

siehe auch Formale Grundlagen 3

Motivation: Automaten für die Modellierung, Spezifikation und Verifikation verwenden!

Definition Ein *Endlicher Automat* $A = (S, I, \Sigma, T, F)$ besteht aus

- Menge von Zuständen S (normalerweise endlich)
- Menge von Initialzuständen $I \subseteq S$
- Eingabe-Alphabet Σ (normalerweise endlich)
- Übergangsrelation $T \subseteq S \times \Sigma \times S$
schreibe $s \xrightarrow{a} s'$ gdw. $(s, a, s') \in T$ gdw. $T(s, a, s')$ "gilt"
- Menge von Finalzuständen $F \subseteq S$

Produkt-Automat

Definition Der Produkt Automat $A = A_1 \times A_2$ von zwei EA A_1 und A_2 mit gemeinsamen Eingabealphabet $\Sigma_1 = \Sigma_2$ hat folgende Komponenten:

$$\begin{aligned}
 S &= S_1 \times S_2 & I &= I_1 \times I_2 \\
 \Sigma &= \Sigma_1 = \Sigma_2 & F &= F_1 \times F_2 \\
 T((s_1, s_2), a, (s'_1, s'_2)) & \text{ gdw. } & T_1(s_1, a, s'_1) & \text{ und } T_2(s_2, a, s'_2)
 \end{aligned}$$

Satz Seien $A, A_1,$ und A_2 wie oben, dann $L(A) = L(A_1) \cap L(A_2)$

Beispiel: Konstruktion eines Automaten Wörter mit Prefix ab und Suffix ba akzeptiert.

(als regulärer Ausdruck: $a \cdot b \cdot \mathbf{1}^* \cap \mathbf{1}^* \cdot b \cdot a,$ wobei $\mathbf{1}$ für alle Buchstaben steht)

Definition Ein EA A akzeptiert ein Wort $w \in \Sigma^*$ gdw. es s_i und a_i gibt mit

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \dots \xrightarrow{a_{n-1}} s_{n-1} \xrightarrow{a_n} s_n,$$

wobei $n \geq 0, s_0 \in I, s_n \in F$ und $w = a_1 \dots a_n$ ($n = 0 \Rightarrow w = \epsilon$).

Definition Die Sprache $L(A)$ von A ist die Menge der Wörter die er akzeptiert.

- Benutze Automaten oder reguläre Sprachen zur Beschreibung von Ereignisströmen!
- "Konformität" der Ereignisströme der Implementierung zu denen der Spezifikation!
- Konformität kann präzisiert werden als Teilmengenbeziehung der Sprachen

Vollständigkeit und Determinismus

Definition Zu $s \in S, a \in \Sigma$ bezeichne $s \xrightarrow{a}$ die Menge der Nachfolger von s definiert als

$$s \xrightarrow{a} = \{s' \in S \mid T(s, a, s')\}$$

Definition Ein EA ist *vollständig* gdw. $|I| > 0$ und $|s \xrightarrow{a}| > 0$ für alle $s \in S$ und $a \in \Sigma$.

Definition ... *deterministisch* gdw. $|I| \leq 1$ und $|s \xrightarrow{a}| \leq 1$ für alle $s \in S$ und $a \in \Sigma$.

Fakt ... *deterministisch und vollständig* gdw. $|I| = 1$ und $|s \xrightarrow{a}| = 1$ für alle $s \in S, a \in \Sigma$.

Definition Der Power-Automat $A = P(A_1)$ eines EA A_1 hat folgende Komponenten

$$S = P(S_1) \quad (P = \text{Potenzmenge}) \quad I = \{I_1\}$$

$$\Sigma = \Sigma_1 \quad F = \{F' \subseteq S \mid F' \cap F_1 \neq \emptyset\}$$

$$T(S', a, S'') \text{ gdw. } S'' = \{s'' \mid \exists s' \in S' \text{ mit } T(s', a, s'')\}$$

Satz A, A_1 wie oben, dann $L(A) = L(A_1)$ und A ist deterministisch und vollständig.

Beispiel: Spam-Filter basierend auf der White-List “abb”, “abba”, und “abacus”!

(Regulärer Ausdruck: “abb” | “abba” | “abacus”)

• Modellierung und Spezifikation mit Automaten:

- Ereigniströme einer Implementierung modelliert durch EA A_1
- Partielle Spezifikation der Ereigniströme als EA A_2

• Konformitäts-Test:

- $L(A_1) \subseteq L(A_2)$
- gdw. $L(A_1) \cap \overline{L(A_2)} = \emptyset$
- gdw. $A_1 \times K(P(A_2))$ keinen erreichbaren Finalzustand hat

• **Beispiel:** Spezifikation $S = (cs|sc|ss)^*$, Implementierung $I = ((s|c)^2)^*$

Definition Der Komplementär-Automat $A = K(A_1)$ eines endlichen Automaten A_1 hat dieselben Komponenten wie A_1 , bis auf $F = S \setminus F_1$.

Satz Der Komplementär-Automat $A = K(A_1)$ eines deterministischen und vollständigen Automaten A_1 akzeptiert die komplementäre Sprache $L(A) = \overline{L(A_1)} = \Sigma^* \setminus L(A_1)$.

Beispiel: Spam-Filter basierend auf der Black-List “abb”, “abba”, und “abacus”!

(Regulärer Ausdruck: “abb” | “abba” | “abacus”)

• Temporale Eigenschaften: (**1** steht für ein beliebiges Zeichen)

- jeden dritten Schritt gilt a : $(\mathbf{1} \cdot \mathbf{1} \cdot a)^*$
- genau jeden dritten Schritt gilt a : $(\bar{a} \cdot \bar{a} \cdot a)^*$
- einem a (acknowledge) muss ein r (request) vorangehen: $(\bar{r})^* \cdot a$
- jedem a muss ein r vorangehen: $(\mathbf{1}^* \cdot a)^* \cdot (\bar{r})^* \cdot a$

• Verfeinerung: (am Beispiel Scheduling dreier Prozesse a, b und c)

- abstrakter Round-Robin Scheduler: $(abc \mid acb \mid bac \mid bca \mid cab \mid cba)^*$
- Round-Robin Scheduler mit Vorrang a vor b : $(abc \mid acb \mid cab)^*$
- Round-Robin Scheduler mit Vorrang a vor b und c vor b : $(acb \mid cab)^*$
- deterministischer Round-Robin Scheduler der Implementierung: $(cab)^*$

- ähnliche Vorgehensweise:
 - geg. aussagenlogische Formel f über den booleschen Variablen $V = \{x_1, \dots, x_n\}$
 - Expansion $E(f) \subseteq 2^n$ ist die Menge der erfüllenden Belegungen von f

$$(a_1, \dots, a_n) \in E(f) \text{ gdw. } f[x_1 \mapsto a_1, \dots, x_n \mapsto a_n] = 1$$
 - z.B. $E(f) \neq \emptyset$ gdw. f erfüllbar (engl. satisfiable)
- Modellierung und Spezifikation:
 - f_1 charakterisiert die für die Implementierung möglichen Konfigurationen
 - f_2 beschreibt eine partielle Spezifikation der gültigen Konfigurationen
- Konformitäts-Test: $f_1 \Rightarrow f_2$ gdw. $E(f_1) \subseteq E(f_2)$ gdw. $E(f_1) \cap \overline{E(f_2)} = \emptyset$
 (oder in der Praxis: ... gdw. $f_1 \wedge \neg f_2$ unerfüllbar)

Probleme bei Abstraktion

```

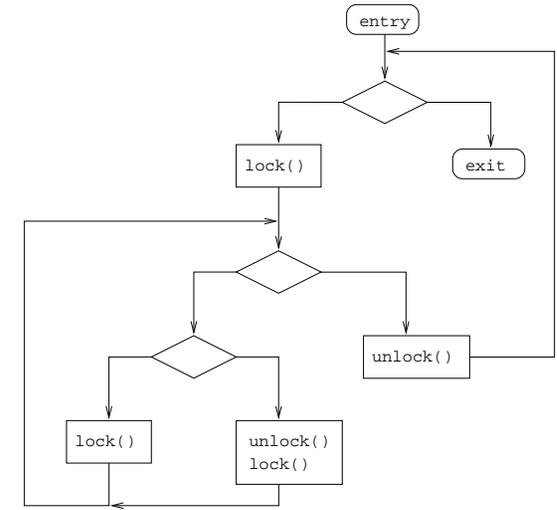
assert (i < n);
lock ();
do {
    ...
    i++;
    if (i >= n)
        unlock ();
    ...
} while (i < n);
    
```

$l \cdot (\varepsilon|u)^*$ verletzt partielle Spezifikation $\overline{\mathbf{1}^* \cdot l \cdot \bar{u}^*}$

(“...” führt weder zu einem lock noch unlock und lässt i und n unberührt)

```

while (...) {
    lock ();
    ...
    while (...) {
        if (...) {
            lock ();
            ...
        } else {
            unlock ();
            ...
            lock ();
        }
        ...
    }
    ...
    unlock ();
}
    
```



$(l \cdot (l | u \cdot l)^* \cdot u)^*$

Predicate Abstraction

verfeinerte Abstraktion zu EA durch Einführung einer Prädikats-Variablen: $b == (i < n)$

<pre> assert (i < n); lock (); do { ... i++; if (i >= n) unlock (); ... } while (i < n); </pre>	abstrahiert zu	<pre> assert (b); lock (); do { ... if (b) b = *; if (!b) unlock (); ... } while (b); </pre>
----------------------------------------------------------------------------------------------------------------------------------------------	----------------	----------------------------------------------------------------------------------------------------------------------------------

$l \cdot \varepsilon^* \cdot u$ erfüllt partielle Spezifikation $\overline{\mathbf{1}^* \cdot l \cdot \bar{u}^*}$

(“...” führt weder zu einem lock noch unlock und lässt i und n unberührt)