4 Model-Checking

Wir schränken uns in der folgenden Darstellung auf endliche Zustandsräume ein, da diese der algorithmischen Analyse (im Rahmen des Model-Checking) zugänglicher sind.

4.1 Automatentheorie und LTL-Model-Checking

Um Informatik-Systeme auf ihre Korrektheit zu prüfen, muss ihr Verhalten mit einer Spezifikation des Systems verglichen werden. Häufig geschieht dies aus Kosten- oder anderen Gründen weniger formal. Ist die Korrektkeit eines Systems besonders wichtig, weil ein Fehlverhalten unverantwortlich oder zu teuer ist, werden Verifikationsmethoden angewandt. Eine davon ist das *Model-Checking*.

4.1.1 Model-Checking bei explizit definierter Spezifikation

Algorithmus 4.1 Model-Checking für FA

`PROCEDURE` $Check(L(TS_{sys}), L(TS_{spec}))$
    `construct complement` $\overline{TS_{spec}}$ `with` $L(\overline{TS_{spec}}) = (A^* \setminus L(TS_{spec}))$
    `construct product` $TS_\cap$ `with` $L(TS_\cap) = L(TS_{sys}) \cap L(\overline{TS_{spec}})$
    `IF` $L(TS_\cap) = \emptyset$ `THEN return true`
    `ELSE return false`
`END PROCEDURE`

Bei dieser Methode wird das *System* durch ein Transitionssystem TS_{sys} modelliert und die *Spezifikation* durch ein Transitionssystem TS_{spec}. Zu prüfen ist dann, ob alle Transitionsfolgen des Systems auch solche der Spezifikation sind, d.h.

$$L(TS_{sys}) \subseteq L(TS_{spec})$$

Um dies zu prüfen, wird die für alle Mengen $P \subseteq R$ und $Q \subseteq R$ Beziehung genutzt:

$$P \subseteq Q \iff P \cap (R \setminus Q) = \emptyset$$

Wenn A die Grundmenge der Aktionen ist, bleibt also zu prüfen, ob folgendes gilt:

$$L(TS_{sys}) \cap (A^* \setminus L(TS_{spec})) = \emptyset$$

Dazu wird in Algorithmus 4.1 ein Transitionssystem $TS_\cap$ konstruiert, welches diesen Durchschnitt akzeptiert.

Dieses Vorgehen hat folgende Vorteile:

1. Wie gezeigt ist ein den Durchschnitt akzeptierendes Transitionssystem mithilfe des Produkttransitionssystems relativ einfach zu konstruieren (in quadratischer Zeit).
2. In linearer Zeit (in bezug auf die Größe des Transitionssystems) kann geprüft werden, ob die akzeptierte Sprache von $TS_\cap$ leer ist.
3. Oft ist das Transitionssystem (der endliche Automat) TS_{spec} deterministisch. Um das Komplement $A^* \backslash L(TS_{spec})$ zu akzeptieren, muss dann nur im vervollständigten Automaten die Endzustandsmenge komplementiert werden.
4. Ist der Durchschnitt nicht leer, dann können alle akzeptierten Folgen dieses Durchschnittes als Beispielablauf für Verletzungen der Spezifikation benutzt werden. Dies ist für die Korrektur des Systems von großem Nutzen.

Beispiel 4.1 (Model-Checking) Abbildung 4.1 zeigt das externe Verhalten des gestörten Sender-Empfänger-Systems von Abb. 2.4 als Transitionssystem TS_{sys} (in Abb. 2.4 als $(S \otimes R)_{extern}$ bezeichnet) und die Spezifikation TS_{spec} des gewünschten korrekten Verhaltens. Dabei wurde einschränkend die Annahme gemacht, dass ein Datum über den Kanal A nicht vor der Abgabe des vorangehenden über den Kanal C eingeht. Dies ist eine realistische Annahme, wie sie häufig bei Protokollen (z.B. dem Alternierbitprotokoll) vorliegt.
Das Transitionssystem $\overline{TS}_{spec}$ in der Mitte rechts akzeptiert $A^* \setminus L(TS_{spec})$. Es ist aus TS_{spec} konstruiert worden, indem es zunächst in Bezug auf die Aktionenmenge $A = \{r_A(d), s_C(d), s_C(\bot)\}$ vollständig gemacht wurde (die Schleife in v_2 gilt in Bezug auf alle Aktionen von A.). Danach wurde das Komplement $\{v_1, v_2\}$ als neue Menge von Endzuständen gewählt. Aus TS_{sys} und $\overline{TS}_{spec}$ wurde dann $TS_\cap$ konstruiert, das den Durchschnitt akzeptiert. $L(TS_\cap)$ ist nicht leer, da z.B. die Folgen $r_A(d)s_C(\bot)$ oder $r_A(d)s_C(\bot)r_A(d)s_C(d)$ enthalten sind. Das System ist also - wie erwartet - nicht korrekt. Die beiden Folgen können bei Tests zum Auffinden des Fehlers benutzt werden.

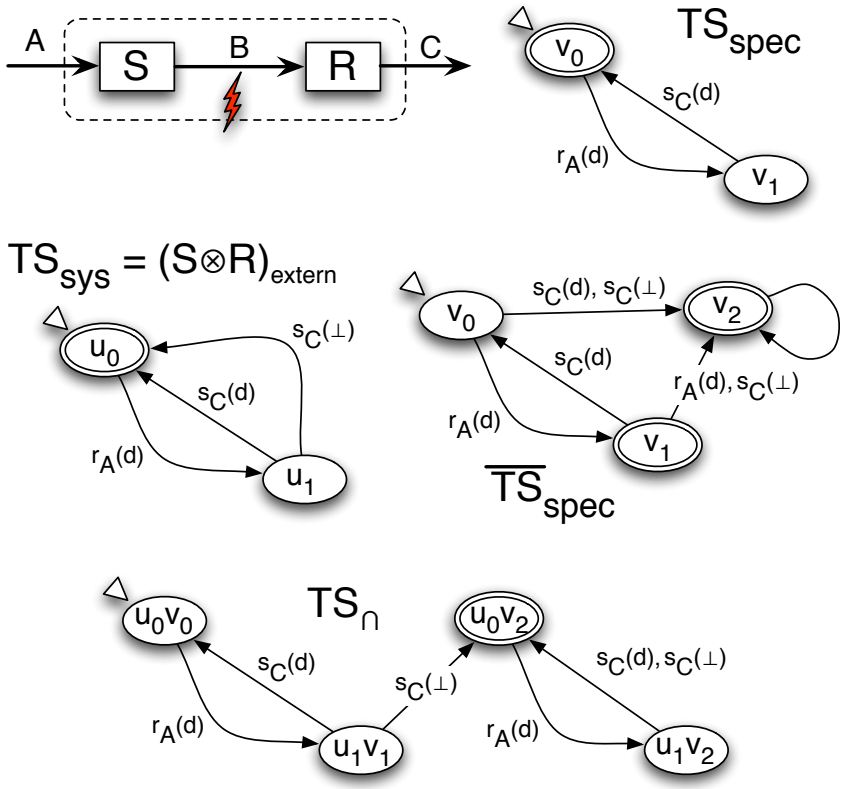

Abbildung 4.1: Model-Checking für das modifizierte gestörte Sender-Empfänger-System.

4.1.2 Model-Checking bei LTL-Spezifikationen

Ein für das Model-Checking fundamentaler Satz sagt, dass zu jeder LTL-Formel eine ihre Sprache akzeptierende Kripke-Struktur in Form eines *Büchi-Automaten* existiert.

Satz 4.2 *Zu jeder LTL-Formel f kann eine Kripke-Struktur (ein Büchi-Automat) M konstruiert werden, die genau die für f gültigen Folgen akzeptiert: $L^\omega(M) = L^\omega(f)$.*

Die Zeit- und Platz-Komplexität dieses Algorithmus ist $2^{\mathcal{O}(|f|)}$.
Diese obere Schranke kann nicht wesentlich verbessert werden, denn es kann eine Folge $\alpha_1, \alpha_2, \alpha_3, \cdots$ von LTL-Formeln derart konstruiert werden, dass die Größe von α_i durch ein Polynom n-ten Grades beschränkt wird, die entsprechenden Kripke-Strukturen aber mindestens 2^n Zustände haben.
Dieses Ergebnis ähnelt demjenigen von Satz 7.27 auf Seite 136. Beim Model-Checking ist die Formel f_{spec} einer System-Spezifikation meist jedoch klein gegenüber der Größe des System-Transitionssystems, so dass letzteres die Komplexität des Verfahrens dominiert.

Beispiel 4.3 Für einfache LTL-Formeln f kann man die ω-reguläre Sprache $L^\omega(f)$ auch direkt angeben. Sei $p \in AP$.
LTL-Formel $f = \Box p$:

$$L^\omega(\Box p) = \left(\sum_{M \ni p, M \subseteq AP} M \right)^\omega$$

LTL-Formel $f = \Diamond p$:

$$L^\omega(\Diamond p) = \mathcal{P}(AP)^* \cdot \left(\sum_{M \ni p, M \subseteq AP} M \right) \cdot \mathcal{P}(AP)^\omega$$

Wird die Spezifikation durch eine (temporal-)logische Formel f_{spec} festgelegt, so wandeln wir diese dann in ein äquivalentes Transitionssystem TS_{spec} um (siehe Satz 4.2).
Zu prüfen ist dann analog:

$$L^\omega(TS_{sys}) \ \subseteq \ L^\omega(TS_{spec}).$$

Um dies nachzuweisen, formen wir wieder um:

$$L^\omega(TS_{sys}) \ \cap \ (A^\omega \setminus L^\omega(TS_{spec})) = \emptyset$$

Eine analoge Formulierung von Algorithmus 4.1 ist in Algorithmus 4.2 dargestellt.
Algorithmus 4.2 ist zwar effektiv (weil alle benutzten Konstruktion für Büchi-Automaten – Komplementbildung, Produktbildung, Test auf Leerheit – dies sind), es ist aber nicht sehr effizient, da die Komplementbildung für Büchi-Automaten eine enorme Zeitkomplexität aufweist: 2^{n^2}, wobei n die Anzahl der Zustände des Ausgangsautomaten ist.
Es gibt aber einen effizienteren Ansatz: Ist die Spezifikation durch eine Formel f_{spec} gegeben, dann kann man einfach zur Negation $\neg f_{spec}$ übergehen und dann zu dieser Formel

Algorithmus 4.2 LTL Model-Checking, erster Ansatz

```
PROCEDURE Check(L(TS_sys), L(TS_spec))
    construct complement TS_spec with L^ω(TS_spec) = (A^ω \ L^ω(TS_spec))
    construct product TS_4 with L^ω(TS_∩) = L^ω(TS_sys) ∩ L^ω(TS_spec)
    IF L^ω(TS_∩) = ∅ THEN return true
    ELSE return false
END PROCEDURE
```

Algorithmus 4.3 LTL Model-Checking, effizientere Variante

```
PROCEDURE Check(L(TS_sys), f)
    construct TS_¬f_spec
    construct product TS_∩ with L^ω(TS_∩) = L^ω(TS_sys) ∩ L^ω(TS_¬f_spec)
    IF L^ω(TS_∩) = ∅ THEN return true
    ELSE return false
END PROCEDURE
```

den Büchi-Automaten $TS_{\neg f_{spec}}$ konstruieren, der dann $L^\omega(TS_{\neg f_{spec}}) = A^\omega \setminus L^\omega(TS_{f_{spec}})$ akzeptiert, also bereits das gewünschte Komplement (vgl. Algorithmus 4.3).

Wie groß ist also die Komplexität des Verfahrens?

Wir wissen bereits, dass zu jeder LTL-Formel f ein äquivalenter Büchi-Automat M konstruiert werden kann und dass die Zeit- und Platz-Komplexität dieses Algorithmus $2^{\mathcal{O}(|f|)}$ ist, da der Automat $2^{\mathcal{O}(|f|)}$ Zustände besitzt. Außerdem wissen wir, dass der Produktautomat A_4 Tripel als Zustände besitzt, also hier $2 \cdot |M| \cdot |TS|$ viele. Um $L^\omega(TS_\cap) = \emptyset$ festzustellen, müssen wir feststellen, ob ein erreichbarer Zustand auf einem Zyklus liegt. Dies braucht höchstenst soviel Zeit und Platz, wie der Produktautomat A_4 Zustände besitzt: $\mathcal{O}(|TS| \cdot |M|)$. Insgesamt erhalten wir $\mathcal{O}(|TS| \cdot 2^{\mathcal{O}(|f|)})$ als Gesamtkomplexität.

4.1.3 Beispiel: Zustandsraumanalyse mit Maude

Das Werkzeug MAUDE erlaubt eine Analyse temporallogischer Formeln [EMS02]. Dabei kommt speziell die Logik LTL zum Einsatz. Die temporallogischen Operatoren für den LTL-Analysator werden durch das Modul `MODEL-CHECKER` in MAUDE-Syntax definiert.

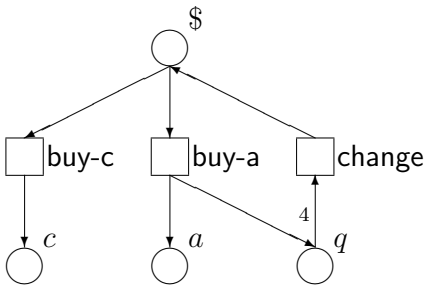

Abbildung 4.2: Ein Beispielnetz

Betrachten wir die Spezifikation des Beispielnetzes aus Abb. 4.2. Es besitzt folgende Entsprechung in der Spezifikationssprache MAUDE [CDE+99]:

```
mod CANDY-PN is
  sorts Place Marking .
  subsort Place < Marking .
  op _ _ : Marking Marking -> Marking [assoc comm] .
  ops $ q a c : -> Place .

  rl [buy-candy] : $ => c .
  rl [buy-apple] : $ => a  q .
  rl [change] : q  q  q  q => $ .
endm
```

Es existieren die Sorten der Stellen (`Place`) und der Markierungen (`Marking`), wobei `Place` eine Subsorte von `Marking` ist. Der Multimengenoperator + auf Markierungen wird abkürzend als Operator ohne Symbol notiert (`op _ _`). Die konkreten Stellen `$`, `q`, `a`, `c` sind Konstanten vom Typ `Place`. Die Transitionen sind als Ersetzungsregeln (`rl [buy-c]` usw.) beschrieben.
Das folgende Modul definiert die für eine Zustandsraumanalyse notwendigen Prädikate:

```
mod CHECK-CANDY is
 inc CANDY-PN .
 inc MODEL-CHECKER .

 subsort Marking < State .
 ops p-candy  p-apple :  -> Prop .
 var M : Marking .
```

```
  eq ((M c)  |= p-candy) = true .
  eq ((M a)  |= p-apple) = true .

  op phi1 : -> Prop .
  eq phi1 = <> (p-candy \/ p-apple) .

  op phi2 : -> Prop .
  eq phi2 = <> p-candy .
endm
```

Importiert wird neben dem Modell `CANDY-PN` des Petrinetzes auch die Spezifikation `MODEL-CHECKER`. Durch `subsort Marking < State` wird definiert, dass Markierungen des Petrinetzes die Zustände der Kripke-Struktur sind. Die Übergänge werden automatisch durch die Ersetzungsregeln erzeugt. Es werden die beiden atomaren Propositionen `p-candy` und `p-apple` definiert. Dabei gilt `p-candy`, wenn in der Markierung mindestens eine Süßigkeit, d.h. eine Marke c vorhanden ist. Dann muss die Markierung die Form $M + c$ besitzt. Dies wird durch die Gleichung `eq ((M c) |= p-candy) = true` ausgedrückt, wobei `|=` ein Prädikat ist, das Zustände (`State`) mit Propositionen (`Prop`) in Relation setzt. Analog gilt `p-apple`, wenn in der Markierung M mindestens eine Marke a vorhanden ist.
Die Formel $\phi_1 = \Diamond(\texttt{p-candy} \vee \texttt{p-apple})$ drückt aus, dass irgendwann eine Marke c (eine Süßigkeit) oder eine Marke a (ein Apfel) vorhanden sein wird. Sie wird in MAUDE durch die Gleichung

```
eq phi1 = <> (p-candy \/ p-apple)
```

notiert. Die Formel `phi2 = <> p-candy` beschreibt, dass irgendwann eine Süßigkeit vorhanden sein wird.
Um zu überprüfen, ob eine Formel ϕ gilt, muss sich die Relation $M_0 \models \phi$ zu „true“ auswerten lassen. Die Gültigkeit für die Formel `phi1` in der Initialmarkierung $M_0 = \$+\$$ ergibt mit der folgenden Ersetzung:

```
Maude> rew [10]  $ $   |= phi1 .
rewrite [10] in CHECK-CANDY : $ $ |= phi1 .
rewrites: 11 in 10ms cpu (7ms real) (1100 rewrites/second)
result Bool: true
```

Die Formel `phi2` ist nicht gültig, was man an der Ausführungsfolge, die mit $w_1 =$ *buy-apple* $\cdot$ *buy-apple* beginnt, erkennt. Dieses Gegenbeispiel konstruiert auch MAUDE in Folge der Analyse:

```
Maude> rew [10] $ $ |= phi2 .
rewrite [10] in CHECK-CANDY : $ $ |= phi2 .
rewrites: 15 in 0ms cpu (6ms real) (~ rewrites/second)
result ModelCheckResult:
  counterexample({c a q,'buy-apple}
     {c a q,'buy-apple},
     {a a q q,deadlock})
```

4.2 CTL-Model-Checking

Die *CTL-Model-Checking* Aufgabe lautet:

- Berechne für eine gegebene (endliche) Kripke-Struktur $M := (S, S_0, R, E_S)$ und eine gegebene CTL-Formel f die Menge:

$$Sat(f) := \{s \in S \mid M, s \models f\}$$

Der Algorithmus 4.4 erweitert $E_S(s)$ für alle $s \in S$ schrittweise zu $label(s)$.
Dadurch enthält $label(s)$ alle Teilformeln von f, die in s wahr sind.
Durch Rekursion über die Schachtelungstiefe von f gilt in Schritt i: alle Teilformeln mit $i-1$ geschachtelten CTL-Operatoren sind behandelt.
Da alle CTL-Operatoren nach Satz 3.11 mittels EXg, EGg und $E[g_1 U g_2]$ ausgedrückt werden können, bleiben nur noch zwei nicht nicht-elementare Unterprozeduren: $CheckEU(f_1, f_2)$ und $CheckEG(f_1)$.

Algorithmus 4.4

```
PROCEDURE Check(f)
    IF f ∈ AP THEN
        FORALL s ∈ S SUCH THAT f ∈ E_S(s) DO label(s) := label(s) ∪ {f};
    IF f = ¬f_1 THEN
        Check(f_1);
        FORALL s ∈ S SUCH THAT f_1 ∉ label(s) DO label(s) := label(s) ∪ {f};
    IF f = f_1 ∨ f_2 THEN
        Check(f_1); Check(f_2);
        FORALL s ∈ S SUCH THAT f_1 ∈ label(s) ∨ f_2 ∈ label(s) DO label(s) := label(s) ∪ {f};
    IF f = EXf_1 THEN
        Check(f_1);
        FORALL s ∈ S SUCH THAT R(s,t) ∧ f_1 ∈ label(t) DO label(s) := label(s) ∪ {f};
    IF f = E[f_1 U f_2] THEN
        Check(f_1); Check(f_2); CheckEU(f_1, f_2)
    IF f = EGf_1 THEN
        Check(f_1); CheckEG(f_1)
END PROCEDURE
```

4.2.1 Die Unterprozedur $CheckEU(f_1, f_2)$

Zunächst markieren wir alle Zustände s, die f_2 erfüllen. Dann fahren wir schrittweise in Gegenrichtung der Transitionen fort und markieren die Zustände, die f_1 erfüllen (vgl. Algorithmus 4.5). Dadurch markieren wir einen Zustand s mit f, falls es einen Pfad von s zu einem s' mit $f_2 \in label(s')$ gibt, so dass für alle Zustände t davor $f_1 \in label(t)$ gilt.

4.2.2 Die Unterprozedur $CheckEG(f_1)$

Für diese Prozedur benötigen wir den Begriff der strengen Zusammenhangskomponente, denn eine Formel f_1 gilt ja nur dann, wenn es einen Kreis in der Kripke-Struktur gibt, auf dem stets f_1 gilt.

Algorithmus 4.5 Auszeichnen mit $E(f_1 U f_2)$

```
PROCEDURE CheckEU(f1, f2)
    T := {s | f2 ∈ label(s)};
    FORALL s ∈ T DO label(s) := label(s) ∪ {E[f1 U f2]};
    WHILE T ≠ ∅ DO
        CHOOSE s ∈ T;
        T := T \ {s};
        FORALL t SUCH THAT R(t, s) DO
            IF E[f1 U f2] ∉ label(t) AND f1 ∈ label(t) THEN
                label(t) := label(t) ∪ {E[f1 U f2]};
                T := T ∪ {t};
            END IF;
        END FORALL;
    END WHILE;
END PROCEDURE;
```

Definition 4.4 *Sei $G = (K, R)$ ein gerichteter Graph, d.h.: $R \subseteq K \times K$:*

a) Eine Knotenmenge $A \subseteq K$ heißt Zusammenhangskomponente*, falls: $\forall a, a' \in A : aR^*a'$.*

b) $A \subseteq K$ heißt strenge Zusammenhangskomponente (SZK) *(strongly connected component: SZK), falls sie maximal ist, d.h.: $\neg\exists k \in K \setminus A\,. : \forall a \in A : kR^*a \wedge aR^*k$.*

c) Sie heißt nichttriviale Zusammenhangskomponente*, falls sie mehr als einen Knoten enthält oder eine Schleife: $|A| > 1$ oder $\exists a \in A : aR^+a$.*

Nun betrachten wir wieder die Formel: $f = EGf_1$:
Sei $M = (S, S_0, R, label)$ die im hier entwickelten CTL-Algorithmus jeweils durch die Abbildung *label* erweiterte Kripke-Struktur. Daraus konstruieren wir $M' = (S', S'_0, R', label')$ mit:

$$\begin{aligned} S' &:= \{s \in S \mid M, s \models f_1\} \\ S'_0 &:= S_0 \cap S' \\ R' &:= R_{|S' \times S'} \\ label' &:= label_{|S'} \end{aligned}$$

d.h. die „Einschränkung" von M auf Zustände, in denen f_1 gilt. Es werden also alle Zustände und anhängende Kanten gestrichen, in denen f_1 nicht enthalten ist.

Lemma 4.5 *Es gilt genau dann $M, s \models EGf_1$, wenn (1.) $s \in S'$ und (2.) es einen Pfad in M' gibt, der von s zu einer nichttrivialen strengen Zusammenhangskomponente in (S', R') führt.*

Beweis: Als Übung. □

Daraus resultiert folgender Algorithmus zur Entscheidung von EGf_1 (vgl. Algorithmus 4.6):

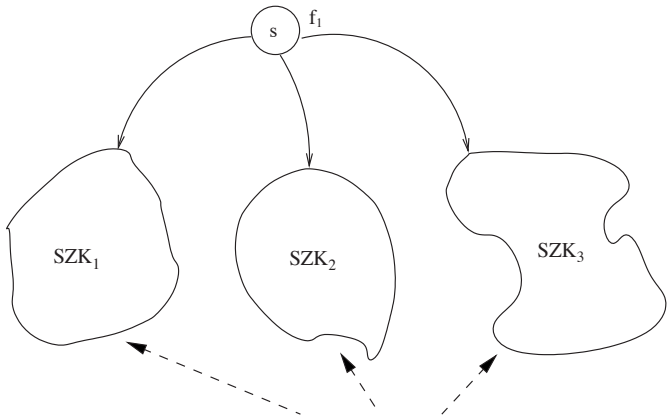

Abbildung 4.3: Strenge Zusammenhangskomponenten mit f_1

Algorithmus 4.6 Auszeichnen mit EGf_1

```
PROCEDURE CheckEG(f1)
    S' := {s | f1 ∈ label(s)};
    SCC := {C | C a nontrivial SCC of S'};
    T := ⋃_{C∈SCC} {s | s ∈ C};
    FORALL s ∈ T DO label(s) := label(s) ∪ {EGf1};
    WHILE T ≠ ∅ DO
        CHOOSE s ∈ T;
        T := T \ {s};
        FORALL t SUCH THAT t ∈ S' AND R(t, s) DO
            IF EGf1 ∉ label(t) THEN
                label(t) := label(t) ∪ {EGf1};
                T := T ∪ {t};
            END IF ;
        END FORALL ;
    END WHILE ;
END PROCEDURE ;
```

1. Konstruiere $M' = (S', S'_0, R', label')$.
2. Konstruiere alle SZK von M'. (Algorithmus von Tarjan mit $O(|S'| + |R'|)$ Zeitkomplexität.)
3. Finde Zustände in nichttrivialen SZK.
4. Suche von diesen aus rückwärts alle Zustände, die dorthin führen. (Das sind maximal $\mathcal{O}(|S| + |R|)$ viele.)

Satz 4.6 *Der Algorithmus 4.4, der für eine Kripke-Struktur $M := (S, S_0, R, E_S)$ und eine CTL-Formel f entscheidet, ob f für M gilt, d.h. ob $M \models f$, besitzt die Zeitkomplexität $\mathcal{O}(|f| \cdot (|S| + |R|))$.*

Beweis: Wendet man Algorithmus 4.4 auf eine Kripke-Struktur an, dann wird für jede Teilformeln von f eine Subprozedur gestartet. Die Anzahl der Teilformeln ist durch $\mathcal{O}(|f|)$ begrenzt.

Für jede Teilformel benötigt die jeweilige Subprozedur maximal $\mathcal{O}(|S| + |R|)$ viele Operationen, denn die Suchverfahren für $E[-U-]$ besuchen i.w. jeden Knoten einmal und die Berechnung der SCCs für $EG-$ geschieht auch in Linearzeit. Insgesamt gibt es also maximal $\mathcal{O}(|f| \cdot (|S| + |R|))$ Operationen. □

Beispiel 4.7 (MW-Ofen) CTL-Spezifikation zu der Kripke-Struktur in Abb. 3.1:

$$f = AG(Start \rightarrow AF\, Heat)$$

Es gilt immer: nach einem Zustand mit „Start" wird später ein Zustand mit „Heat" erreicht.
Zunächst muss f mit Hilfe der Äquivalenzen von Satz 3.11 so umgeschrieben werden, dass nur noch die Operationen enthalten sind, die der Algorithmus verarbeiten kann. Dabei steht wegen der besseren Lesbarkeit Fg für **true** $U\ g$.

$$\begin{aligned} AGf &\Leftrightarrow \neg EF(\neg f) \Leftrightarrow \neg EF(\neg(\neg Start \vee AF\, Heat)) \\ &\Leftrightarrow \neg EF(Start \wedge \neg AF\, Heat) \qquad\qquad (AFf \Leftrightarrow \neg EG(\neg f)) \\ &\Leftrightarrow \neg EF(Start \wedge EG\neg Heat) \end{aligned}$$

Bezeichnet $Sat(Start) = \{2, 5, 6, 7\}$ die Menge der Zustände, in denen $Start$ gilt und entsprechend $Sat(\neg Heat) = \{1, 2, 3, 5, 6\}$, dann ist, um $f = EG\neg Heat$ zu behandeln, $\{1, 2, 3, 5\}$ die einzige SZK von $Sat(\neg Heat)$, d.h. der Zustand 6 wird ausgeschlossen.
Zustände, die mit $EG\neg Heat$ zu markieren sind also $T = \{1, 2, 3, 5\} = Sat(EG\neg Heat)$.
Es folgt:

$$Sat(Start \wedge EG\neg Heat) = \{2, 5, 6, 7\} \cap \{1, 2, 3, 5\} = \{2, 5\}$$

und durch Rückwärtspfade:

$$Sat(EF(Start \wedge EG\neg Heat)) = \{1, 2, 3, 4, 5, 6, 7\}.$$

Damit ergibt sich schließlich:

$$Sat(\neg EF(Start \wedge EG\neg Heat)) = \emptyset$$

und folglich $M, 1 \not\models AG(Start \rightarrow AF\, Heat)$ d.h. die Spezifikation f gilt *nicht* im Anfangszustand.

4.3 CTL-Model-Checking mit Fairness

Fairness-Spezifikationen sind für viele Anwendungen wichtig. Sie lassen sich oft in LTL ausdrücken, aber nicht in CTL, was wegen der besseren Komplexitätseigenschaften wünschenswert wäre.
Hier zwei Beispiele:

- „Eine Alternative einer sich ständig wiederholenden Alternative wird irgendwann einmal auch gewählt" z.B. Hardware Arbiter.
- „Ein gestörter Kanal übermittelt immer wieder einmal eine Nachricht korrekt" z.B Alternierbitprotokoll.

Eine Lösung dieses Problems besteht darin, dass die Fairness-Spezifikation auf die Kripke-Struktur verlagert wird (man spricht auch von „fairer Semantik"). Da Fairness-Bedingungen durch Endzustände in Transitionssystemen darstellbar sind, führt man Endzustände für Kripke-Strukturen ein und verlagert die Fairness-Spezifikation von der temporallogischen Formel auf das Systemtransitionssystem, also von f_{spec} auf TS_{sys}. Es wird dann CTL-Model-Checking auf die akzeptierten unendlichen Folgen angewandt, anstatt auf alle möglichen Folgen von TS_{sys}. Da eine Endzustandsmenge nur eine einzige Fairness-Spezifikation ausdrücken kann, benötigt man für mehrere solche Spezifikationen mehrere Endzustandsmengen. Dieses Modell heißt „*faire Kripke-Struktur*" oder „*verallgemeinerter Büchi-Automat*".

Definition 4.8 *Eine* faire *Kripke-Struktur*

$$M := (S, S_0, R, E_S, \{E_F^1, \cdots, E_F^k\})$$

besteht aus einer Kripke-Struktur $M := (S, S_0, R, E_S)$ *und den* $k \geq 1$ *Endzustandsmengen* $E_F^i \subseteq S$.
Ein Pfad $\pi = s_0 s_1 s_2 \cdots \in SS(M)$ *heißt* fair, *falls* $\text{infinite}(\pi) \cap E_F^i \neq \emptyset$ *für alle* $i \in \{1, \cdots, k\}$ *gilt*

Fairness wird also über eine Akzeptanzbedingung definiert, die wir bereits vom Büchi-Automaten kennen (vergl. Definition 1.10 auf Seite 13).
Bezogen auf die Einschränkung auf faire Pfade spricht man von *fairer Gültigkeit* (in Zeichen: $\models_F$) und ändert die Bedingungen 1, 5 und 6 von Definition 3.10 auf Seite 54 wie folgt:

1.	$M, s \models_F p$	$\Longleftrightarrow$	Es gibt einen **fairen** Pfad, der bei s anfängt mit $p \in E_S(s)$.
5.	$M, s \models_F Ef$	$\Longleftrightarrow$	Es gibt einen in s beginnenden **fairen** Pfad π mit $M, \pi \models f$.
6.	$M, s \models_F Af$	$\Longleftrightarrow$	Für alle in s beginnenden **fairen** Pfade π gilt $M, \pi \models f$.

Beispiel 4.9

$$E_F^i = \{s | s \text{ erfüllt } \neg send_i \vee receive_i \text{ für Kanal } i\}$$

Ein fairer Pfad impliziert: in jedem Kanal wird unendlich oft empfangen, falls gesendet wird.

Definition 4.10 *Sei $M := (S, S_0, R, E_S, E_F^1, \cdots, E_F^k)$ eine faire Kripke-Struktur. Eine starke Zusammenhangskomponente $C \subseteq S$ heißt fair, falls $\forall i \in \{1, \cdots, k\} : E_F^i \cap C \neq \emptyset$. Ferner sei $M' := (S', S_0', R', E_S', F_1, \cdots, F_k)$ mit:*

$$\begin{aligned} S' &= \{s \in S \mid M, s \models_F f_1\} \qquad (\models_F \text{ siehe Seite 70}) \\ R' &= R_{|S' \times S'} \\ E_S' &= E_{S|S'} \\ F_i &= E_F^i \cap C \end{aligned}$$

Lemma 4.11 *Es gilt genau dann $M, s \models_F EGf_1$, wenn (1.) $s \in S'$ und (2.) es einen Pfad in M' gibt, der von s zu einer fairen, nichttrivialen starken Zusammenhangskomponente in (S', R') führt.*

Daraus folgt eine Prozedur $CheckFairEG(f_1)$ um Spezifikation in fairer Semantik zu prüfen:

$$fair := EG\,True \qquad \text{„Es gibt eine unendliche Folge"}$$

und

$$\begin{array}{lcl} M, s \models_F p & \iff & M, s \models p \wedge fair \\ M, s \models_F EXf_1 & \iff & M, s \models EX(f_1 \wedge fair) \\ M, s \models_F E[f_1 U f_2] & \iff & M, s \models E[f_1 U (f_2 \wedge fair)] \end{array}$$

Satz 4.12 *Es gibt einen Algorithmus, der für eine faire Kripke-Struktur $M := (S, S_0, R, E_S, E_F^1, \cdots, E_F^k)$ und eine CTL-Formel f in $\mathcal{O}(|f| \cdot (|S| + |R|))$ Zeitkomplexität entscheidet, ob f für M in der fairen Semantik gilt, d.h. ob $M \models_F f$.*

Beispiel 4.13 Prüfe $f = AG(Start \rightarrow AF\,Heat)$, wobei vorausgesetzt wird, dass die Benutzer den Ofen immer korrekt bedienen. Dabei interpretieren wir „immer korrekt bedienen" als „unendlich oft gilt $Start \wedge Close \wedge \neg Error$".
Daher setzen wir $M := (S, S_0, R, E_S, E_F^1)$ (also $k = 1$) mit

$$E_F^1 = \{s \mid s \models Start \wedge (Close \wedge \neg Error)\} = \{6, 7\}.$$

Entsprechrend $Sat(f)$ definieren wir $Sat_F(f) := \{s \in S \mid M, s \models_F f\}$. Mit $Sat_F(Start) = Sat(Start)$, $Sat_F(\neg Heat) = Sat(\neg Heat)$ wie vorher ist die ZSK $\{1, 2, 3, 5\}$ *nicht* fair, da sie disjunkt zu $E_F^1 = \{6, 7\}$ ist. Also:

$$\begin{aligned} Sat_F(EG\neg Heat) &= \emptyset \\ Sat_F(EF(Start \wedge EG\neg Heat)) &= \emptyset \\ Sat_F(\neg EF(Start \wedge EG\neg Heat)) &= \{1, \cdots, 7\} \end{aligned}$$

Die Spezifikation ist in der fairen Semantik erfüllt, da die Formel f im Anfangszustand gilt: $M, 1 \models_F AG(Start \rightarrow AF\,Heat)$.

Aufgabe 4.14 (CTL-Model-Checking) Prüfen Sie die folgende Spezifikation für das Ofenbeispiel durch den CTL-Algorithmus: $AG(Start \wedge \neg Close \wedge \neg Heat \wedge Error \Rightarrow EF\neg Error)$.