Struktur:

* Resource versus Communication Deadlock
* Analysemethoden: statisch und dynamisch
* Einfache wege, Deadlocks vorherzusagen

Um den Absturz von Programmen durch Deadlocks zu verhindern, wurden in der Vergangenheit viel Forschung betrieben, um Deadlocks vorherzusagen. Dies geschieht durch die Analyse des Programms. Bei den Analysemethoden für Deadlocks unterscheidet man zwischen der statischen und der dynamischen Deadlock-Analyse.

Die statische Deadlock-Analyse betrachtet ausschließlich den Quellcode für die Vorhersage von Deadlocks. Es gibt statische Ansätze, die dazu in der Lage sind, die Abwesenheit von Deadlocks zu beweisen. Allerdings sind statische Analysemethoden nicht gut skalierbar und zeigen in Programmen oft Deadlocks an, wo jedoch keine sind.

Eine Analysemethode, die solche „False-Positives“ anzeigt, wird auch „unsound“ genannt. Mit dem Begriff „sound“ wird in der Deadlock-Analyse eine Methode bezeichnet, wenn diese keine false positives vorhersagt.

Die dynamische Deadlock-Analyse betrachtet eine beispielhafte Ausführung des Programms in Form eines Trace als Basis für die Vorhersage. Die Ereignisse innerhalb des Trace werden auf bestimmte Weise umgeordnet, sodass Deadlocks erkannt werden können. Im Gegensatz zur statischen Deadlock-Analyse wird hierbei nicht das Ziel verfolgt, einen Beweis für die Abwesenheit von Deadlocks zu erbringen. Stattdessen wird sich darauf konzentriert durch Umordnungen von Traces theoretisch mögliche Deadlocks zu entdecken. Dynamische Methoden erkennen zwar nicht alle, aber dennoch viele Deadlocks. Außerdem sind sie dadurch, dass sie keinen Beweis erbringen müssen, skalierbarer und liefern nur wenige bis kein False-Positives. Ein solcher dynamischer Ansatz wird in der vorliegenden Arbeit im Kapitel „Sync-preserving Deadlocks“ vertieft beschrieben. Er erkennt die meisten Deadlocks, ist sehr gut zu skalieren und ist darüber hinaus sound.

Die gerade beschriebenen Analysemethoden müssen in der Lage sein, verschiedene Typen von Deadlocks vorherzusagen. Zum einen gibt es den sog. „Resource-Deadlock“, bei dem alle Threads des Programms auf die Freigabe einer Ressource warten, die von einem anderen Thread des Programms bereits reserviert wurde. Das Deadlock-Beispiel aus dem vorherigen Kapitel ist ein solcher Resource -Deadlock.

Weiterhin gibt es den sog. „Communication-Deadlock“, der dann auftritt, wenn alle Threads darauf warten mit einem anderen Thread kommunizieren zu können. Der folgende Code enthält einen solchen Communication-Deadlock:

Ein Channel in Go ist ein Objekt, welches ermöglicht, Daten zwischen Threads für nebenläufige Berechnungen auszutauschen. In Zeile 2 des obigen Codebeispiels wird ein solcher Channel erstellt. Der nebenläufige Thread in den Zeilen 4 - 7 möchte Daten aus dem Channel empfangen. Da es keinen Sender gibt, der Daten in dem Channel platziert, blockiert der nebenläufige Thread. Da der Main-Thread in Zeile 9 ebenfalls Daten aus demselben Channel empfangen will, aber kein Sender vorhanden ist, blockiert auch dieser. Dies führt dazu, dass sich beide Threads des Programms in einem Wartezustand befinden und ein Communication-Deadlock auftritt.