Linearisierbarkeit als Korrektheitseigenschaft für nicht-blockierende Datenstrukturen

Jakob Krainz
jakob@hawo-net.de

ABSTRACT

In diesem Papier wird Linearisierbarkeit definiert und ihre Vor– und Nachteile auch in Hinblick auf alternative Korrekteitseigenschaften für konkurrente Datenobjekte beschrieben.

Linearisierbarkeit ist eine Korrektheitseigenschaft für konkurrente Datenobjekte. Sie erlaubt es, eine abstrakte Spezifikation der Objekte aus dem Uniprozessorbereich zu verallgemeinern um sie in Multiprozessorsystemen anwenden zu können.

Die Linearisierbarkeit einer Datenstruktur ist damit ein Hilfsmittel zum Nachweis des korrekten Verhaltens eines Programms für Multiprozessorsysteme. Außerdem kann Linearisierbarkeit dabei helfen, die Echtzeiteigenschaften eines Programms zu zeigen.

1. EINLEITUNG

Eine interessante und wichtige Problemstellung im Kontext der Entwicklung von Multiprozessor–Applikationen ist die formale Beschreibung des Verhaltens der Applikation und dessen Verifikation. Im Allgemeinen werden beim Entwurf Objekte verwendet, deren zulässige Operationen aus dem Bereich der Uniprozessor–Applikationen bekannt sind, wie z. B. Warteschlangen und Binärbäume.

Diese Objekte sind jedoch meist ungeeignet für nebenläufige Zugriffe: Ihre abstrakten Spezifikationen gelten nur für sequentielle Verwendung und die nebenläufige Verwendung ihrer konkreten Implementierungen kann zu schwer reproduzierbaren Fehlern führen[1]. Gesucht ist daher eine Möglichkeit, das aus dem Single-Thread-Bereich bekannte abstrakte Verhalten einer Datenstruktur auf den Multiprozessorbereich zu verallgemeinern.

[1]Fehler z. B., die aus Wettlaufsituationen folgen, die aus nicht reproduzierbarer minimaler Änderungen des zeitlichen Ablaufs entstehen.

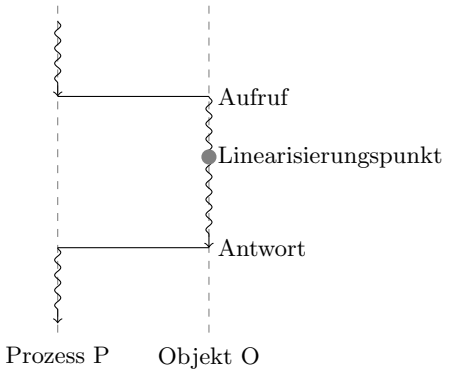

Abbildung 1: Linearisierungspunkt eines Zugriffs

Hierzu gibt es mehrere Möglichkeiten. Ich möchte zuerst einen Überblick geben über die von Herlihy und Wing [2] eingeführte Linearisierbarkeit und dann ihre Vorteile, die sie gerade für große Systeme oder Szenarien mit Echtzeitanforderungen interessant machen. Abschließend möchte ich sie mit zwei anderen Korrektheitseigenschaften, der Serialisierbarkeit und der sequentiellen Konsistenz, vergleichen.

2. LINEARISIERBARKEIT

2.1 Definition

2.1.1 Informelle Definition

Eine Datenstruktur ist linearisierbar, wenn jede Ablauffolge von nebenläufigen Zugriffen auf diese Datenstruktur "umformbar" ist in eine Ablauffolge von nicht nebenläufigen Zugriffen. Zusätzlich darf sich die zeitliche Abfolge von nicht nebenläufigen Zugriffen bei dieser Umformung nicht ändern (vgl. [7], S. 301f; [2], S. 464).

Alternativ kann man Linearisierbarkeit auch folgendermaßen informell definieren (vgl. [2]§1.2, S. 464f; Darstellung 1):

Definition 1. Eine Ablauffolge ist linearisierbar, wenn zu jedem Zugriff ein Zeitpunkt während dieses Zugriffes, der Linearisierungspunkt, existiert, so dass der Zugriff zu exakt diesem Zeitpunkt eine atomare Zustandstransition bewirkt.

In Darstellung 1 ist ein Zugriff eines Prozesses auf ein Objekt mit zugehörigem Linearisierungspunkt illustriert.

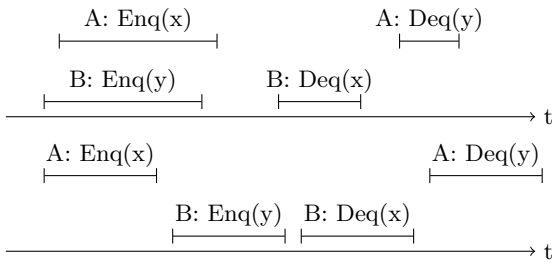

Abbildung 2: Eine linearisierbare Ablauffolge und die dazugehörige Linearisierung

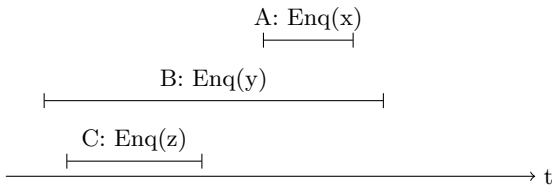

Abbildung 3: Eine mehrdeutig linearisierbare Ablauffolge

Diese Definition bietet zunächst nur eine hinreichende Bedingung für Linearisierbarkeit[2]. Wichtig ist, dass man sich bei jeder linearisierbaren Datenstruktur vorstellen kann, dass ihre Zugriffe irgendwann zwischen Aufruf und Antwort eine atomare Zustandstransition hervorrufen.

Ein Zugriff wie in Darstellung 1 beginnt mit einem Aufruf einer Methode an einem Objekt und endet, wenn dieser Aufruf zurückkehrt. Nebenläufigkeit von Zugriffen entsteht dadurch, dass ein Zugriff gestartet wird obwohl ein anderer Zugriff das Objekt noch bearbeitet. In Abbildung 2 gilt das für die zwei Enqueue–Zugriffe.

Hier bearbeiten zwei Prozesse A und B eine gemeinsame Datenstruktur, nämlich eine FIFO–Warteschlange. Die zwei Enqueue–Zugriffe passieren nebenläufig; die zwei Dequeue–Zugriffe sequentiell.

Diese Ablauffolge ist linearisierbar, indem die beiden nebenläufigen Zugriffe sequentialisiert werden, sie also in nicht nebenläufige Zugriffe umgeformt werden.

Hierbei gäbe es zwei Möglichkeiten: Entweder passiert zuerst das Enqueue von A und dann das Enqueue von B oder andersherum. Die zweite Möglichkeit verletzt jedoch die FIFO–Eigenschaft der Datenstruktur, denn das Dequeue von x passiert vor dem Dequeue von y; also kann das Enqueue von y nicht vor dem Enqueue von x passiert sein.

In Abbildung 3 finden auf der FIFO–Warteschlange drei Enqueues statt. Der Enqueue von Prozess C ist zu Ende bevor das Enqueue von A beginnt. Deswegen muss die Warteschlange am Ende der Ablauffolge "z" vor "x" enthalten,

[2]Tatsächlich gibt es viele Datenstrukturen, die linearisierbar sind obwohl sie die diese Bedingung nicht erfüllen können, da die nötigen Zustandsänderungen mangels Hardwareunterstützung nicht atomar ausführbar sind.

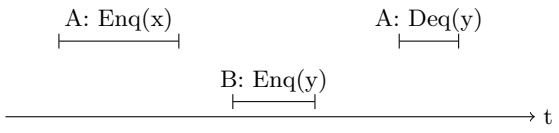

Abbildung 4: Eine nicht linearisierbare Ablauffolge

da sonst die FIFO–Eigenschaft verletzt ist. Das Enqueue von Prozess B passiert jedoch parallel zu den beiden anderen Enqueues. Damit ist nicht festlegbar, ob "y" vor oder nach "z" oder "x" in der Warteschlange steckt. Die möglichen Endzustände der Warteschlange sind "[z,x,y]", "[z,y,x]" und "[y,z,x]".

Die Ablauffolge in Abbildung 4 ist nicht linearisierbar. In einer Linearisierung muss die zeitliche Reihenfolge von nicht nebenläufigen Zugriffen erhalten bleiben. Da die Ablauffolge nur Zugriffe enthält die sich nicht überschneiden, kann bei einer Linearisierung nichts an der Reihenfolge der Zugriffe geändert werden. Die sequentialisierte Ablauffolge widerspricht damit zwangsläufig der abstrakten Definition der FIFO–Warteschlange, also kann sie nicht linearisiert werden.

Aus diesen Beispiel ist ersichtlich: Ablauffolgen sind potentiell auf mehrere Arten linearisierbar; eine Linearisierung respektiert jedoch die abstrakte Spezifikation[3] der Datenstruktur.

2.1.2 Formale Definition

Zu einer formal akzeptablen Definition müssen wir zuerst einige Begriffe klären.

Die grundlegenden Elemente unseres Modells sind Prozesse und Objekte.

Definition 2. Ein Prozess ist eine Recheneinheit, die sequentiell arbeitet und auf Objekte zugreift; diese Zugriffe sind ebenfalls sequentiell, d. h. ein Prozess startet einen neuen Zugriff erst nach Beendigung des Vorhergehenden.

Definition 3. Ein Zugriff besteht aus zwei (atomaren) Ereignissen: dem Aufruf und der zugehörigen Antwort[4]. Jedem dieser Ereignisse kann man genau einen Prozess (den Zugreifenden) und ein Objekt (das Bearbeitete) zuordnen.

Definition 4. Eine Ablauffolge[5] besteht aus einer zeitlichen Abfolge von Aufrufen und Antworten.

Definition 5. Zwei Zugriffe einer Ablauffolge sind nebenläufig, wenn die zwei Zeitintervalle zwischen Aufruf und Antwort sich überlappen, d. h. einer der beiden Aufrufe findet zwischen dem anderen Aufruf und dessen Antwort statt.

Zwei nicht nebenläufige Zugriffe sind zueinander sequentiell. Ein Zugriff an sich ist sequentiell, wenn er zu allen anderen Zugriffen sequentiell ist.

[3]die im Allgemeinen nur für den sequentiellen Fall gilt.
[4]die zeitlich später stattfindet
[5]im Original "history" [2], S. 467

Eine Ablauffolge ist sequentiell, wenn alle Zugriffe in ihr sequentiell sind.

Auf der Menge der Zugriffe in einer Ablauffolge H entsteht eine partielle Ordnungsrelation $<_H$, die der zeitlichen Abfolge entspricht: Zu zwei Zugriffen Z_1 und Z_2 in einer Ablauffolge H gilt: $Z_1 <_H Z_2$ genau dann, wenn Z_1 und Z_2 nicht nebenläufig sind und Z_1 vor Z_2 passiert ist.

Definition 6. Zu einer Ablauffolge H und einem Prozess P ist die Teilablauffolge[6] H|P definiert als die Ablauffolge von Zugriffen, die in H sind und bei denen P beteiligt ist. H|P ist für alle Prozesse P sequentiell.

Analog ist eine Teilablauffolge von H auch für ein Objekt O definierbar: H|O ist die Teilablauffolge von Zugriffen, die in H sind und bei denen auf O zugegriffen wird.

Definition 7. In einer Ablauffolge kann es Zugriffe geben, deren Antwortereignis noch nicht passiert ist; diese Zugriffe sind unvollständig[7].

Zu einer Ablauffolge H ist die Teilablauffolge complete(H) definiert als die Teilablauffolge, die aus allen vollständigen Zugriffen besteht, d. h. die Teilablauffolge, bei der alle unvollständigen Zugriffe weggelassen werden.

Der Begriff der Teilablauffolge erlaubt uns, die Umformung von Ablauffolgen genau zu definieren:

Definition 8. Zwei Ablauffolgen H und H' über der gleichen Menge von Prozessen und Objekten sind äquivalent wenn für alle beteiligten Prozesse P gilt: H|P und H'|P bestehen aus Zugriffen, die in der gleichen Reihenfolge auf die gleichen Objekte mit den gleichen Parametern und Ergebnissen stattfinden; lediglich der zeitliche Abstand zwischen zwei Zugriffen oder die Dauer der einzelnen Zugriffe darf sich ändern.

Damit kann nun die Linearisierbarkeit definiert werden:

Definition 9. Eine Ablauffolge H ist linearisierbar, wenn zwei weitere Ablauffolgen H' und S existieren, die folgenden Bedingungen genügen:

- H' entsteht, indem an H einige Antworten zu unvollständigen Zugriffen angehängt werden, diese also vervollständigt[8].
- S ist zu complete(H') äquivalent[9].
- S ist sequentiell.
- Die Spezifikation aller Objekte erlaubt S[10]
- Für alle Zugriffe Z_1 und Z_2 in H, die nicht nebenläufig sind, für die also gilt: $Z_1 <_H Z_2$, gilt ebenfalls $Z_1 <_S Z_2$.

Mit anderen Worten: Um eine Ablauffolge zu linearisieren ist für nebenläufige Zugriffe eine sequentielle Reihenfolge zu wählen. Das Ergebnis muss eine erlaubte sequentielle Ablauffolge darstellen, darf also die sequentielle Spezifikation der beteiligten Objekte nicht verletzen.

Damit stellt Linearisierbarkeit eine Möglichkeit dar, aus einer Spezifikation des Verhaltens einer Datenstruktur für sequentielle Szenarien eine Spezifikation für nebenläufige Szenarien zu machen.

Wichtig hierbei ist, dass Linearisierbarkeit Wettlaufsituationen[11] außerhalb der Operationen nicht ausschließt: Zu einer Ablauffolge kann es mehrere zulässige Linearisierungen geben, dies entspricht intuitiv unserer Vorstellung von Nebenläufigkeit (vgl. Darstellung 3).

2.2 Anwendbarkeit der Linearisierbarkeit

Aus der Linearisierbarkeit eines Objekts sind Schlussfolgerungen über dessen Verhalten ziehbar. Ich möchte nun ein Beispiel dazu, das von Herlihy und Wing ([2], S. 482 ff.) stammt, nachvollziehen. In diesem Beispiel wird das korrekte Verhalten einer FIFO–Warteschlange aus ihrer Linearisierbarkeit abgeleitet.

Intuitiv ist eine Warteschlange, deren Verhalten in sequenziellen Szenarien festgelegt ist, in nebenläufigen Szenarien dann korrekt, wenn sie die Reihenfolge der Elemente erhält, deren Einfüge–Zugriffe nicht nebenläufig waren, wenn keine Elemente verschwinden und wenn keine Elemente in der Warteschlange erscheinen ohne eingefügt worden zu sein.

Ich will nun zeigen, dass diese Bedingungen von einer linearisierbare FIFO–Warteschlange erfüllt werden.

Sämtliche Werte in der Warteschlange seien zur Vereinfachung des Beweises unterschiedlich. Des Weiteren sei die Warteschlange zu Beginn der Ablauffolge leer.

Satz 1. *Wenn zwei sequentielle Enqueue–Operationen die Werte x und y nacheinander in die Warteschlange einfügen und x und y entfernt werden, so sind die zwei Dequeue–Operationen entweder nebenläufig oder die Dequeue–Operation von x findet vor der Dequeue–Operation von y statt.*

Satz 2. *Wenn zwei sequentielle Zugriffe die Werte x und y nacheinander in die Warteschlange einfügen und y bereits entfernt wurde, so gibt es entweder eine vollständige Dequeue–Operation von x oder es gibt eine unvollständige Dequeue–Operation von x, die nebenläufig ist zu Dequeue–Operation von y.*

[6] im Original "subhistory" [2], S. 467
[7] im Original "pending" [2], S. 467
[8] Diejenigen unvollständigen Zugriffe, die bereits Auswirkungen auf die Datenstruktur hatten, werden hier vervollständigt.
[9] Diejenigen unvollständigen Zugriffe, die noch keine Auswirkungen auf die Datenstruktur hatten, werden hier weggelassen.
[10] vgl. Beispiel 2
[11] engl. "Race Conditions"

SATZ 3. *Wenn x aus der Warteschlange entfernt wurde, so gab es für x eine Enqueue–Operation, bei der x in die Warteschlange eingefügt wurde; diese Enqueue–Operation findet entweder vor der Dequeue–Operation von x statt oder dazu nebenläufig.*

BEWEISSKIZZE: Beweis durch Widerspruch: Es wird angenommen, dass die Warteschlange sich nicht derart verhält und dennoch linearisierbar ist. Die Linearisierung der Ablauffolge müsste dann per Definition die Spezifikation der FIFO–Warteschlange einhalten.

Allerdings muss jede Linearisierung der Ablauffolgen die zeitliche Ordnung erhalten und verletzt damit zwangsläufig die Spezifikation. Damit ist die Warteschlange nicht linearisierbar.

2.3 Eigenschaften der Linearisierbarkeit

Ich möchte nun auf die nützlichen Eigenschaften der Linearisierbarkeit, Blockierungsfreiheit und Lokalität, näher eingehen.

2.3.1 Blockierungsfreiheit

Die erste wesentliche Eigenschaft der Linearisierbarkeit ist, dass sie keine Blockade von Prozessen erzwingt. Eine Implementierung kann während eines Zugriffs nebenläufige Zugriffe blockieren, aber dies ist nicht notwendig um Linearisierbarkeit zu erreichen. Des Weiteren ist eine totale[12] Operation immer[13] durchführbar.

Wenn zu einem Zeitpunkt in einer Ablauffolge ein Zugriff, der zu einer totalen Operation gehört, unvollständig ist, so existiert eine Linearisierung der Teilablauffolge bis zu diesem Zeitpunkt, die die fehlende Antwort beinhaltet. Diese Linearisierung enthält keine Zugriffe, die zu diesem Zeitpunkt noch nicht gestartet wurden.

Die Forderung, dass die Operation total ist, ist notwendig: Eine nicht totale Operation, z. B. ein Dequeue auf einer leeren Liste, kann durchaus blockieren, wenn dies erwünscht ist. Die Antwort des Zugriffs verzögert sich dann bis ein weiterer Enqueue stattgefunden hat.

Mit anderen Worten: Ein unvollständiger Zugriff (zu einer totalen Operation) muss zu jedem Zeitpunkt nur schlimmstenfalls auf die Beendigung aller anderen nebenläufigen unvollständigen Zugriffe auf dasselbe Objekt warten.

Damit ist die Ausführungsdauer eines Zugriffes schlimmstenfalls bedingt durch seine eigene Ausführungsdauer und die Ausführungsdauer aller nebenläufigen Zugriffe.

2.3.2 Lokalität

Die andere wesentliche Eigenschaft der Linearisierbarkeit ist ihre Lokalität. Um die Linearisierbarkeit einer Ablauffolge H zu zeigen, ist es hinreichend zu zeigen, dass für jedes Objekt O die Teilablauffolge H|O linearisierbar ist ([2], S. 470f).

[12]Eine totale Operation auf einer Datenstruktur ist eine Operation, die definiert ist für jeden Zustand der Datenstruktur, die, informell gesprochen, immer durchführbar ist.

[13]Sie kann immer gestartet werden, wird jedoch möglicherweise erst nach anderen nebenläufigen Zugriffen bearbeitet.

Dies verringert den Aufwand zum Nachweis der Linearisierbarkeit eines Systems beträchtlich. Wäre Linearisierbarkeit nicht lokal, so müssten zum Nachweis der Linearisierbarkeit sämtliche möglichen Permutationen der Teilablauffolgen H|O auf Linearisierbarkeit überprüft werden. Aufgrund der Lokalität jedoch sind die Teilablauffolgen unabhängig voneinander auf Linearisierbarkeit überprüfbar. Damit ist der Nachweis der Linearisierbarkeit eines Systems von Objekten zurückführbar auf den Nachweis der Linearisierbarkeit der einzelnen Objekte.

2.4 Nachweis der Linearisierbarkeit

Eine bis jetzt unbeantwortet gebliebene Frage ist, wie die Linearisierbarkeit einer Implementierung einer Datenstruktur nachgewiesen werden kann.

2.4.1 Atomare Zustandsänderungen

Ein bekanntes Beispiel für eine mit Compare–And–Swap implementierte linearisierbare FIFO–Warteschlange stammt von Michael und Scott [4].

Die Autoren bringen in ihrem Papier ein einfaches und hinreichendes Argument für Linearisierbarkeit: Da die Operationen auf ihrer Warteschlange mittels atomarer Instruktionen realisiert sind, gibt es einen Augenblick während des Zugriffs, an dem diese ihre Auswirkung zeigen (vgl. [4], §3.2, S.5), also einen Linearisierungspunkt (siehe 2.1.1).

Ein weiteres Beispiel für eine auf diese Art implementierte linearisierbare Datenstruktur ist der im Folgenden vorgestellte Mutex.

Der Mutex hat intern zwei Zustände, “gesperrt” und “nicht gesperrt”, hier durch “l” und “u” bezeichnet. Es gibt drei abstrakte Operationen:

1. unlock: setze den Zustand auf “u”.
2. try_lock: wenn der Zustand “u” ist, setze ihn auf “l” und gib “wahr” zurück; sonst “falsch”.
3. lock: wenn der Zustand “u” ist, setze ihn auf “l”.

Die Implementierung verwendet atomare Compare–And–Swap – Instruktionen und sieht folgendermaßen aus:

- unlock(m):
  CAS(m, “l”, “u”)
- try_lock(m):
  return CAS(m, “u”, “l”)
- lock(m):
  while not try_lock(m) do nothing

Die Implementierung dieses Mutex ist linearisierbar, da jeder Zugriff seine Wirkung atomar annimmt, also einen Linearisierungspunkt besitzt. Die Linearisierung einer Ablauffolge ist daher eindeutig bestimmt, indem die Zugriffe nach der Reihenfolge ihrer Linearisierungspunkte sortiert werden.

Alle totalen Operationen auf dem Mutex sind offensichtlich nicht blockierend. Die "lock"–Operation kann zwar blockieren, dies ist jedoch kein Widerspruch zur Blockierungsfreiheit des Mutex. Die "lock"–Operation ist nämlich keine totale Operation, deswegen darf sie blockieren.

Wenn eine nicht totale Operation gestartet wird und das Objekt in einem Zustand ist, bei dem das Ergebnis nicht definiert ist, so gibt es zwei Möglichkeiten zu reagieren: Die Operation liefert eine Fehlermeldung [14] oder sie blockiert solange bis sie nach einer der nächsten Zustandsänderungen definiert ist. Linearisierbarkeit steht mit keiner dieser Vorgehensweisen in Konflikt.

2.4.2 Atomarität durch Locking

Eine einfache aber recht unperformante ([4], §4, S. 6ff) Methode eine Datenstruktur zu linearisieren ist, sie mit einem Mutex zu versehen und jeden Zugriff mit einem Lock auf diesem Mutex zu beginnen und mit einem Unlock zu beenden; während der Operation darf der Prozess den Mutex nicht freigeben. Diese Methode kann in dieser Form nur verwendet werden, wenn alle Operationen total sind, da sonst eine Operation auf einen Zustand der Datenstruktur treffen kann, für den sie nicht definiert ist und alle weiteren Zugriffe blockiert.

Die Linearisierung einer bei dieser Implementierung entstehenden Ablauffolge gestaltet sich sehr einfach: Nebenläufige Zugriffe werden linearisiert indem sie nacheinander in der Reihenfolge ausgeführt werden, in der sie den Mutex erhalten haben.

Diese Methode besitzt mehrere Nachteile. Zuerst benötigt sie einen Mutex pro Datenstruktur, was je nach Mutex–Implementierung einen deutlichen Mehrverbrauch an Speicher bewirken kann. Des Weiteren sind Operationen auf einem Mutex im Allgemeinen als Bibliotheksfunktionen implementiert, die schlimmstenfalls sogar Systemaufrufe nach sich ziehen können, was die Ausführungszeit u. U. stark vergrößert. Außerdem verbietet die Verwendung nur eines Mutex die nebenläufige Ausführung von Operationen, die an sich ohne Weiteres nebenläufig ausführbar wären (z. B. lesende Zugriffe[15]).

Interessanterweise ist diese Implementierung der Linearisierbarkeit immer noch nicht blockierend (nach der obigen Definition). Zwar muss ein (zu einer totalen Operation gehörender) Zugriff potentiell auf die Beendigung aller nebenläufigen Zugriffe warten. Es kann jedoch nicht passieren, dass ein Zugriff eine Datenstruktur "sperrt", so dass diese durch einen weiteren, später stattfindenden Zugriff "entsperrt" werden muss, was bei serialisierbaren Datenstrukturen während einer Transaktion passieren kann (siehe 2.5.1).

2.4.3 Eine allgemeine Beweismethode

Eine allgemeinere Methode, die Linearisierbarkeit einer Implementierung einer abstrakten Datenstruktur zu zeigen, wurde von Herlihy und Wing ([2], S. 474ff) eingeführt und eignet sich für komplexere Datenstrukturen, deren Operationen nicht atomar implementierbar sind.

Manche heutige Rechner bieten zwar garantiert atomare Instruktionen wie zum Beispiel "Compare–And–Swap" oder "Load–Linked / Store–Conditional", diese bieten jedoch im Allgemeinen nicht die Möglichkeit, mehr als zwei Datenwörter gleichzeitig zu ändern. Infolgedessen muss eine Implementierung einer Operation unter Umständen in mehrere atomare Instruktionen zerlegt werden.

Die Beweismethode bedient sich des bekannten Konzepts der invarianten Bedingung: Zu jeder atomaren Instruktion die in einem Zugriff stattfindet und jedem Zustand eines Objekts, der vor dieser Instruktion möglich ist und der die invariante Bedingung erfüllt, wird bewiesen, dass die Bedingung auch für den neuen Zustand nach der Instruktion gilt.

Um im sequentiellen Fall nachzuweisen, dass die Implementierung mit der abstrakten Spezifikation zusammenpasst, wird einem internen Zustand eines Objekts ein abstrakter Zustand zugeordnet. Diese Zuordnung ist eindeutig und während eines Zugriffes undefiniert.

Im nebenläufigen Fall ist die Situation nicht so einfach. Zuerst muss der abstrakte Zustand des Objekts zu jedem Zeitpunkt definiert sein, um nebenläufige Zugriffe zu ermöglichen. Überraschenderweise ist es zudem nicht mögich, den abstrakten Zustand eindeutig festzulegen (vgl. [2], S. 476f).

Die Lösung ist, einem internen Zustand mehrere abstrakte Werte zuzuordnen. Ein interner Zustand entspricht dann einer Menge von möglichen abstrakten Zuständen. Damit ist die korrekte, linearisierbare Implementierung nachweisbar: Die Menge von möglichen abstrakten Zuständen muss sich während eines Zugriffs auf eine Art ändern, die die Spezifikation des abstrakten Objekts erfüllt.

Mit dieser Methode ist die Verifikation von Implementierungen, bei denen mehrere atomare Operationen eingesetzt werden müssen machbar.

2.5 Vergleich mit anderen Korrektheitseigenschaften

Wir werden die Linearisierbarkeit jetzt zwei anderen Korrektheitseigenschaften gegenüberstellen, um Vor- und Nachteile aufzuzeigen.

2.5.1 Serialisierbarkeit

Der Begriff der Serialisierbarkeit ([7], S. 274f) kommt aus dem Themengebiet Datenbanken und basiert darauf, mehrere Zugriffe zu sogenannten Transaktionen zusammenzufassen.

Diese Transaktionen stellen atomare Zustandsübergänge des kompletten Systems dar und laufen im Allgemeinen so ab, dass ein Prozess auf ein oder mehrere Objekte lesend zugreift, dann einen neuen Zustand des gesamten Systems berechnet und die nötigen Änderungen dann in ein oder meh-

[14] Dies wäre eine Änderung der Definition der Operation; das Problem wird quasi wegdefiniert.

[15] Üblicherweise werden sogenannte "Reader–Writer–Locks" [6] verwendet, um in dieser Situation wieder Parallelität herzustellen. Genauso sind bei Verwendung von "Bedingungs–Variablen" [5] auch nicht totale Operationen zulässig. Wir betrachten hier trotzdem nur den einfacheren Fall, da diese Optimierungen im Prinzip dieselben Nachteile besitzen.

reren Objekten speichert.

Eine Ablauffolge von Transaktionen ist serialisierbar, wenn sie äquivalent[16] ist zu einer Ablauffolge in der diese Transaktionen sequentiell ausgeführt werden.

Ein großer Vorteil der Serialisierbarkeit ist, dass der globale Zustand aller Objekte sehr leicht zu überblicken bzw. zu kontrollieren ist. Alle Änderungen des globalen Zustands werden durch die möglichen Transaktionen definiert. Wenn nun eine Eigenschaft des globalen Zustands zu beweisen ist, so ist "nur" nachzuweisen, dass diese Eigenschaft von allen möglichen Transaktionen erhalten bleibt (und zu Beginn gilt). Dies ist insbesondere hilfreich wenn die Datenbank komplexe Konsistenzbedingungen erfüllen muss.

Im Gegensatz dazu ist es bei einem linearisierbaren System schwierig, Eigenschaften des globalen Zustands nachzuweisen. Dies liegt vor allem daran, dass der globale Zustand schlimmstenfalls aus dem kartesischen Produkt der Zustände aller Objekte besteht, d. h. jede beliebige Mischung von Objektzuständen ist prinzipiell möglich. Das ist ein enormer Nachteil wenn man komplexe Konsistenzanforderungen hat.

Allerdings besitzt Serialisierbarkeit auch Nachteile gegenüber Linearisierbarkeit. Ein wesentliches Problem mit serialisierbaren Systemen ist die Möglichkeit von unvereinbaren Transaktionen.

Stellen wir uns zum Beispiel zwei Transaktionen vor, die auf den Objekten X und Y arbeiten und den Code "Y := X + 1" bzw. "X := Y + 1" implementieren. Diese Transaktionen zerfallen in je zwei Zugriffe: "Lese X (bzw. Y)" und "Schreibe Y (bzw. X)".

Es ist klar, dass diese zwei Transaktionen nicht nebenläufig ablaufen dürfen[17]; jedes System, das Serialisierbarkeit implementiert muss dies verhindern.

Daraus folgt, dass während einer Transaktion eine zweite Transaktion möglicherweise nicht gestartet werden kann: Der Prozess der die zweite Transaktion starten will ist blockiert. Da die Berechnungen die ein Prozess während seiner Transaktion ausführt sehr viel länger dauern können als die simple Addition in unserem Beispiel, ist der Prozess der die zweite Transaktion durchführen will potentiell sehr lange blockiert.

Eine totale Operation auf einem linearisierbaren Objekt jedoch muss allerhöchstens auf nebenläufige Operationen warten; ein Objekt kann nicht zwischen zwei sequentiellen Zugriffen blockiert sein.

Dies ermöglicht einen wesentlich höheren Grad der Gleichzeitigkeit als bei serialisierbaren Systemen, was die Performance steigert.

[16]im Sinne von Definition 8 in Sektion 2.1.2

[17]Wenn X und Y auf 2 initialisiert werden, so sind die zwei akzeptablen Zustände nach den zwei Transaktionen (X=3, Y=4) und (X=4, Y=3); würden die Transaktionen parallel ablaufen, so wäre (X=3, Y=3) ein möglicher Endzustand – dies wäre aber nicht serialisierbar.

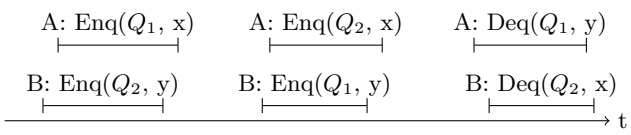

Abbildung 5: Eine nicht sequentiell konsistente Ablauffolge

Solange für unvereinbare Transaktionen nicht eine akzeptable Ausführungszeit im Schlimmstfall garantieren werden kann, ist Linearisierbarkeit daher für Systeme mit Echtzeitanforderungen klar vorzuziehen.

2.5.2 Sequentielle Konsistenz

Eine weitere Korrektheitseigenschaft ist die sogenannte sequentielle Konsistenz ([3]; [7], S. 300f). Eine Ablauffolge ist sequentiell konsistent wenn sie zu einer sequentiellen Ablauffolge äquivalent ist. Die zeitliche Ordnung von nicht nebenläufigen Zugriffen muss dabei nicht erhalten bleiben. Jede linearisierbare Ablauffolge ist auch sequentiell konsistent; jedoch die Ablauffolge in Darstellung 4 ist zwar nicht linearisierbar, aber sequentiell konsistent.

Ein Nachteil der sequentiellen Konsistenz ist, dass sie nicht lokal ist. Hierzu betrachten wir Abbildung 5 (vgl. [2], S. 472).

Die Teilablauffolgen H|Q_1 und H|Q_2 sind sequentiell konsistent (vgl. Abbildung 4); die gesamte Ablauffolge ist jedoch nicht sequentiell konsistent.

Damit ist die sequentielle Konsistenz nicht lokal. Um sie nachzuweisen reicht es also nicht, die Objekte getrennt voneinander zu betrachten[18].

Allerdings besitzt sequentielle Konsistenz Vorteile, zum einen, was die Performanz angeht. Attiya und Welch [1] haben nachgewiesen dass Linearisierbarkeit signifikant höhere Laufzeitkosten hervorruft, wenn die Prozesse voneinander entfernt arbeiten und nicht perfekt synchronisierte Uhren haben, eine Situation, die in verteilten Systemen und NUMA–Architekturen auftritt. Außerdem ist sequentielle Konsistenz leichter nachzuweisen. Dies ist wenig überraschend, da aus Linearisierbarkeit bereits sequentielle Konsistenz folgt.

3. FAZIT

Linearisierbarkeit bietet eine einfache, leicht zu verwendende Möglichkeit, Aussagen über nebenläufigen Programme zu machen. Sie bietet dabei gegenüber anderen Korrektheitseigenschaften Vorteile; ihre Lokalität erleichtert die Verifikation von größeren Systemen, ihre Blockierungsfreiheit hilft bei Echtzeitanforderungen z. B. beim Finden der Höchstzugriffsdauer einer Operation.

Nachteile der Linearisierbarkeit bestehen zum einen darin, dass Aussagen über den globalen Zustand des Systems schwer zu treffen sind. Wenn ein System daher umfangreiche Konsistenzbedingungen erfüllen soll, sind andere Korrektheitseigenschaften interessant.

[18]Es sei denn man weist gleichzeitig Linearisierbarkeit nach.

Zum anderen benötigt die Implementierung der Linearisierbarkeit unter Umständen[19] mehr Resourcen als schwächere Korrektheitseigenschaften[20]; der Nachweis der Linearisierbarkeit einer Implementierung ist ebenfalls aufwändiger als bei z. B. der sequentiellen Konsistenz[21].

4. REFERENZEN

[1] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91 – 122, 1994.

[2] M. P. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463 – 492, 1990.

[3] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690 – 691, Sept. 1979.

[4] M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing (PODC '96)*, 1996. http://www.cs.rochester.edu/u/michael/PODC96.html.

[5] The NetBSD Foundation. *pthread_cond_wait(3)*. Posix Threads Library, nachlesbar unter http://netbsd.gw.com/cgi-bin/man-cgi?pthread_cond_wait+3 (Stand vom Mi. 6. Mai 2009, 1:00).

[6] The NetBSD Foundation. *rwlock(9)*. Reader / Writer lock - Funktionen im NetBSD - Kernel. nachlesbar unter http://netbsd.gw.com/cgi-bin/man-cgi?rwlock+9+NetBSD-5.0 (Stand vom Mi. 6. Mai 2009, 1:00).

[7] A. Tanenbaum and M. van Steen. *Distributed Systems*. Prentice - Hall, Upper Saddle River, NJ, 2002. ISBN 0-13-088893-1.

[19] z. B. in verteilten Systemen, siehe 2.5.2

[20] Der interessierte Leser sei für einen groben Überblick über schwächere Korrektheitseigenschaften an das Buch von Tanenbaum über verteilte Systeme [7] verwiesen

[21] siehe 2.5.2