorithmus umgesetzt werden soll.
In Abschnitt 4.1 wird die Implementierung zur Erstellung der Trace-Datei in PEARL beschrieben. Anschließend werden die Implementierungen des Programms 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 beschrieben. Dabei werden vor allem die definierten Performanz Anforderungen validiert. Abschließend werden in Kapitel 6 offene Punkte und mögliche Weiterentwicklung beschrieben.

2 Analyse

2.1 Deadlockerkennung allgemein

Im Gegensatz zu Single-Threaded-Applikationen 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 wollen. 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 verhindert werden Synchronisierungsmechanismen 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.
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 jetzt der Thread *T1* das Lockobjekt *L2* anfordert und der Thread *T2* das Lockobjekt *L1*, kommt es zu einem *Deadlock*. 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. Die statische Deadlockerkennung analysiert den Quellcode und wird hier nicht näher betrachtet. Die dynamische Deadlockerkennung analysiert eine Anwendung zur Laufzeit und läuft in folgenden drei Schritten ab:

1. Erstellung einer Trace-Datei
2. Erstellung eines Graphen basierend auf den Informationen aus der Trace-Datei
3. Auffinden von potenziellen Deadlocks durch die Identifizierung von Zyklen innerhalb des Graphen

Eine Trace-Datei enthält einen *execution trace* des ausführenden Programms. Ein *execution trace* ist eine Abfolge von Events. Ein Event e_i wird durch eine der folgenden Methoden definiert: starten eines Threads, Inbesitznahme eines Lockobjekts und Freigabe eines Lockobjekts. Das Starten eines neues Threads ist definiert durch ein Thread-Start-Event:

```
s(ausführender Thread, Name des neuen Threads)
```

Zum Beispiel bedeutet `s(main,T1)`, dass der Thread *main* den Thread *T1* gestartet hat. Die Inbesitznahme eines Lockobjekts ist definiert durch ein Lock-Event:

```
l(ausführender Thread, Name des Lockobjekts)
```

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

```
u(ausführender Thread, Name des Lockobjekts)
```

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. 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 den Lock *L2* anfordert.

2.2 PEARL

Die Programmiersprache PEARL wurde in den 1970er Jahren vom Institut für Regelungstechnik der Universität Hannover entwickelt [2]. 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. Das Deutsche Institut für Normung standardisierte PEARL mehrmals, unteranderem 1998 in der DIN 66253-2 als PEARL90 [4] und zuletzt 2018 als SafePEARL [5]. 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. Um mehrere *TASKs* zu synchronisieren gibt es zwei Möglichkeiten: *SEMA* und *BOLT* Variablen.
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 [3, S. 9–17]. 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. Das Zustandsdiagramm zur *SEMA* Variable ist in Abb. 2.2.1 dargestellt.

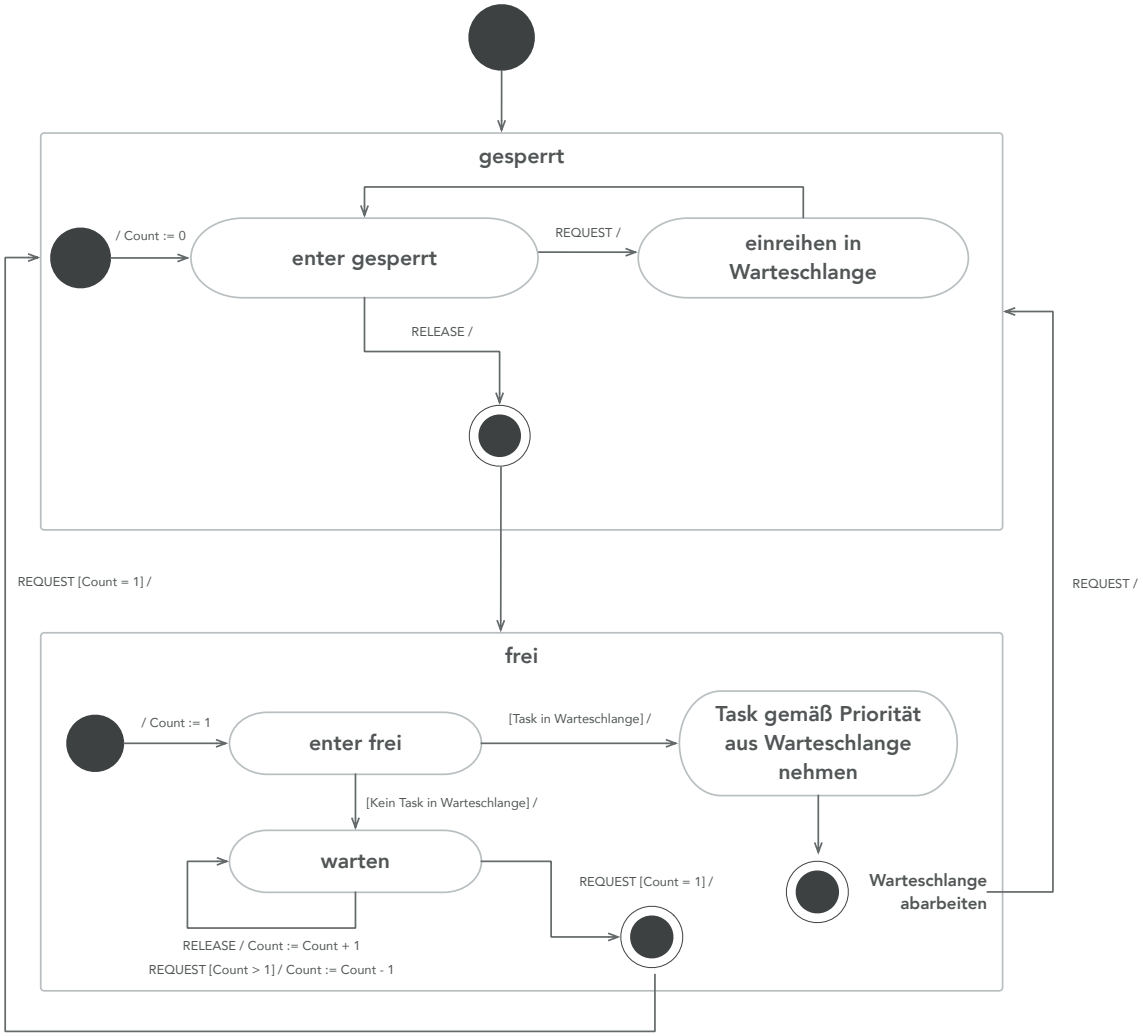

Abbildung 2.2.1: Zustandsdiagramm einer SEMA Variablen

BOLT Variablen haben im Gegensatz zu *SEMA* Variablen drei Zustände „gesperrt“, „Sperre möglich“ und „Sperre nicht möglich“ [3, S. 9–17]. 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 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 ie 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. Das Zustandsdiagramm zur *BOLT* Variable ist in Abb. 2.2.2 dargestellt.

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 [7]. Unterstützt wird der PEARL90 Standard bis auf einige wenige Unterschiede [6].
OpenPEARL besteht aus drei wesentlichen Komponenten:

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 prüft deren Kompatibilität.

In Quellcode 2.1 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 1 bis 11 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. 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. Diese Situation wird als Deadlock bezeichnet.

2.4 MagicLock

Der nachfolgende Abschnitt basiert auf den Ausführungen in [1].
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.
MagicLock analysiert einen *execution trace*[1] einer Programmausführung ohne Deadlocks. Ein möglicher *execution trace* von dem Beispielprogramm aus Quellcode 2.1 ist:

σ = s(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 Lock-Objekt und L eine Menge von Lock-Objekten. Das Triple sagt aus, dass der Thread t das Lock-Objekt m in Besitz nimmt, während er jedes Lock-Objekt in L besitzt.
Bei einem Thread-Start-Event wird ein neuer Thread-Identifier und eine leere Menge an Locks für den neu erzeugten Thread erstellt. Zum Beispiel wird bei den Event *s(main,T1)*

[1] siehe Abschnitt 2.1

ein neuer Thread-Identifier 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 Lock-Objekt L2 aus der Menge L_{T2} entfernt. Daraus folgt die Lock-Dependency-Relation:

D_σ = (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 Lock-Objekten im *execution trace*. Der Algorithmus entfernt alle Lock-Objekte aus der Menge aller Lock-Objekte *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 Lock-Objekt nur Teil eines Zyklus sein kann, wenn dieses mindestens eine eingehende und mindestens eine ausgehende Kante besitzt. Zusätzlich werden alle Lock-Objekte entfernt, welche nur von einem einzigen Thread in Besitz genommen bzw. freigegeben wurden. Wenn nur ein Thread ein Lock-Objekt benutzt, kann dieses Lock-Objekt nicht Teil eines Deadlocks sein. Mit den reduzierten Lock-Objekten wird im nächsten Schritt die Zyklensuche vorbereitet.
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. Zwei Lock-Dependencies sind gleich wenn gilt:

Gegeben sind zwei Lock-Dependencies $T_1 = (t_1, m_1, L_1)$ und $T_2 = (t_2, m_2, L_2)$:
$T_1 = T_2 \Leftrightarrow t_1 = t_2 \wedge m_1 = m_2 \wedge L_1 = L_2$

Anschließend werden die Partitionen gegeneinander auf Lock-Dependency-Chains geprüft. Eine Lock-Dependency-Chain ist eine Sequenz von Lock-Dependencies für die gilt:

$d_{chain} = (T_1, T_2, \ldots, T_k)$ mit $T_i = (t_i, m_i, L_i)$, wenn $m_1 \in L_2 \ldots m_{k-1} \in L_k, t_i \neq t_j$ und $L_i \cap L_j = \emptyset$ für $1 \leq i, j \leq k (i \neq j)$

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

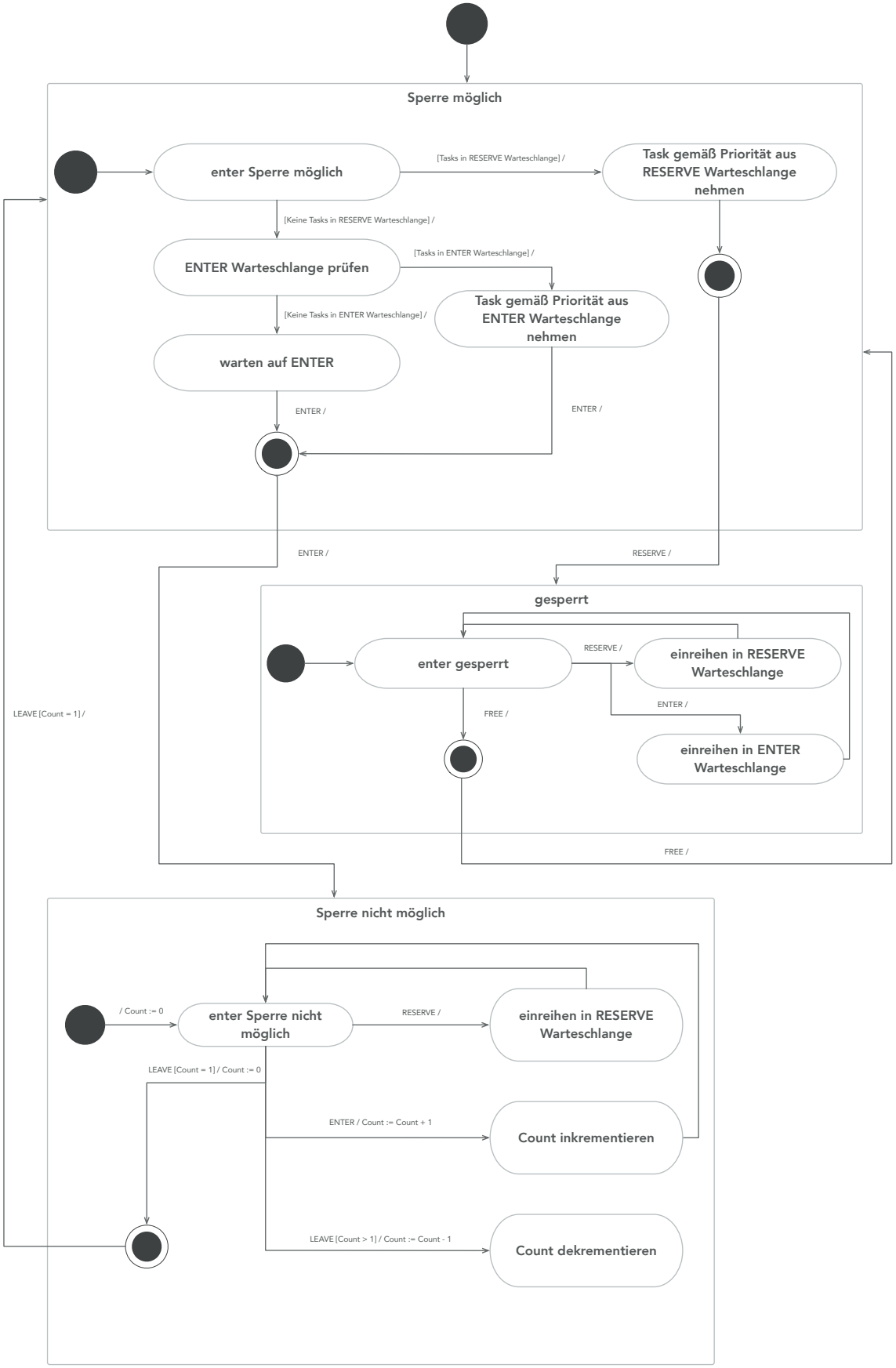

Abbildung 2.2.2: Zustandsdiagramm einer BOLT Variablen

```
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 2.1: Beispiel einer OpenPEARL Anwendung mit einem potenziellen Deadlock

3 Design

3.1 Erzeugung der Trace-Datei

- Definition der notwendigen Informationen der Trace-Datei
- Aufzeigen der Herausforderungen in Bezug auf Performance und Speicherauslastung
- Anforderungen definieren:
  - Laufzeit des Programms soll sich um maximal x% erhöhen
  - Speicherauslastung des Programms soll sich um maximal y% erhöhen
- Aktuelle Idee: Das Erstellen der Trace-Datei wird in der Laufzeitumgebung umgesetzt. Die Implementierung erfolgt in einer eigenen Klasse. Die Klasse kann optional von der Semaphore Implementierung der Laufzeitumgebung verwendet werden. Beim Aufruf von Request oder Release wird geprüft, ob die Informationen geloggt werden sollen. Falls das Logging aktiviert ist, werden die Informationen in eine Warteschlange eingereiht. Die Warteschlange wird über eine einfach verkettete Liste realisiert. Das Einfügen in eine einfach verkettete Liste ist konstant und geschieht daher in O(1). Bei jeder x-ten Aktivierung, wobei x die maximale Größe der Warteschlange ist, werden alle vorhanden Einträge aus der Warteschlange entfernt und in die Trace-Datei geschrieben. Das Entfernen des ersten Eintrags aus der Warteschlange (einfach verkettete Liste) ist wieder konstant und daher O(1). Die Liste wird für jeden Eintrag einmal durchlaufen. Die Laufzeit beträgt daher O(x). Das Öffnen und Schließen der Trace-Datei hat die Laufzeit y. Das Schreiben eines Logeintrags, also eine Zeile in die Trace-Datei, hat die Laufzeit z. Damit hat jeder x-te Request/Release Aufruf die Laufzeit O(y + x · z). Somit hat jeder Request/Release Aufruf die Laufzeit O($\frac{y+x \cdot z}{x}$).

  Zusätzlich zu jedem x-ten Request/Release Aufruf das Schreiben in die Trace-Datei auch zeitich angestoßen werden. Zum Beispiel sollte alle 60 Sekunden oder nach x Aufrufen die Warteschlange geleert und in die Trace-Datei geschrieben werden. Bei jedem x-ten Request/Release Aufruf muss der Timer der Warteschlange wieder zurückgesetzt werden. Bedeutet wenn nach 50 Sekunden der x-te Aufruf kommt, muss der Timer wieder von vorne beginnen, damit die Trace-Datei nicht nach 10 Sekunden erneut beschrieben wird.

Der zusätzliche Speicherbedarf beträgt $x \cdot i$, wobei i die Größe der Information eines Logeintrags entspricht. Ein Logeintrag benötigt die Informationen:

1. Aktion (REQUEST oder RELEASE) => 1 Bit
2. ID des Threads => 16-Bit Ganzzahl (maximale Thread Id für Linux beträgt 32768) theoretisch über 15 Bit abbildbar, da immer positiv
3. Name des Lockobjekts => 11 Bit Ganzzahl (maximal Variablennamenlänge für C++ beträgt 2048)

Mindestgröße für i beträgt 1 + 16 + 11 = 28 Bit. Die Größe eines Zeigers in C++ beträgt auf einem 64 Bit System 64 Bit. Ein Eintrag in der Warteschlange benötigt somit 28 Bit + 64 Bit = 92 Bit Speicher. Die Warteschlange benötigt maximal daher 92 Bit · x zusätzlichen Speicher.

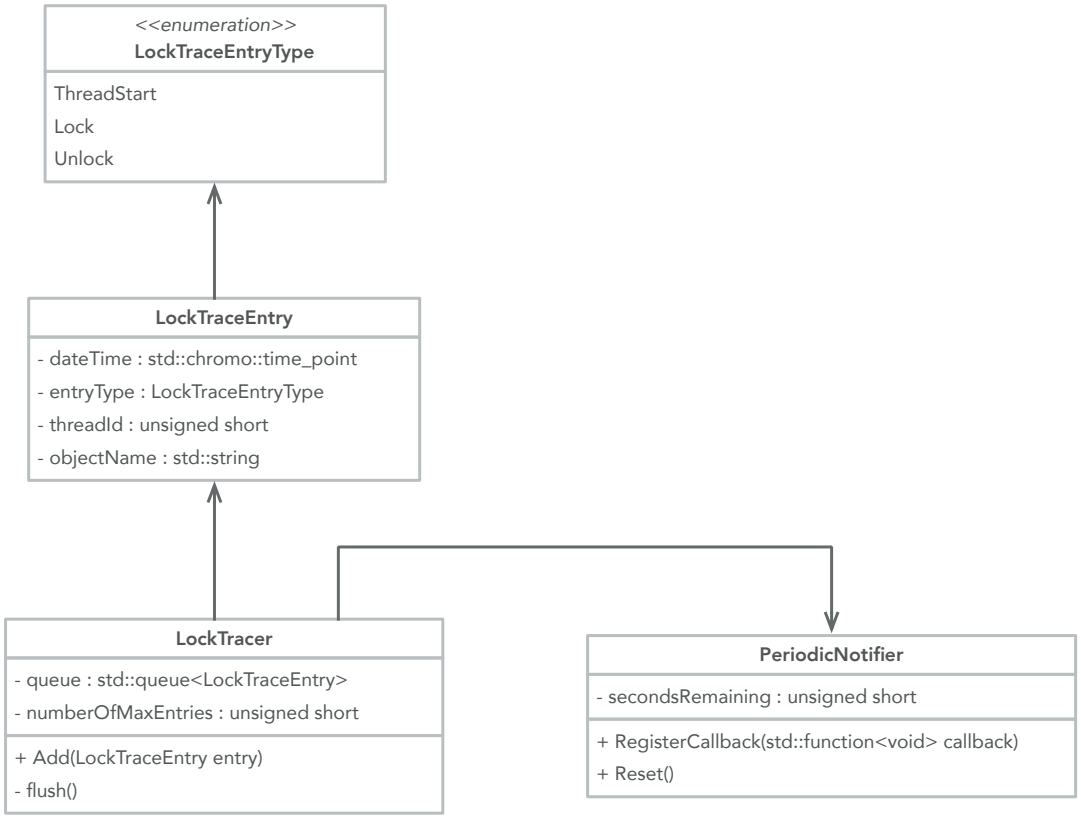

Abbildung 3.1.1: UML Klassendiagramm für die Erzeugung der Trace-Datei

3.2 Analysieren der Trace-Datei

- Externes Programm geschrieben in Java
- Anforderungen definieren:
  - Darstellung der Trace-Datei (welcher Thread hat welches Synchronisationsmittel wann genommen und wieder freigegeben)

3.3 Erweiterung: Potenzielle Deadlocks

- Programm aus Abschnitt 3.2 wird erweitert
- Es sollen potenzielle Deadlocks mit Hilfe des aus Abschnitt 2.4 beschriebenen Verfahrens bestimmt werden
- Potenzielle Deadlocks sollen als gerichtete Graphen visualisiert werden

4 Implementierung

4.1 Trace Funktion

- Vorstellung der Implementierung der Trace Funktionalität in PEARL

4.2 Analyse Programm

- Vorstellung der Implementierung des Analyse Programms in Java
- Grafiken/Screenshots mit Analyse Beispielen

4.3 Visualisierung von potenziellen Deadlocks

- Vorstellung der Implementierung des Algorithmus zur Erkennung von potenziellen Deadlocks MagicLock klassifiziert dazu jedes Lock-Objekt in eine der folgenden Mengen:
  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$\}$
- Grafiken/Screenshots mit Analyse Beispielen

5 Validierung

5.1 Trace Funktion

- Beispielprogramme in PEARL definieren
- Anforderungen validieren:
  - Laufzeiten der einzelnen Programme mit und ohne Trace Funktionalität bestimmen und vergleichen
  - Speicherauslastung der einzelnen Programme mit und ohne Trace Funktionalität bestimmen und vergleichen

5.2 Analyse Programm

- Die aus Abschnitt 5.1 erstellten Trace-Dateien mit dem erstellten Analyse Programm auswerten

5.3 Visualisierung von potenziellen Deadlocks

- Die aus Abschnitt 5.1 erstellten Trace-Dateien auswerten und die potenziellen Deadlocks visuell als gerichtete Graphen aufzeigen

6 Ausblick

6.1 Offene Punkte

- Aufzeigen was nicht gemacht wurde und warum (eventuell welche Synchronisationsmittel nicht erkannt werden (nur Sema keine Bolt Variablen))

6.2 Weiterentwicklung

- Alle möglichen Synchronisationsmittel erkennen
- Trace Funktionalität in den OpenPEARL Compiler integrieren, so dass es mittels compiler flag an und ausgeschaltet werden kann

Literatur

[1] 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. 13).

[2] 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).

[3] 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. 7, 11, 12).

[4] *Informationstechnik - Programmiersprache PEARL - PEARL 90*. Norm. Apr. 1998 (siehe S. 11).

[5] *Informationsverarbeitung - Programmiersprache PEARL - SafePEARL*. Norm. März 2018 (siehe S. 11).

[6] Marcel Schaible und Rainer Müller. *Differences between PEARL90 and OpenPEARL*. 2019. URL: `https://sourceforge.net/p/openpearl/wiki/Differences%20between%20PEARL90%20and%20OpenPEARL/` (besucht am 13. Dez. 2019) (siehe S. 12).

[7] Marcel Schaible und Rainer Müller. *OpenPEARL*. 2019. URL: `https://sourceforge.net/projects/openpearl/` (besucht am 13. Dez. 2019) (siehe S. 12).