Struktur:

* Viele dynamische Ansätze fokussieren sich darauf, Deadlock Patterns zu erkennen (in Form von zyklischer Locks) im Trace
* Lock-Dependency-Methode
* Deadlock-Analyse anhand Lock Graphen erklären (Beispiele)
* Ein Deadlock Pattern ist eine notwendige aber keine hinreichende Bedingung
* False Positives aufzeigen

Bei der dynamischen Deadlock-Analyse wird versucht, theoretisch mögliche Deadlocks anhand eines Trace des Programms zu erkennen. Genauer gesagt wird versucht, sog. „Deadlock-Patterns“ in einem Trace zu erkennen. Deadlock-Patterns sind Muster, die darauf hindeuten, dass ein Trace einen potenziellen Deadlock enthält. Ein Deadlock-Pattern zu erkennen ist eine notwendige, aber keine hinreichende Bedingung für die Detektion eines Deadlocks. Das bedeutet, dass zwar ein Deadlock-Pattern für die Vorhersage eines Deadlocks vorhanden sein muss, jedoch nicht hinter jedem Deadlock-Pattern ein tatsächlicher Deadlock stehen muss.

Ein einfacher Algorithmus, um dynamisch Deadlocks zu erkennen ist die „Lock-Dependency-Methode“. Sie basiert auf der Idee von sog. „Lock-Graphen“. Ein Lock-Graph baut sich aus den Lock-Operationen nebenläufiger Threads auf gemeinsame Ressourcen bzw. Mutexes auf. Ein Lock auf eine Ressource stellt hierbei einen Knoten im Graph dar. Eine Kante von einem Lock-Knoten zu einem anderen Lock-Knoten entsteht dann, wenn ein nebenläufiger Thread einen der beiden Locks bereits hält und den anderen reservieren möchte. Nachdem der Lock-Graph aus dem Trace aufgebaut wurde, wird nach Zyklen im Lock-Graph gesucht. Der Zyklus im Lock-Graphen ist das Deadlock-Pattern dieses Algorithmus. Wenn mindestens ein Zyklus vorhanden ist, dann geht die Lock-Dependency-Methode davon aus, dass ein Deadlock vorhanden ist. Man betrachte den folgenden Trace:

**T1 T2**

**1. acq(y)**

**2. acq(x)**

**3. rel(x)**

**4. rel(y)**

**5. acq(x)**

**6. acq(y)**

**7. rel(y)**

**8. rel(x)**

Die Operationen acq (acquire) und rel (release) stehen hier für die Reservierung und die Freigabe eines Mutexes. Es ist zu sehen, dass Thread T1 erst y und dann x reserviert. Unser Graph bekommt dadurch die Knoten x und y, sowie eine Kante von y nach x. In Thread T2 wird erst x und dann y reserviert. Wir haben also wieder die Knoten x und y, die aber bereits in unserem Lock-Graph existieren. Hinzu kommt aber eine Kante von x nach y, da T2 x bereits hält, bevor y reserviert wird. Daraus ergibt sich der folgende Lock-Graph:

**y -> x**

**x -> y**

Der Lock-Graph wird im nächsten Schritt auf Zyklen untersucht. Wie im obigen Beispiel leicht zu erkennen ist, gibt es einen Zyklus zwischen den Knoten x und y. Die Lock-Dependency-Methode sagt hier also einen Deadlock voraus. Dies ist auch richtig, da der obige Trace auch so verlaufen kann, dass im ersten Schritt T1 y reserviert und im zweiten Schritt T2 x reserviert. Der nächste Schritt für beide Threads wäre dann die Reservierung des Mutex, der vom jeweils anderen Thread bereits gehalten wird. Da dies für beide Threads jedoch nicht möglich ist, resultiert diese Umordnung des Trace in einem Deadlock.

Allerdings sagt die Lock-Dependency-Methode auch häufig False-Positives vorher. So wie in dem folgenden Beispiel:

**T1 T2**

**1. acq(y)**

**2. acq(x)**

**3. rel(x)**

**4. rel(y)**

**5. acq(x)**

**6. acq(y)**

**7. rel(y)**

**8. rel(x)**

Hier werden alle Reservierungen und Freigaben der Ressourcen innerhalb von T1 ausgeführt. T2 benötigt keine Ressourcen, die zu reservieren wären. Aus dem obigen Trace resultiert der gleiche Lock-Graph wie schon im ersten Trace:

**y -> x**

**x -> y**

Da in diesem Lock-Graph ein Zyklus steckt sagt die Lock-Dependency-Methode wieder einen Deadlock vorher. Dieser kann in der Realität jedoch niemals auftreten, da nur T1 Ressourcen reserviert und freigibt.

Das Vorhersagen von False-Positives ist problematisch, da Softwareentwickler anschließend viel Zeit mit der Suche und Verhinderung des Deadlocks verbringen würden, ohne dass dieser überhaupt existiert. Aus diesem Grund wurde eine weitere dynamische Analysemethode entwickelt, die keine False-Positives vorhersagt und dennoch einen Großteil der potenziellen Deadlocks erkennt. Auf diese Methode wird im nachfolgenden Kapitel eingegangen.