2.7 Eine Anwendung: Verifikation

Ein abschließendes ausführliches Beispiel:

- Wir betrachten zwei Prozesse P_1 und P_2, die auf eine gemeinsame Ressource zugreifen wollen.
- Jeder Prozeß hat einen sogenannten kritischen Bereich, in dem auf die Ressource zugegriffen wird. Es darf sich also jeweils nur ein Prozeß im kritischen Bereich befinden.
- Es stehen gemeinsame Variable zur Verfügung, über die sich die Prozesse synchronisieren können. Diese Variablen sind jedoch keine Semaphore, d.h. eine atomare Operation, bei der gleichzeitig gelesen und geschrieben wird, ist nicht möglich.

Wir möchten zeigen, daß der wechselseitige Ausschluß gewährleistet ist und daß gewisse Fairneßbedingungen (jeder Prozeß kommt irgendwann an die Reihe) eingehalten werden.

Was hat das mit formalen Sprachen zu tun?

- Jeder Ablauf eines Prozesses ist ein Wort, die Menge der Abläufe ist also eine Sprache.
- Ebenso ist die Menge der Abläufe des Gesamtsystems *Sys* eine Sprache L_{Sys}.
- Und auch die Menge der erlaubten bzw. verbotenen Abläufe ist eine Sprache L_{Spec}.

Damit ist also das Inklusionsproblem „$L_{Sys} \subseteq L_{Spec}$?“ bzw. das Disjunktheitsproblem „$L_{Sys} \cap L_{Spec} = \emptyset$?“ zu lösen.

Wir haben gesehen: Sind beide Sprachen regulär, so ist dies algorithmisch machbar!

2.7.1 Erster Versuch:

Die Prozesse P_1, P_2 verwenden eine gemeinsame Boolesche Variable `f`, die mit `false` initialisiert wird.

Programmcode für P_1, P_2

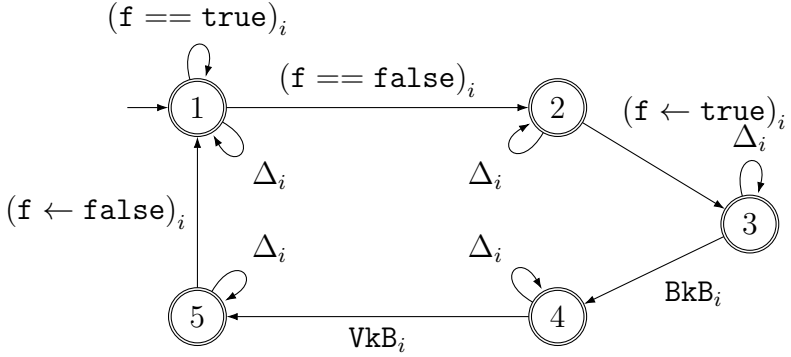
```
   while true do
1: if (f == false) then do
     begin
2:     f ← true
3:     [Betrete kritischen Bereich]
4:     [Verlasse kritischen Bereich]
5:     f ← false
     endif
   enddo
```

Das folgende Alphabet Σ besteht aus den Programm-Befehlen und den Abfragen der Booleschen Variablen:

$$\begin{aligned}
&\{(\texttt{f} \leftarrow \texttt{true})_i, (\texttt{f} \leftarrow \texttt{false})_i\} \\
\cup\ &\{\texttt{BkB}_i, \texttt{VkB}_i \mid i \in \{1,2\}\} \\
&\qquad \text{(Prozeß } i \text{ betritt/verläßt kritischen Bereich)} \\
&\{(\texttt{f} == \texttt{false})_i, (\texttt{f} == \texttt{true})_i \mid i \in \{1,2\}\} \\
&\qquad \text{(Synchronisation von Prozeß } i \text{ mit Variable } \texttt{f})
\end{aligned}$$

Der Index $i \in \{1,2\}$ gibt an, ob die jeweilige Aktion vom ersten oder vom zweiten Prozeß ausgeführt wird.

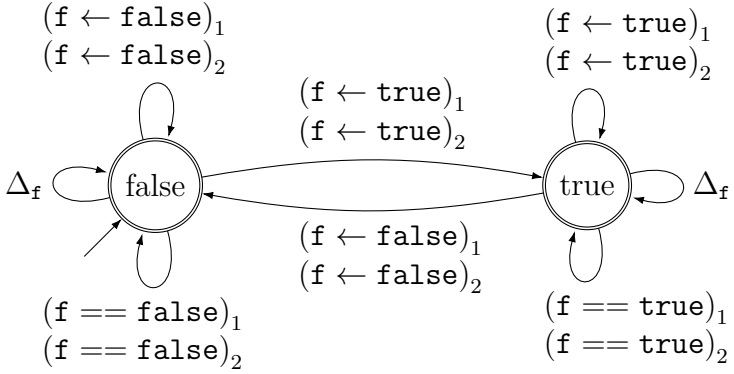
Beschreibung der Abläufe des Prozesses i als NFA P_i:
(mit $j = 3-i$ und $\Delta_i = \{(\texttt{f} \leftarrow \texttt{true})_j, (\texttt{f} \leftarrow \texttt{false})_j, (\texttt{f} == \texttt{true})_j, (\texttt{f} == \texttt{false})_j, \texttt{BkB}_j, \texttt{VkB}_j\}$)



Bemerkungen:

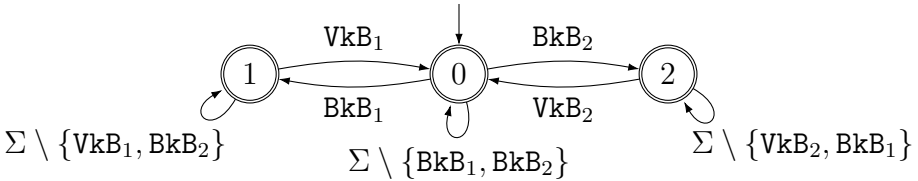
- Zustände 1, 2, 3, 4, 5 entsprechen den entsprechend markierten Programmzeilen
- Bedeutung der Schleifen mit Alphabetsymbolen aus Δ_i: Der Prozeß i interessiert sich nicht für die Aktionen des anderen Prozesses, ändert bei diesen also seinen Zustand nicht. Sie werden also einfach „mitgehört“ und „ignoriert“.

Beschreibung der Booleschen Variable `f` durch einen NFA F:
(mit $\Delta_{\mathtt{f}} = \{\mathtt{BkB}_1, \mathtt{VkB}_1, \mathtt{BkB}_2, \mathtt{VkB}_2\}$):



Die Sprache aller Abläufe des Gesamtsystems ist $L(P_1) \cap L(P_2) \cap L(F)$.

Der NFA *WA*, der diejenigen Abläufe beschreibt, die den wechselseitigen Ausschluß erfüllen (höchstens ein Prozeß ist im kritischen Bereich) sieht folgendermaßen aus:



Damit ist $L(P_1) \cap L(P_2) \cap L(F) \subseteq L(WA)$ zu zeigen.

Es stellt sich heraus, daß $L(P_1) \cap L(P_2) \cap L(F) \subseteq L(WA)$ nicht gilt, d.h. daß unsere Implementierung den gegenseitigen Ausschluß nicht sichert.

Die meisten Werkzeuge, die das Inklusionsproblem automatisch lösen, liefern im Mißerfolgsfall ein Gegenbeispiel. In unserem Fall ist dies z.B. das Wort

$$(\texttt{f} == \texttt{false})_2\ (\texttt{f} == \texttt{false})_1\ (\texttt{f} \leftarrow \texttt{true})_2\ \texttt{BkB}_2\ (\texttt{f} \leftarrow \texttt{true})_1\ \texttt{BkB}_1.$$

Daraus kann ein Grund für die Verletzung des wechselseitigen Ausschlusses abgelesen werden: Die beiden Prozesse können nacheinander die Variable auslesen, anschließend setzen beide die Variable und betreten den kritischen Bereich.

Jedenfalls ist der Algorithmus unbrauchbar!

2.7.2 Zweiter Versuch:

Wir betrachten nun das Verfahren zum wechselseitigen Ausschluß von Leslie Lamport (1978).
Dabei betrachten wir zwei Prozesse P_1 und P_2 mit unterschiedlichem Programmcode und zwei Boolesche Variable `f1` und `f2` (initialisiert mit `false`).

Prozeß P_1

```
  while true do
1:  f1 ← true;                (#)
2:  while (f2 == true)
      do skip enddo;
3:  [Betrete krit. Bereich];
4:  [Verlasse krit. Bereich];
5:  f1 ← false
  enddo;
```

`skip`: Null-Operation
(hat keine Auswirkungen)

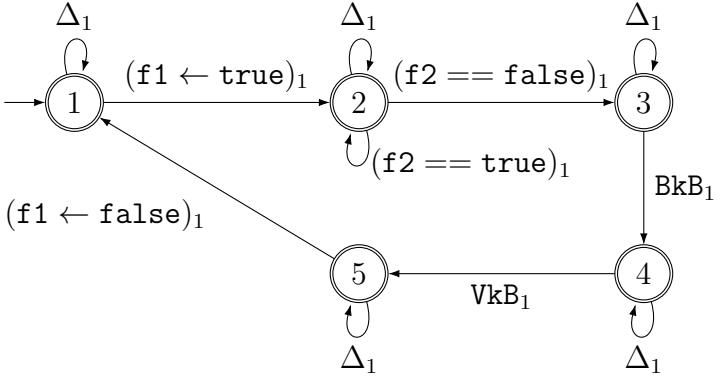
Prozeß P_2

```
  while true do
1:  f2 ← true;                (#)
2:  if (f1 == true) then do
3:    f2 ← false;
4:    while (f1 == true)
        do skip enddo;
      goto 1
    endif;
5:  [Betrete krit. Bereich];
6:  [Verlasse krit. Bereich];
7:  f2 ← false
  enddo;
```

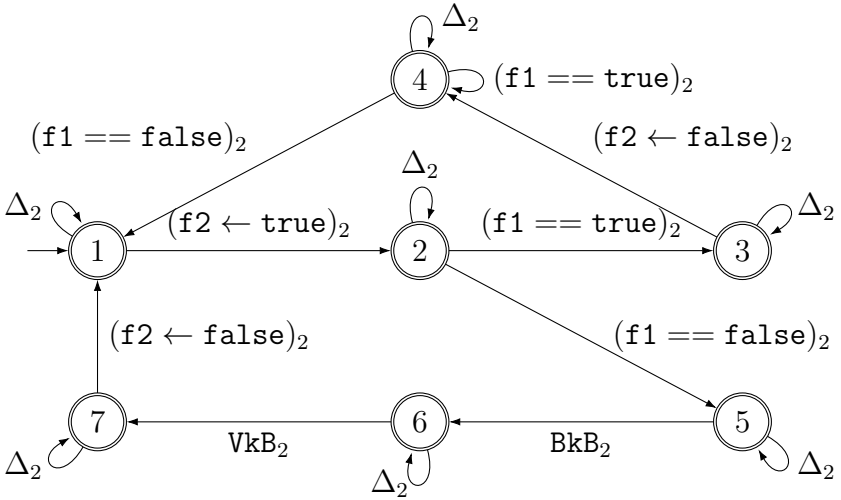
In diesem Fall betrachten wir folgendes Alphabet Σ:

$$\begin{aligned}\Sigma \;=\; \{&(\texttt{f1} \leftarrow \texttt{true})_1, (\texttt{f1} \leftarrow \texttt{false})_1, (\texttt{f1} == \texttt{true})_2, (\texttt{f1} == \texttt{false})_2,\\ &(\texttt{f2} \leftarrow \texttt{true})_2, (\texttt{f2} \leftarrow \texttt{false})_2, (\texttt{f2} == \texttt{true})_1, (\texttt{f2} == \texttt{false})_1,\\ &\texttt{BkB}_1, \texttt{VkB}_1, \texttt{BkB}_2, \texttt{VkB}_2\}.\end{aligned}$$

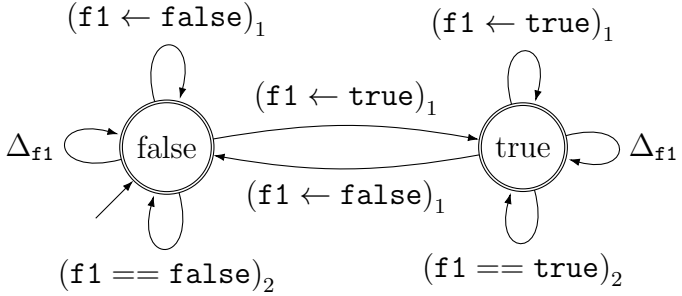
NFA für den Prozeß P_1:
(mit $\Delta_1 = \{(\texttt{f2} \leftarrow \texttt{true})_2, (\texttt{f2} \leftarrow \texttt{false})_2, (\texttt{f1} == \texttt{true})_2, (\texttt{f1} == \texttt{false})_2, \texttt{BkB}_2, \texttt{VkB}_2\}$):



NFA für den Prozeß P_2:
(mit $\Delta_2 = \{(\texttt{f1} \leftarrow \texttt{true})_1, (\texttt{f1} \leftarrow \texttt{false})_1, (\texttt{f2} == \texttt{true})_1, (\texttt{f2} == \texttt{false})_1, \texttt{BkB}_1, \texttt{VkB}_1\}$)



NFA V_1 für die Variable `f1`:
(mit $\Delta_{\texttt{f1}} = \{(\texttt{f2} \leftarrow \texttt{true})_2, (\texttt{f2} \leftarrow \texttt{false})_2, (\texttt{f2} == \texttt{true})_1, (\texttt{f2} == \texttt{false})_1, \texttt{BkB}_1, \texttt{BkB}_2, \texttt{VkB}_1, \texttt{VkB}_2\}$)

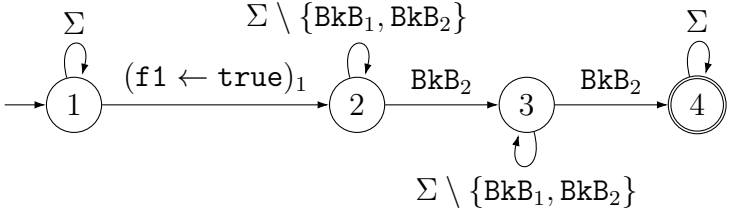


Analog sieht der NFA V_2 für die Variable `f2` aus.
In diesem Fall ist der wechselseitige Ausschluß erfüllt, d.h. es gilt $L(P_1) \cap L(P_2) \cap L(V_1) \cap L(V_2) \subseteq L(WA)$.

Neben dem wechselseitigen Ausschluß soll noch folgende Fairneß-Bedingung für jeden Prozeß i überprüft werden:

(F_i) „Sobald Prozeß i seine Bereitschaft bekundet hat, den kritischen Bereich zu betreten, indem er die Anweisung `(#)` ausführt, kann der andere Prozeß j nicht zweimal hintereinander den kritischen Bereich betreten, ohne daß Prozeß i zwischendurch den kritischen Bereich betritt."

NFA NF_1 für die Abläufe, die (F_1) *nicht* erfüllen:



Man kann zeigen (bzw. einen Computer zeigen lassen), daß $L(P_1) \cap L(P_2) \cap L(V_1) \cap L(V_2) \cap L(NF_1) = \emptyset$. Damit ist Fairneß also für den Prozeß 1 erfüllt.

Hingegen gilt $L(P_1) \cap L(P_2) \cap L(V_1) \cap L(V_2) \cap L(NF_2) \neq \emptyset$, d.h. Fairneß für den Prozeß 2 ist nicht erfüllt. Die automatische (fehlschlagende) Überprüfung liefert auch gleich ein Gegenbeispiel: Das Wort

$$\begin{array}{llll} (\texttt{f2} \leftarrow \texttt{true})_2 & (\texttt{f1} \leftarrow \texttt{true})_1 & (\texttt{f1} == \texttt{true})_2 & (\texttt{f2} \leftarrow \texttt{false})_2 \\ (\texttt{f2} == \texttt{false})_1 & \texttt{BkB}_1 & \texttt{VkB}_1 & (\texttt{f1} \leftarrow \texttt{false})_1 \quad (\texttt{f1} \leftarrow \texttt{true})_1 \\ (\texttt{f2} == \texttt{false})_1 & \texttt{BkB}_1 & & \end{array}$$

ist im Schnitt enthalten.

Die Interpretation dieses Gegenbeispiels ist Ihnen überlassen.

2.7.3 Zusammenfassung:

- Wir haben mit Hilfe von endlichen Automaten zwei Protokolle modelliert, die wechselseitigen Ausschluß realisieren sollen.
- Mit Hilfe der Lösungsverfahren für das Inklusions- bzw. Disjunktheitsproblem haben wir überprüft, ob diese Protokolle tatsächlich wechselseitigen Ausschluß und Fairneß realisieren.

Das bedeutet: die vorgestellten Verfahren können zur Programmverifikation eingesetzt werden.

Bemerkung: Bei realen Programmen hat man allerdings noch damit zu kämpfen, daß die Zustandsmenge des Programms unendlich ist. Damit wird vieles algorithmisch unlösbar („unentscheidbar“) und muß durch approximative Verfahren gelöst werden (vgl. Master-Vorlesung „Verifikation“).

Zusammenfassung reguläre Sprachen

- äquivalente Beschreibungsformen: DFAs, NFAs, reguläre Ausdrücke
- Abschlußeigenschaften: Vereinigung, Produkt, Iteration, Schnitt, Komplement
- Nicht-Regularitätsbeweise: Pumping-Lemma
- minimale DFAs
- Algorithmen für Wort-, Leerheits-, Endlichkeits-, Disjunktheits-, Inklusions- und Äquivalenzproblem
- Anwendung in Verifikation