10. Linearisierbarkeit

Ziel: Formale Spezifikation nebenläufiger Programme, da sonst keine Verifikation möglich.

Motivierendes Beispiel:

```
class LQueue {
    int head, tail = 0;
    int[] elems = new int[77];

    bool enqueue (int x) {
        lock();
        bool flag = false;
        if (tail - head != 77) {
            elems[tail % 77] = x;
            tail ++;
            flag = true;
        }
        unlock();
        return flag;
    }
```

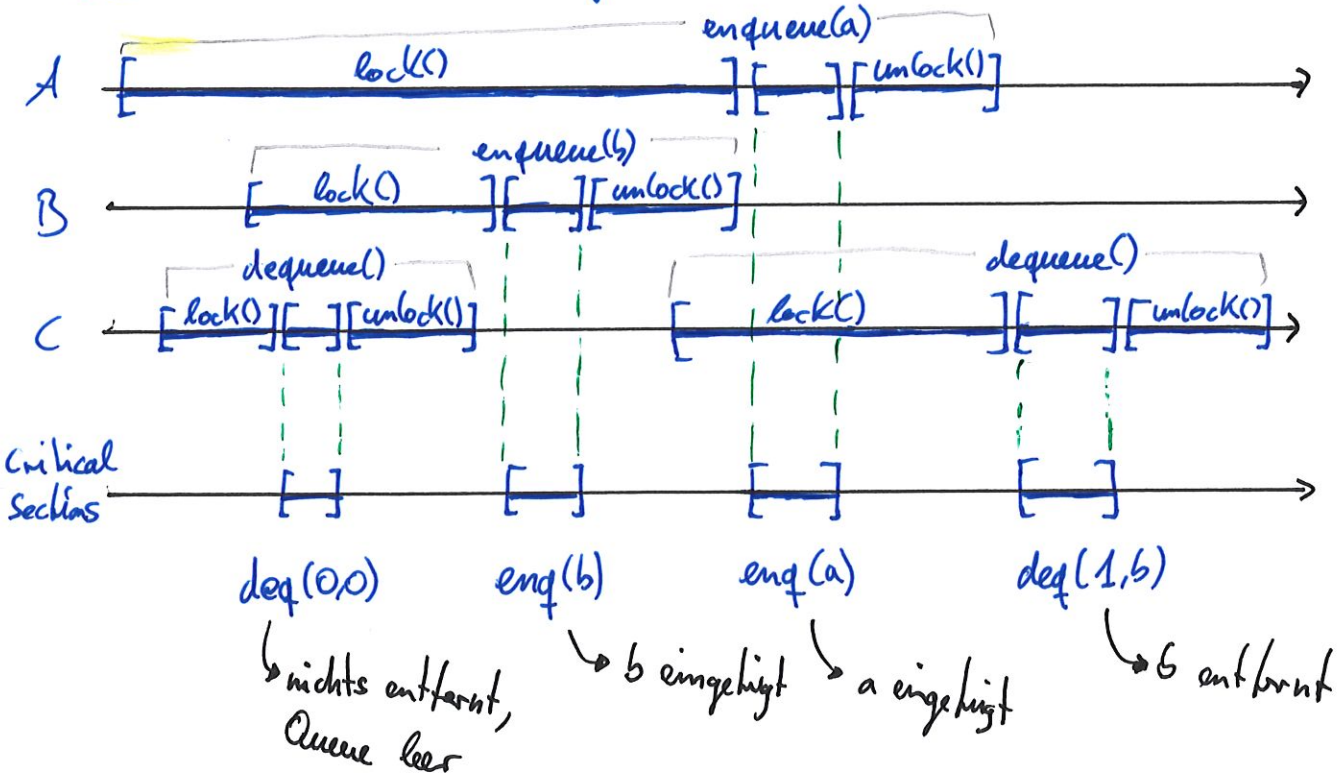
```
    bool x int dequeue () {
        lock();
        bool flag = false;
        int x = 0;
        if (tail != head) {
            x = elems[head % 77];
            head ++;
            flag = true;
        }
        unlock();
        return flag, x;
    }
}
```

Beobachtung: alle Zugriffe werden durch ein Lock geschützt.

↳ die Methoden arbeiten quasi sequentiell

↳ es handelt sich um eine korrekte FIFO-Queue!

Die Korrektheit wird an folgender Illustration klar.



Die Critical Sections der Threads A,B,C überlappen sich nicht, da sie von einem einzelnen Lock beschützt werden.

Der Code, der während der Critical Sections ausgeführt wird, ist der einer sequentiellen FIFO-Queue.

↳ Die Rückgabewerte von dequeue() in Thread C sind die erwarteten Werte: "leer" am Anfang, b am Ende.

↳ Sofern alle Methoden über ein Lock synchronisiert werden, folgt Korrektheit des nebenläufigen Programms aus der Korrektheit des sequentiellen Programms.

Problem: Jede Methode zu locken ist ineffizient und macht Nebenläufigkeit überflüssig.

Wollen: - Tatsächlich nebenläufige Programme spezifizieren / verifizieren.
- Spezifikation über sequentielle Programme wünschenswert, da einfacher.

Ansatz: Linearisierbarkeit

↳ Überführe nebenläufige Ausführungen in sequentielle Ausführungen.

↳ Prüfe sequentielle Ausführung auf Korrektheit.

Sequentielle Spezifikation & Verifikation:

Die Verifikationsaufgabe ist typischerweise einfacher für sequentielle Programme.

↳ haben das bereits für Beweissysteme gesehen:

Separation Logic vs CSL Rely/Guarantee
Hoare-Logic RGSep

Zur Spezifikation bieten sich verschiedene Techniken an, zum Beispiel:

① Automaten (Petri Netze):

Wir können Automaten definieren, die obige Sequenzen wie

deq(0,0). enq(b). enq(a). deq(1,b)

akzeptiert und Sequenzen wie

deq(0,0). enq(b). enq(a). deq(1,a)

als falsch ablehnt.

Vorteil: Dieser Ansatz kann sehr kompakte und elegante Spezifikationen liefern.

Nachteil: unsere bisherigen Verifikationstechniken übertragen sich nicht sofort auf Automaten.

② Programme.

Wir können Programme wählen, deren sequenzielles Verhalten als Spezifikation verwendet wird, also als Korrekt gilt.

Beispiel:

Obige LQueue kann durch folgende sequentielle Queue spezifiziert werden:

```
class Queue {
    int head, tail = 0;
    int[] elems = new int[77];

    bool enq(int x) {
        if (tail - head = 77) return false;
        elems[tail % 77] = x;
        tail++;
        return true;
    }
```

```
    bool x int deq() {
        if (tail == head) return false, 0;
        x = elems[head % 77];
        head++;
        return true, x;
    }
```

- Die Spezifikation Queue ist sehr ähnlich zu LQueue, weshalb die Korrektheit von LQueue leicht zu folgen ist.

- Beachte aber, dass Queue kein korrektes Verhalten zeigt, wenn mehrere Threads nebenläufig arbeit.
  ↳ Queue ist (eigentlich) eine sequentielle Spezifikation!

Bemerkung: sequentielle Spezifikationen (unabhängig vom Typ) für Datenstrukturen nennt man Abstract Data Types (ADT).

Im Folgenden gehen wir nun davon aus, dass wir eine sequentielle Spezifikation gegeben haben und für einzelne Programmausführungen prüfen können, ob diese die Spezifikation erfüllen.

Nebenläufige Spezifikation:

Wir definieren nun Linearisierbarkeit. [Herlihy & Wing '90]

Intuitiv verlangt Linearisierbarkeit, dass

1) jeder Methodenaufruf atomar, also instantan, erscheint

und

2) die resultierende sequentielle Ausführung der Spezifikation genügt.

Um diese Eigenschaften formal auszuarbeiten, benötigen wir einige Definitionen.

Definition:

- Wir modellieren Programmausführungen als Historien.
- Eine History ist eine Folge aus Ereignissen (Events).
- Ein Ereignis ist entweder ein Methodenaufruf (invocation) oder ein Methodenende (response).

Methodenaufrufe haben die Form: $\langle A, x.m(\bar{a}) \rangle$

mit:
- A ist ausführender Thread
- x ist ein Objekt, z.B. LQueue
- m ist eine Methode, z.B. enq
- $\bar{a} = a_1, \ldots, a_n$ ist eine Liste an Argumenten

Methodenenden haben die Form $\langle A, x : \bar{r} \rangle$

wobei $\bar{r} = r_1, \ldots, r_n$ eine Liste an Rückgabewerten ist.

Beispiel:

Die History zum obigen Ablauf von LQueue ist:

$$H = \langle A, q.enq(a)\rangle . \langle C, q.deq()\rangle . \langle B, q.enq(b)\rangle . \langle C, q:0,0\rangle .$$
$$\langle C, q.deq()\rangle . \langle B, q:1\rangle . \langle A, q:1\rangle . \langle C, q:1,b\rangle$$

- Ein Methodenende $\langle A; x:\bar{r}\rangle$ passt zu einem Methodenaufruf $\langle B, y:m(\bar{a})\rangle$ falls $A = B$ und $x = y$ gilt.
- Ein Methodenaufruf ist offen in H, falls in H kein passendes Methodenende folgt.

Beispiel: (Fortsetzung)

- $\langle A, q:1\rangle$ passt zu $\langle A, q.enq(a)\rangle$
- in $H' = \langle D, q.enq(d)\rangle . H$ ist $\langle D, q.enq(a)\rangle$ offen
- in H ist $\langle C, q.deq()\rangle$ nicht offen

- $H.H'$ ist eine Erweiterung von H, falls H' nur aus Methodenenden besteht, die zu offenen Methodenaufrufen aus H passen. (H' kann auch leer sein.)
- complete(H) ist die Teilsequenz von H aus der genau die offenen Methodenaufrufe entfernt wurden.

Beispiel: (Fortsetzung)

- $H'.\langle D, q:1\rangle$ ist eine Erweiterung von H'
- complete(H') = complete(H)

- H ist sequentiell, falls jeder Methodenaufruf sofort von einem passenden Methodenende gefolgt wird.

- Eine Projektion von H auf Thread A ist die Teilsequenz von H, die genau aus den Ereignissen von A besteht.
  Wir schreiben: $H|A$.

  Ähnlich definiert ist die Projektion von H auf Objekt x, $H|x$.

Beispiel: (Fortsetzung)

- $H|A = \langle A, q.enq(a)\rangle . \langle A, q:1\rangle$
- $H|C = \langle C, q.deq()\rangle . \langle C, q:0,0\rangle . \langle C, q.deq()\rangle . \langle C, q:1,b\rangle$
- $H|A$ ist sequentiell

⊣

Bemerkung: Wir nehmen an, dass immer $H|A$ sequentiell ist.
Man sagt auch, dass H wohlgeformt ist falls $H|A$ sequentiell ist für alle Threads A.

- Zwei Histories H und H' sind äquivalent, falls $H|A = H'|A$ für alle Threads A gilt.

- Eine sequentielle History H ist gültig, falls für alle Objekte x die History $H|x$ der Spezifikation für x genügt.

Beispiel: (Fortsetzung)

- $H'' = \langle C, q.deq()\rangle . \langle C, q:0,0\rangle . \langle B, q.enq(b)\rangle . \langle B, q:1\rangle .$
  $\langle A, q.enq(a)\rangle . \langle A, q:1\rangle . \langle C, q.deq()\rangle . \langle C, q:1,b\rangle$ ist gültig.

- H und H'' sind äquivalent.

⊣

- Definiere $m_0 \prec_H m_1$ für History H und Methodenaufrufe m_0, m_1 falls das erste auf m_0 passende Methodenende in H vor m_1 liegt. D.h. m_0 wird beendet bevor m_1 aufgerufen wird. Die partielle Relation $\prec_H$ wird auch das Echtzeit-Verhalten genannt. Falls H sequentiell ist, dann ist $\prec_H$ eine totale Ordnung.

Mit den obigen Definitionen können wir nun den Begriff der Linearisierbarkeit konkretisieren. Die Grundidee ist, dass nebenläufige (nicht-sequentielle) Histories H auf eine sequentielle History S abgebildet werden, sodass gilt:

1) H und S sind äquivalent,
2) H und S weisen das selbe Echtzeitverhalten auf, und
3) S ist ~~valide~~ gültig.

Es ist zu beachten, dass in S keine Methodenaufrufe offen sind (da sequentiell). In H können Aufrufe dagegen offen sein. Diese offenen Aufrufe können:

- noch keine Änderungen vorgenommen haben (pure)

oder

- bereits Änderungen vorgenommen haben (impure / effectful).

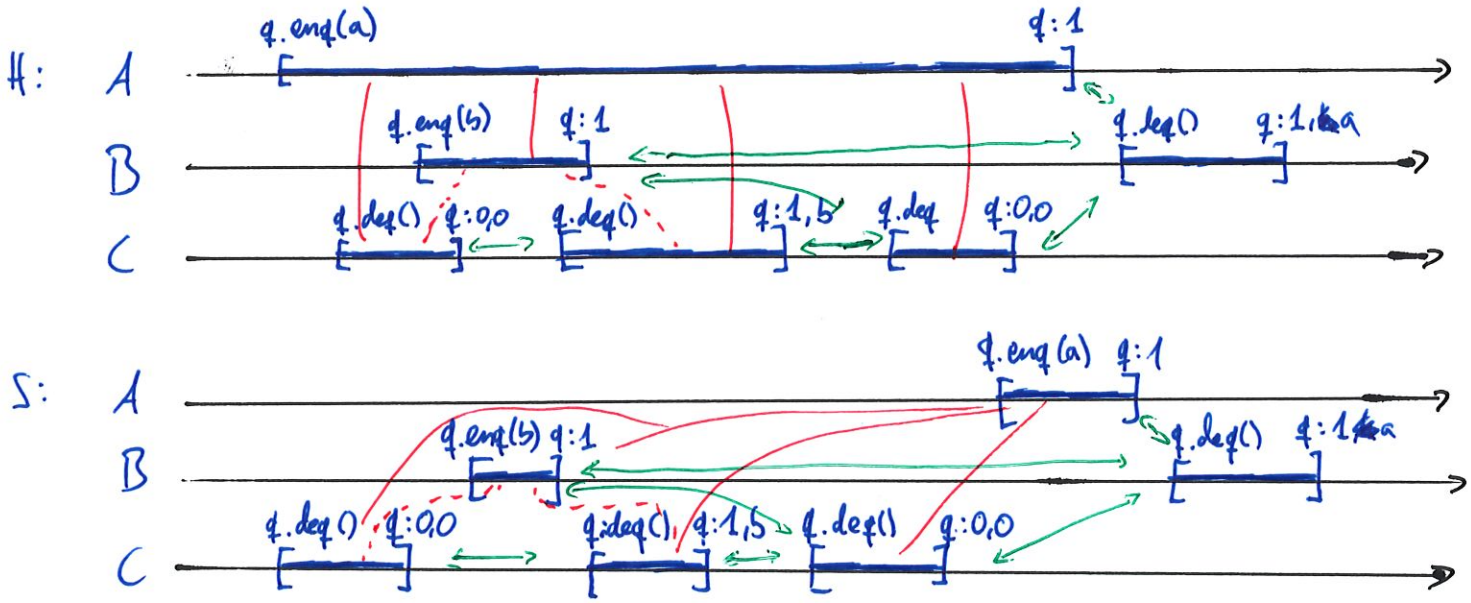
Um Äquivalenz mit S herzustellen entfernen wir diejenigen offenen Aufrufe, die noch keine Änderung vorgenommen haben.
Da offene Aufrufe, die Änderungen vorgenommen haben, die folgenden Ereignisse beeinflusst haben / haben könnten, können wir diese nicht einfach entfernen. Stattdessen erweitern wir H um entsprechende Methodenenden.

Definition: Linearisierbarkeit

Eine History H ist **linearisierbar**, falls eine Erweiterung H' von H sowie eine sequentielle, ~~legale~~ gültige History S existieren mit:

- complete(H') ist äquivalent zu S und
- für alle Methodenaufrufe m_0, m_1 gilt: $m_0 \, t_H \, m_1 \Rightarrow m_0 \, t_S \, m_1$ $(t_H \subseteq t_S)$ ■

Beispiel:



Beobachtung:

- S erhält das Echtzeitverhalten von H: z.B. wird das enq von A vor den deq von B ausgeführt.

- Wir dürfen überlappende Aufrufe bel. umsortieren.

- Die Abbildung auf sequentielle Histories erlaubt es dem Benutzer eines linearisierbaren Objekts anzunehmen, dass die Aufrufe von Methoden des Objekts atomar ausgeführt werden.
  ⇝ Benutzer müssen sich nicht um Nebenläufigkeit kümmern.

Bemerkung: zur Praxisnähe

Frage: Ist die Abbildung auf sequentielle Historien, die Linearisierbarkeit vornimmt, praktikabel?

Antwort: Ja, sie ist sogar natürlich.

Tatsächlich werden API-Dokumentationen von Programmiersprachen, wie z.B. Java, typischerweise pro Methode angegeben und das in folgender Form:

> "Falls vor der Ausführung der Methode m ... gilt, dann gilt nach der Ausführung von m ... oder es wird Fehler ... erzeugt."

Diese ist der Spezifikation entspricht Hoare-Tripeln wie wir sie kennen.
Wir wissen auch, das zur Spezifikation von nebenläufigen Programmen wesentlich komplexere Ansätze nötig werden: CSL, Rely/Guarantee, RGSep.

Dennoch werden auch für nebenläufige Programme Hoare-artige Spezifikationen benutzt. Betrachte das Beispiel ConcurrentLinkedQueue aus Java.
Die Methoden enq und deq (add und poll) erwähnen keine Operationen anderer Threads. Wir wissen aber, dass wir diese für eine RGSep-Spezifikation brauchen und dass Hoare-Logik nicht mit nebenläufigen Programmen (gut) umgehen kann.

Dennoch werden Hoare-artige API-Dokumentationen benutzt.
Sie sind einfach für Benutzer zu verstehen.
Sie sind korrekt, weil Linearisierbarkeit (und ähnliche Korrektheitseigenschaften) eine Abbildung auf sequentielle Historien vornehmen.

Neben den Vorteilen in der Praxis, hat Linearisierbarkeit auch eine grundlegende Eigenschaft, die der Verifikation zu Gute kommt, nämlich: Kompositionalität.

Kompositionalität:

Um den Nachweis der Linearisierbarkeit zu führen, genügt es jedes Objekt als linearisierbar nachzuweisen.

Theorem:

H ist linearisierbar gdw. $H|x$ ist linearisierbar für alle Objekte x □

Hier gehen wir davon aus, dass Spezifikation Prefix-abgeschlossen sind, also: falls $S.S'$ gültig ist, so ist auch S gültig.

Beweis „⇐": // für Verifikation interessante Richtung

Schreibe $lin(H,S)$ falls S die gültige sequentielle History ist, die die Linearisierbarkeit von H bezeugt.

Wir zeigen die Behauptung für zwei Objekte x, y.

Genauer zeigen wir:

$$\forall H, S_x, S_y.\ lin(H|x, S_x) \wedge lin(H|y, S_y)$$
$$\Longrightarrow \exists S.\ lin(H,S) \wedge S|x = S_x \wedge S|y = S_y$$

Wir gehen per Induktion nach der Anzahl $\#(H)$ an Methodenaufrufen in H vor.

IA: $\#(H) \leq 1$. OBdA gilt $H|x = H$ und $H|y = \varepsilon$. Letzteres gibt $S_y = \varepsilon$.
Wähle $S = S_x$. Es gilt: $lin(H,S)$, $S|x = S_x$, $S|y = S_y$ ✓

IV: Beh. gilt für $\#(H) \leq n$.

IS: $n \to n+1$

- Seien H, S_x, S_y gegeben mit $lin(H|x, S_x)$ und $lin(H|y, S_y)$.
  Es ex. R_x mit: • $H|x . R_x$ ist eine Erweiterung von $H|x$
  • $lin(complete(H|x . R_x), S_x)$.
  Ähnlich für R_y.
- Konstruiere $H' = complete(H . R_x . R_y)$.
- Sei m_x maximal bzgl. t_{S_x} und m_y maximal bzgl. t_{S_y}.
  OBdA ist m_x maximal bzgl. t_H.
- Betrachte Dekomposition: $H' = H_1 . m_x . H_2 . r_x . H_3$,
  wobei r_x das erste zu m_x passende Methodenende ist.
- Konstruiere $G = H_1 . H_2 . H_3$, also H' ohne Teilwort $/$ Ende m_x / r_x.

- Es gilt: H' ist äquivalent zu $G.m_x.r_x$.
  Um das zu erkennen, seien m_x, r_x Ereignisse von Thread A und Thread $B \neq A$ bel.
  Wir erhalten:
  - $H'|B = H_1|B . H_2|B . H_3|B = G|B = G.m_x.r_x|B$
  - $H'|A = H_1|A . m_x . H_2|A . r_x . H_3|A$
    $\overset{\circledast}{=} H_1|A . m_x . r_x \overset{\circledast}{=} H_1|A . H_2|A . H_3|A . m_x . r_x = G.m_x.r_x|A$

    zu $\circledast$:
    - $H_2|A = \varepsilon$, weil H' und damit H sonst nicht wohlgeformt
      ↳ ein Ereignis von A in H_2 widerspricht H/H#sequentiell
    - $H_3|A = \varepsilon$, weil wenn H_3 ein Ereignis enthält, dann muss es ein Methodenaufruf in H_3 gehen; dieser aber wäre größer bzgl. $\downarrow_H$, was der Wahl m_x widerspricht

- Es gilt: lin $(G|_Y, S_Y)$ weil $G|_Y = H'|_Y = \text{comple}(H|_Y . R_Y)$.

- Da m_x maximal bzgl. $\downarrow_{S_x}$ und S_x sequentiell, gilt: $S_x = S'.m_x.r_x$

- Es gilt: lin $(G|_X, S')$ weil:
  - $G.m_x.r_x|_X$ äquivalent zu $H'|_X = \text{comple}(H|_X . R_X)$ äquivalent zu $S_x = S'.m_x.r_x$
    Also $G|_X$ äquivalent zu S'.
  - Betrachte m_0, m_1 mit $m_0 \downarrow_{G|_X} m_1$. Nach Konstruktion G: $m_0 \neq m_x.r_x \neq m_1$.
    Damit gilt: $m_0 \downarrow_{H'|_X} m_1$. Also $m_0 \downarrow_{S_x} m_1$ und $m_0 \downarrow_{S'} m_1$.
  - S' sequentiell, da S_x sequentiell.
  - S' gültig, da S_x gültig.

- Wende IV auf G, S', S_Y an. Wir erhalten S^* mit:
  - lin (G, S)
  - $S|_X = S'$
  - $S|_Y = S_Y$

- Es gilt: $- S.m_x.r_x \mid x = S\mid x.m_x.r_x = S'.m_x.r_x = S_x$

  $- S.m_x.r_x \mid y = S\mid y = S_y$

- Verbleibt zu zeigen: $lin(H, S.m_x.r_x)$. Dies gilt, weil:
  - complete $(H') = H'$ und H' ist Erweiterung von H.
  - H' ist äquivalent zu $G.m_x.r_x$ ist äquivalent zu $S.m_x.r_x$, weil G äquivalent zu S gilt nach $lin(G,S)$
  - Betrachte $m_0 \; t_H \; m_1$. Nach Konstruktion, $m_0 \; t_{H'} \; m_1$. Da m_x maximal bzgl t_H, ist $m_x \neq m_0 \neq r_x$. Damit erhalten wir $m_0 \; t_{G.m_x.r_x} \; m_1$. Nach $lin(G,S)$ dann $m_0 \; t_{S.m_x.r_x} \; m_1$.
  - $S.m_x.r_x$ ist sequentiell, da S sequentiell nach $lin(G,S)$ Beachte, dass in S keine offenen Methodenaufrufe vorkommen.
  - $S.m_x.r_x$ ist gültig, da S_x und S_y gültig sind.

■