Sicherheits- und Lebendigkeitseigenschaften konkurrierender Prozesse

Ein Vergleich zwischen FSP und StateCharts

Martin Pitt
Technische Universität Dresden
Fakultät Informatik
martin@piware.de

13. September 2002

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

„Die Mathematik befriedigt den Geist durch ihre außerordentliche Gewissheit“

Johannes Kepler (1571-1630), dt. Astronom

Dieses Dokument entstand im Rahmen des Komplexpraktikums „Formale Modelle in der Anwendung“ des Institutes für Theoretische Informatik, was von Prof. Reichel durchgeführt wurde.

In diesem Praktikum ging es darum, sich selbstständig Konzepte und Methoden zur Beschreibung, Modellierung und Verifikation von parallelen Prozess-Systemen zu erarbeiten und anhand zweier ausgewählte Programmpakete, Labeled Transition System Analyzer (LTSA) und StateMate Magnum 3.0 einen Einstieg in die praktische Verwendung davon zu finden.

Ich hatte die Aufgabe, mich mit Sicherheits- und Lebendigkeitseigenschaften solcher Systeme und deren maschineller Verifikation zu beschäftigen. Dieses Dokument beschreibt meine Ergebnisse und gibt auch einen subjektiven Eindruck von der Arbeit mit den beiden Programmpaketen.

Voraussetzungen

Dieses Dokument setzt folgendes Vorwissen voraus:

- allgemeine Kenntnisse über finite Automaten
- ein prinzipielles Verständnis von den Problemen paralleler Prozesse (Interferenz, Deadlocks, etc.)
- die Kenntnis des von LTSA verwendeten Prozess-Kalküls, beschrieben in Kapitel 1 bis 3 von [MaKr]
- Kenntnis von UML, der von StateMate in den State-Charts verwendeten Beschreibungssprache

Kapitel 2

Sicherheit

2.1 Begriffs-Definition

Unter **Sicherheit** versteht man die *partielle Korrektheit* eines Systems: es passiert niemals etwas schlechtes oder gefährliches.

Bei sequentiellen Programmen heißt dies: jedes ausgegebene Ergebnis ist korrekt. Bei parallelen Prozessen bedeutet das zusätzlich, dass niemals Interferenz beim Zugriff auf einen gemeinsamen Speicher oder Deadlocks auftreten.

2.2 Modellierung in FSP

Um Sicherheitseigenschaften paralleler Prozesse formell nachweisen zu können, muss zunächst vereinbart werden, wie man den Begriff „etwas schlechtes oder gefährliches“ formal darstellen kann. Dann kann man diesen Formalismus in das FSP-Modell einbauen, um es mechanisch zu verifizieren.

FSP verwendet dabei einen ganz einfachen Formalismus: immer, wenn „etwas schlechtes oder gefährliches“ passiert, findet eine Transition in den Fehlerzustand `ERROR` statt. Dies kann auf zwei verschiedene Weisen erfolgen.

2.2.1 Explizite Transition nach `ERROR`

Diese Methode soll anhand eines einfachen Beispiels beschrieben werden. Man stelle sich ein einfaches Service-Objekt vor, das Kommandos entgegen nimmt und diese mit einer Antwort quittiert. Dabei soll es nicht zulässig sein, dass nach einem empfangenen `COMMAND` gleich wieder ein `COMMAND` eintrifft, noch bevor eine Antwort gesendet wird:

```
ACTUATOR = ( command -> ACTION ),
ACTION = ( respond -> ACTUATOR | command -> ERROR ).
```

Abbildung 2.1 zeigt den Automaten-Graphen des `ACTUATOR`s.

2.2.2 Angabe von Sicherheitseigenschaften

Manchmal ist es nicht so einfach, nicht bequem oder sogar statisch unmöglich, alle Möglichkeiten, *was nicht* sicher ist, zu modellieren, sondern es ist viel leichter festzustellen, *was* sicher ist.

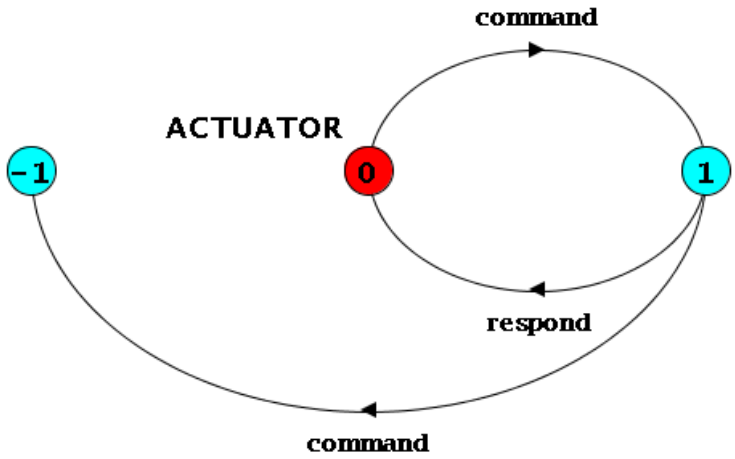

Abbildung 2.1: Automaten-Graph des `ACTUATOR` in FSP

Definition: Eine **Sicherheitseigenschaft** P definiert einen deterministischen(!) Prozess, der sicherstellt, dass alle Transitions-Sequenzen, die im Alphabet von P enthalten sind, von P akzeptiert werden.

Schreibweise: `property` $< Prozessdefinition >$

LTSA fügt dem so entstehenden Prozess *automatisch* Transitionen zu `ERROR` für alle nicht explizit angegebenen Transitions-Folgen hinzu. Als Beispiel schreiben wir den `ACTUATOR` mit Hilfe einer Sicherheitseigenschaft:

```
property SAFE_ACTUATOR =
         ( command -> respond -> SAFE_ACTUATOR ).
```

Abbildung 2.2 zeigt den nun entstehenden Automaten.

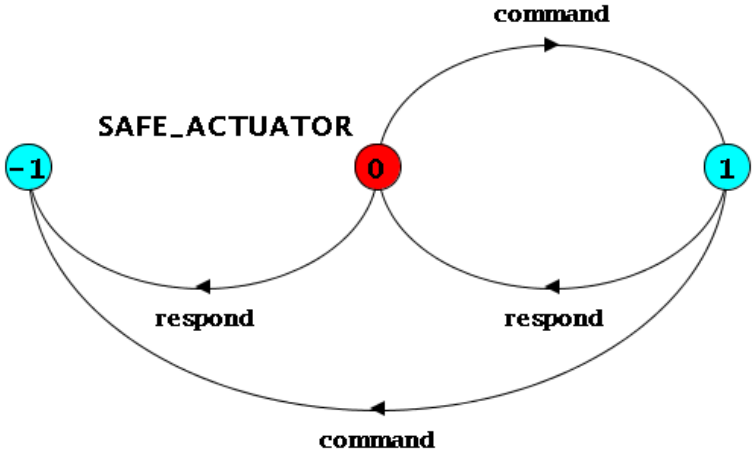

Abbildung 2.2: Automaten-Graph des `SAFE-ACTUATOR` in FSP

Um aus einer solchen Sicherheitseigenschaft einen Nutzen zu ziehen, komponiert man sie mit einem bestehenden System. Dabei synchronisieren sich alle Transitionen auf der Alphabet-Schnittmenge. Das verändert den normalen Ablauf des Prozess-Systems nicht (der `property`-Automat wird sozusagen nur „mitgeschleift“), führt aber bei fehlerhaften Transitionen zu `ERROR`.

Der LTSA kann solche Verletzungen der Sicherheitseigenschaften durch erschöpfende Suche erkennen und zeigt bei einer Verletzung sogar den kürzesten Pfad dahin an, so dass man als Entwerfer genau weiß, wo das Problem liegt.

Noch ein kleines Beispiel: möchte man beweisen, dass in einem bestehenden Prozess-System P niemals die Transition `disaster` (die in den einzelnen Prozessen durchaus vorkommen kann!) ausgeführt wird, definiert man:

```
property SAFE = STOP +{disaster}.
```

und komponiert beides zu `||CHECK = ( P || SAFE ).` Setzt man den LTSA via „Check Safety“ darauf an, bekommt man entweder den Beweis des Nicht-Auftretens oder den kürzesten Pfad der Verletzung der `SAFE`-Eigenschaft.

Ausführlichere Beispiele folgen in den nächsten Abschnitten.

2.3 Modellierung in StateCharts

Nach ausführlichem Studium der Handbücher, der Online-Hilfe und einer Internet-Recherche bin ich zu folgenden Schlüssen gekommen:

- StateCharts haben keinen ausgezeichneten Fehlerzustand
- StateCharts unterstützen keine Sicherheitseigenschaften
- es gibt zwar einen Befehl „Check Model“, dieser testet aber nur auf syntaktische Korrektheit.

Man kann zwar das schon verwendete Beispiel `ACTUATOR` problemlos in StateCharts übersetzen (siehe Abbildung 2.3), “Check Model” meldet aber keine Verletzung der Sicherheit und Vollständigkeit.

Dies bedeutet, dass StateMate in seiner puren Form Model Checking nicht beherrscht. Es gibt allerdings Zusatz-Pakete, die dies leisten. I-Logix selbst hat unter `www.realtime-info.be/vpr/layout/display/pr.asp?PRID=1999` eine Lösung angekündigt und es gibt auch Erweiterungen anderer Hersteller (z. B. kann man StateCharts in Promela übersetzen und diese dann mit Spin analysieren). Dies wurde aber im vorliegenden Praktikum nicht getan, zum einen, weil die Zusatz-Pakete nicht zur Verfügung standen, zum anderen, weil der zeitliche Umfang beschränkt war.

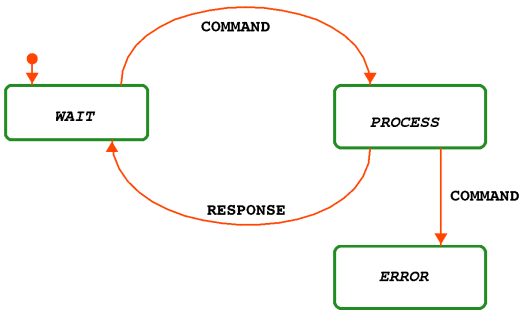

Abbildung 2.3: Der `ACTUATOR` in StateCharts

2.4 Beispiel: Semaphore

Weil man anhand einer Semaphore Sicherheitseigenschaften leicht formulieren und zeigen kann, wird dieses Konzept als Beispiel herangezogen. Die Modellierung zeigt auch die unterschiedlichen Strategien und Probleme, die bei der Arbeit mit beiden Systemen auftreten.

2.4.1 Modellierung in FSP

Das folgende FSP-Modell definiert eine klassische Semaphore und ein Testprogramm `LOOP`, was einen durch die Semaphore geschützten kritischen Abschnitt besitzt. Dabei interessiert nur der Eintritt und das Verlassen dieses Abschnittes, dargestellt durch die Ereignisse `enter` und `exit`.

```
const Max = 3
range Int = 0..Max

const SemaMax = 1 /* 1 is safe, 2 is unsafe */

SEMAPHORE( N=1 ) = SEMA[N],
SEMA[v:Int]    = ( up->SEMA[v+1] | when(v>0) down->SEMA[v-1] ).

LOOP = ( mutex.down -> enter -> exit -> mutex.up -> LOOP ).

||SEMADEMO = ( p[1..3]:LOOP || {p[1..3]}::mutex:SEMAPHORE(SemaMax) ).
```

Wir lassen also 3 `LOOP`-Prozesse parallel laufen. Um die Sicherheit des Systems zu beweisen, verwenden wir die folgende Eigenschaft:

```
property MUTEX = ( p[i:1..Max].enter -> p[i].exit -> MUTEX ).

||CHECK = ( SEMADEMO || MUTEX ).
```

Das heißt, nachdem der Prozess i den kritischen Bereich betreten hat, muss er als nächstes auch den kritischen Bereich verlassen. Alle anderen Ereignisse (ein anderer Prozess betritt oder verlässt den kritischen Abschnitt) stellen einen Fehler dar.

Den Graphen der Eigenschaft `MUTEX` zeigt die Abbildung 2.4, das entstehende System die Abbildung 2.5. Wie man leicht sieht (und wie der LTSA auch beweist), gibt es keine Transition nach `ERROR`, das heißt, unser System ist korrekt. Ändert man jedoch Test-weise die Konstante `SemaMax` auf 2 (d. h., maximal 2 Prozesse dürfen die Semaphore gleichzeitig passieren), wird die `MUTEX`-Eigenschaft erwartungsgemäß verletzt.

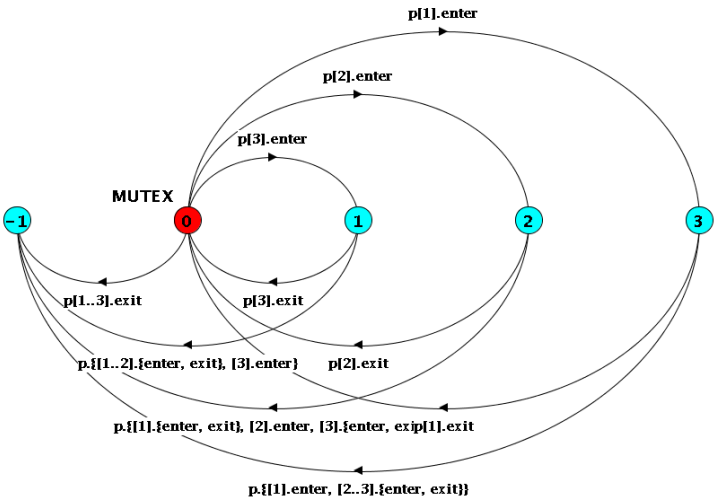

Abbildung 2.4: Graph der Eigenschaft `MUTEX`

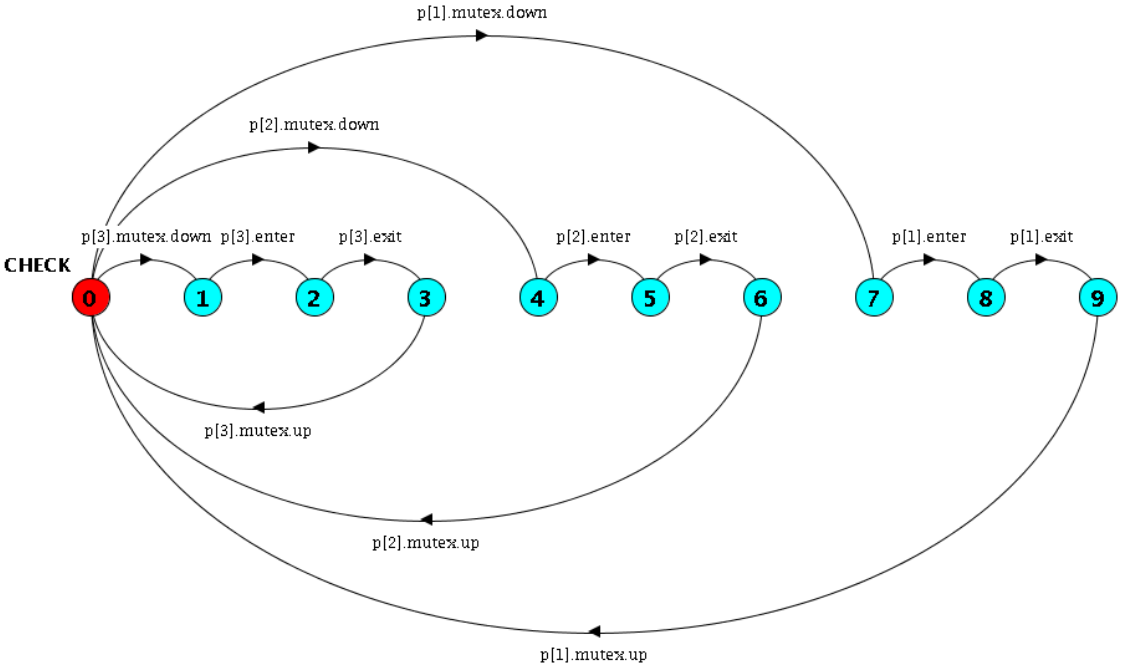

Abbildung 2.5: Graph des Semaphoren-Systems

2.4.2 Modellierung in StateMate

Abbildung 2.6 zeigt eine Modellierung in StateCharts, die sich möglichst eng an das entsprechende FSP-Modell anlehnt. Beim Zeichnen sind mir folgende Eigenschaften der StateCharts aufgefallen:

- Das Zeichnen dauert um Größenordnungen länger als das Eintippen der FSP-Prozesse
- Es ist nicht möglich, die Anzahl der Prozesse statisch zu ändern, da StateCharts im Gegensatz zu FSP nicht parametrisierbar sind; das heißt, man ist bei oben stehendem Modell auch auf die zwei `LOOP`-Prozesse beschränkt
- StateCharts haben keine harte Synchronisation, das heißt: existieren zwei parallele Prozesse mit einer gemeinsamen Transition, d. h. $A \xrightarrow{t} A'$ und $B \xrightarrow{t[con]} B'$, so kann B bei falscher Bedingung *con* nicht verhindern, dass A in den Zustand A' übergeht. Das bedeutet, dass die `LOOP`-Prozesse selbst testen müssen, ob sie die Semaphore betreten dürfen.
- Wegen oben beschriebener Konsequenzen der fehlenden harten Synchronisation und wegen fehlender Parametrisierung geht auch die Modularisierbarkeit verloren, da Teil-Prozesse jetzt genau über ihre Umgebung Bescheid wissen müssen (da sie selbst Tests vornehmen müssen).

„Check Model“ liefert keine Verletzungen der Korrektheit und Vollständigkeit (was sich aber, wie schon beschrieben, auf die syntaktische Ebene beschränkt).

2.4.3 Inkorrekte Modelle und Synchronisation in StateMate

Bei den Vorträgen der anderen Teilnehmer des Komplexpraktikums wurde mir bewusst, dass das obige Modell nicht korrekt ist. Ich hatte die Modelle immer nur im „Auto-Run“-Modus des Simulators getestet, um nicht nach jeder neuen Transition den „Play“-Knopf drücken zu müssen. Martin Peschke machte uns aber bewusst, dass es im Simulator auch möglich ist, eine erlaubte

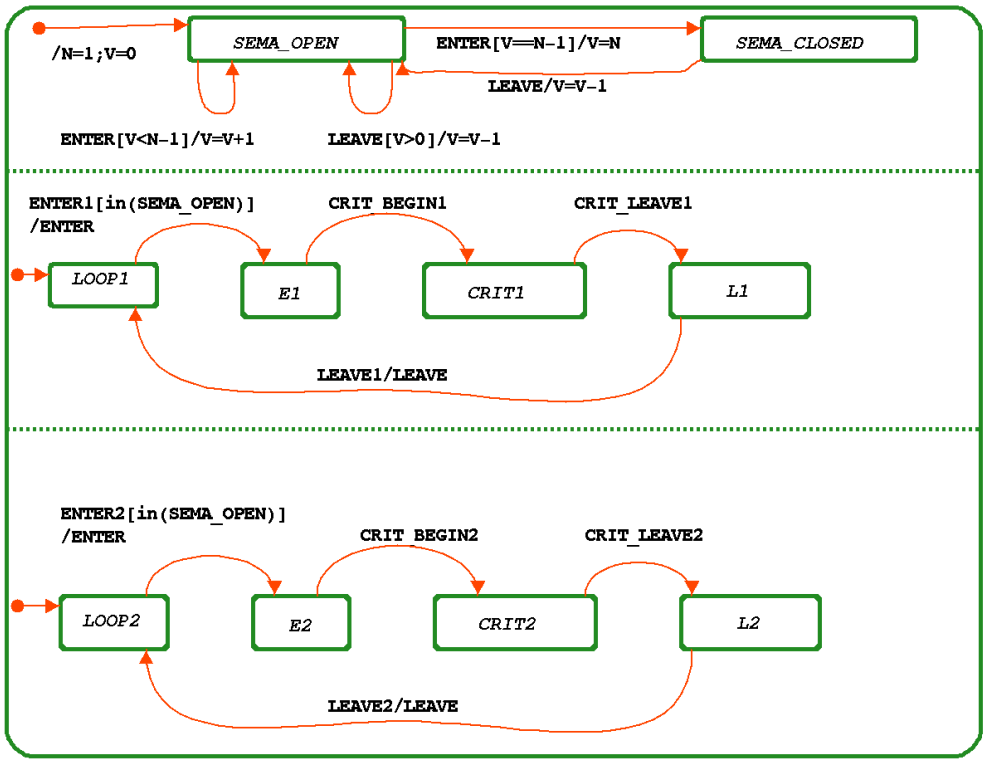

Abbildung 2.6: Semaphore in StateMate (Achtung: nicht korrekt!)

Transitions*menge* anzugeben, d. h., mehrere Ereignisse laufen gleichzeitig parallel ab. Dieses Verhalten ist an sich vernünftig und auch realistisch, nur ist es mir nicht aufgefallen. Leider tritt beim Zugriff auf Variablen (im Aktionsteil einer Transition) Interferenz auf, so dass, wenn gleichzeitig die Ereignisse `ENTER1` und `ENTER2` auftreten, der Zähler `V` nicht den Wert 2, sondern 1 bekommt.

Da jedes Sicherheits-Prinzip eine „Verankerung“ auf der tieferen Ebene benötigt, gilt das auch für die Vermeidung der Interferenz. Der einzige den Teilnehmern bekannte Mechanismus, mit dem man Interferenz in den StateCharts ausschließen kann, ist die Verwendung von queues (Warteschlangen), bei denen der Zugriff von StateMate synchronisiert wird. Das heißt, *alle* untereinander agierenden Prozess-Systeme müssen mit Hilfe von Warteschlangen implementiert werden. Das bedeutet auch, dass die FSP-Modelle prinzipiell nicht 1:1 in StateCharts übersetzt werden können.

Da ich diese Erkenntnisse erst zum Schluss des Komplexpraktikums durch die Vorträge der anderen Teilnehmer gewonnen habe, meine eigentliche Aufgabe das automatische Model Checking war, und Martin Peschke sich intensiv mit der sicheren Modellierung von Semaphoren auseinander gesetzt hat, habe ich auf eine nachträgliche korrekte Modellierung verzichtet.

Kapitel 3

Lebendigkeit

3.1 Begriffs-Definition

Lebendigkeit ergänzt ein partiell korrektes (also sicheres) System zur *totalen* Korrektheit. Umgangssprachlich ausgedrückt bedeutet das: irgendwann passiert auch einmal das Gewünschte, also etwas Gutes.

Bei sequentiellen Programmen ist die wichtigste Lebendigkeitseigenschaft, dass ein Algorithmus auch irgendwann terminiert. Nun sollen nur die wenigsten parallelen Prozess-System terminieren. Hier ist die wichtigste Lebendigkeitseigenschaft, dass ein gewünschter Ressourcen-Zugriff auch nach endlicher Zeit gewährt werden muss.

Da die komplette Behandlung von Lebendigkeit zur Beschreibung temporale Logik erfordert, wird sich hier auf das Konzept **Fortschritt** beschränkt. Fortschritt bedeutet, dass jeder Prozess nach endlicher Zeit auch eine gewünschte Transition ausführen kann, das heißt, kein Prozess „verhungert“.

Dazu braucht man noch die folgenden zwei Begriffe:

- **Faire Wahl:** Wenn eine Auswahl aus einer Transitions-Menge unendlich oft getroffen wird, dann wird jede Transition unendlich oft ausgeführt. Dies wird durch die verwendete Scheduling-Strategie bestimmt.
- **Terminale Menge:** Zustandsmenge, in der jeder Zustand von jedem anderen erreichbar ist und es keine Transition aus der Menge heraus gibt.

Mit Hilfe dieser beiden Begriffe lässt sich das Konzept „Fortschritt“ im Kalkül der endlichen Automaten beschreiben. Wenn das Modell terminale Mengen enthält und „Fortschritt“ im Sinne der jeweiligen Anwendung Transitionen erfordert, die nicht mehr in der terminalen Menge enthalten sind, dann ist die Fortschritts-Eigenschaft verletzt.

3.2 Modellierung in FSP

In FSP notiert man eine Fortschritts-Eigenschaft wie folgt:

`progress` $\{a_1, \ldots, a_n\}$

Dies stellt sicher, dass bei unendlich langer Ausführung des Systems *mindestens eine* der Aktionen a_1 bis a_n unendlich oft ausgeführt wird. Man beachte, dass dies eine *disjunktive* Menge ist. Möchte

man mehrere Aktionen fair behandeln, können beliebig viele weitere `progress`-Klauseln verwendet werden, die dann *konjunktiv* behandelt werden.

Durch die Suche nach terminalen Mengen kann LTSA Verletzungen dieser Eigenschaften entdecken.

Die Default-Annahme von LTSA ist faires Scheduling, d. h., werden terminale Mengen nicht schon durch die Automaten-Struktur absichtlich konstruiert, sind Progress-Eigenschaften immer erfüllt.

Als Beispiel mag das Werfen einer Münze dienen. Nach jedem Wurf (`toss`) folgt entweder Kopf (`heads`) oder Zahl (`tails`). Bei einer fairen Münze würde man erwarten, dass, wirft man sie unendlich oft, unendlich oft beide Ergebnisse auftreten.

```
COIN = ( toss -> heads -> COIN | toss -> tails -> COIN ).

progress HEADS = {heads}
progress TAILS = {tails}
```

Da, wie oben gesagt, LTSA per Default faires Scheduling annimmt, sind beide Fortschritts-Eigenschaften erfüllt.

Anders ist dies, wenn wir am Anfang des Prozesses heimlich eine Münze auswählen. Eine ist fair, die andere eine Trick-Münze mit „Kopf“ auf beiden Seiten:

```
COIN = ( pick -> FAIR | pick -> TRICK ),
  FAIR = ( toss -> heads -> FAIR | toss -> tails -> FAIR ),
  TRICK = ( toss -> heads -> TRICK ).

progress HEADS = {heads}
progress TAILS = {tails}
```

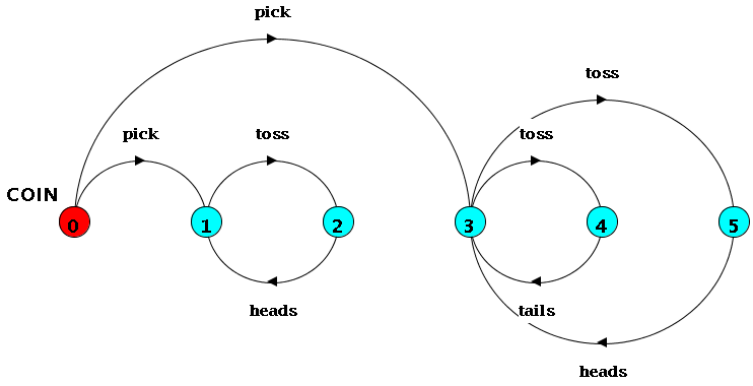

Abbildung 3.1: Graph des unfairen Münz-Wurfes mit Trick-Münze

Abbildung 3.1 zeigt den entstehenden Automaten. Überprüft man auf Fortschritt, belegt LTSA durch die Ausgabe

```
Progress violation: TAILS
Trace to terminal set of states:
        pick
Actions in terminal set:
        {heads, toss}
```

erwartungsgemäß, dass die Fortschritts-Eigenschaft für „Zahl“ verletzt ist.

Dies ist zugegebenermaßen ein unrealistisches Beispiel, da terminale Mengen normalerweise nicht konstruiert werden, sondern aus dem Zusammenspiel mehrerer Prozesse entstehen.

In realistischen Systemen findet nicht immer faires Scheduling statt. Auf einem hoch belasteten System könnten zum Beispiel pro Zeiteinheit mehr Anfragen eintreffen, als das System verarbeiten kann. Sichergestellt soll nun werden, dass auch jede Anfrage einmal beantwortet wird.

In FSP kann Hoch-Last oder andere Arten von „Stress-Tests“ durch Prioritäts-Operatoren modelliert werden, die die faire Default-Scheduling-Strategie modifizieren:

$||C = processdef$ << $\{a_1, \ldots, a_n\}$ spezifiziert einen Prozess, bei dem die angegebenen Aktionen eine höhere Priorität haben als die anderen. Das heißt, andere Transitionen außer a_1 bis a_n können nur stattfinden, wenn keine der Transitionen a_1 bis a_n verfügbar ist. Der Operator >> für niedrige Priorität verhält sich dual zu <<.

Ein ausführliches Beispiel zu deren Anwendung folgt im nächsten Kapitel.

3.3 Modellierung in StateMate

Nach eingehender Recherche der Dokumentation, der Online-Hilfe und des Internets und nach längerem Experimentieren mit StateMate selbst kam ich zu folgenden Schlüssen:

- StateMate unterstützt keine expliziten Fortschritts-Eigenschaften und Scheduling-Strategien
- StateMate kann nicht nach terminalen Mengen suchen; das ist an sich kein Problem, da man die Modelle sowieso per Hand zeichnet, sie deshalb nicht allzu groß werden können und es keine Parametrisierung gibt
- im Internet gibt es (nach einer zeitlich beschränkten Suche von etwa 30 Minuten) keine Informationen über Zusatz-Module, die Lebendigkeitseigenschaften testen können.

Das heißt, dass maschinelle Verifikation von Lebendigkeits- oder wenigstens Fortschritts-Eigenschaften mit StateMate nicht möglich ist und deshalb auch nicht durchgeführt werden konnte.

Kapitel 4

Beispiel: Single-Lane-Bridge

Dieses Kapitel beschreibt die Anwendung von „Model Checking“ anhand eines größeren Beispiels.

Das Szenario: eine Straße führt über eine Brücke mit nur einer Fahrspur. Fahrzeuge kommen beliebig von beiden Seiten. Modelliert werden soll nun das Verhalten und die Strategie einer Brücke mit folgenden Anforderungen:

- sie soll sicher sein, d. h., Fahrzeuge dürfen immer nur in *einer* Richtung fahren
- sie soll lebendig sein, d. h, selbst unter Hoch-Last (Stau) kommen alle Fahrzeuge nach endlicher Zeit herüber

4.1 Modellierung in FSP

4.1.1 Fahrzeuge und Sicherheitseigenschaft

```
const NUMCARS = 2
range ID = 1..NUMCARS
range COUNT = 0..NUMCARS

CAR = ( enter -> exit -> CAR ).

||CONVOY(N=NUMCARS) = ( [ID]:CAR ).
||CARS = ( left:CONVOY || right:CONVOY ).
```

`NUMCARS` legt die maximale Anzahl Fahrzeuge fest, die von einer Seite der Brücke kommen. Das eigentliche Fahren auf der Brücke ist uninteressant, für die Modellierung ist es hinreichend, zu wissen, wann ein Fahrzeug die Brücke betritt (`enter`) und wann es die Brücke wieder verläßt (`exit`). Der Prozess `CARS` modelliert dann schließlich die Menge aller Fahrzeuge, die momentan noch beliebig von beiden Seiten die Brücke benutzen können.

Bevor die eigentliche Brücke konstruiert wird, wird die Sicherheitseigenschaft formuliert. Diese Reihenfolge (zuerst die Anforderungen, dann die Implementierung formulieren) ist auch sehr anzuraten, ansonsten hat man immer das fertige Modell im Hinterkopf und versucht dann eher, die Anforderungen an die Implementierung anzupassen anstatt umgekehrt.

Hier wird die Sicherheitseigenschaft durch einen einfachen Zwei-Richtungs-Zähler ausgedrückt. Sobald von links ein Fahrzeug die Brücke betritt, dürfen keine Aktionen von der rechten Seite mehr erfolgen und umgekehrt:

```
property ONEWAY =
  ( left[ID].enter -> LEFTUSE[1] | right[ID].enter -> RIGHTUSE[1] ),
  LEFTUSE[i:COUNT] = ( left[ID].enter -> LEFTUSE[i+1]
                     | when (i > 1) left[ID].exit -> LEFTUSE[i-1]
                     | when (i == 1) left[ID].exit -> ONEWAY ),
  RIGHTUSE[i:COUNT] = ( right[ID].enter -> RIGHTUSE[i+1]
                      | when (i > 1) right[ID].exit -> RIGHTUSE[i-1]
                      | when (i == 1) right[ID].exit -> ONEWAY ).
```

4.1.2 Brücke

Nun wird die Brücke implementiert. Die Brücke zählt dazu mit, wieviele Fahrzeuge von links und von rechts sich gerade auf ihr befinden und nutzt dies, um nur die jeweils erlaubten Aktionen zuzulassen. Komponiert mit dem Prozess `CARS` gibt das das komplette System, die weitere Komposition mit `ONEWAY` erlaubt das Testen der Sicherheit, welchen das Modell auch besteht.

Abbildung 4.1 zeigt den entstehenden Automaten.

```
BRIDGE = BRIDGE[0][0],
  BRIDGE[l:COUNT][r:COUNT] =
    ( when (r == 0) left[ID].enter  -> BRIDGE[l+1][r]
    | left[ID].exit   -> BRIDGE[l-1][r]
    | when (l == 0) right[ID].enter -> BRIDGE[l][r+1]
    | right[ID].exit  -> BRIDGE[l][r-1] ).

||ONELANEBRIDGE = ( CARS || BRIDGE ).

||ONELANEBRIDGECHECK = ( ONELANEBRIDGE || ONEWAY ).
```

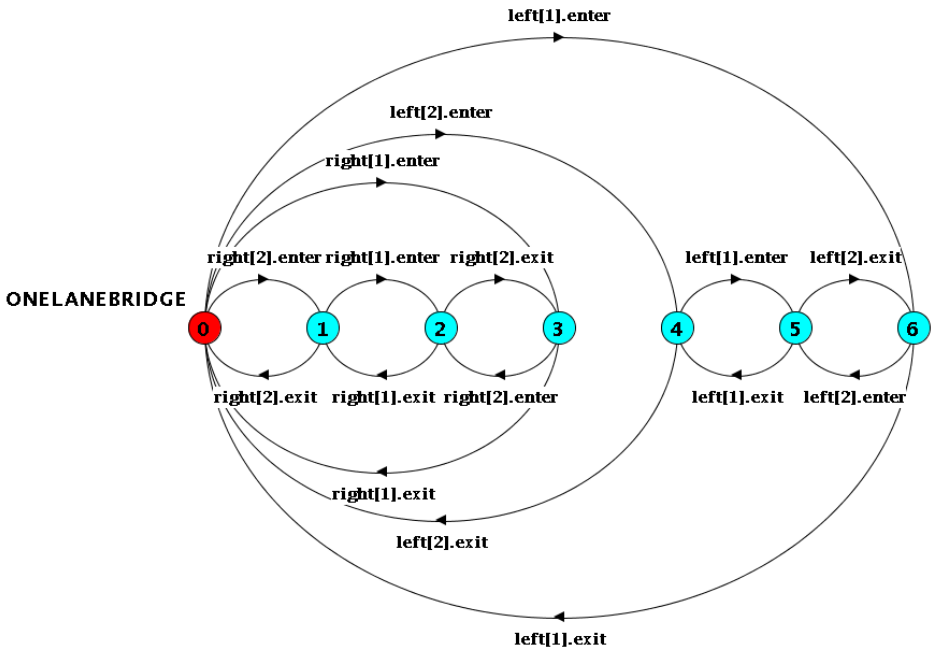

Abbildung 4.1: Automat des ersten Single-Lane-Bridge-Entwurfs

4.1.3 Lebendigkeit

Da die Brücke jetzt zertifiziert sicher ist, soll sich jetzt der Lebendigkeit zugewandt werden. Dazu wird zunächst formuliert, was „lebendig“ bei unserem System bedeuten soll:

```
progress PROGLEFT = { left[ID].enter }
progress PROGRIGHT = { right[ID].enter }
```

Wartende Fahrzeuge sollen also nicht verhungern. Diese Eigenschaften sind wegen LTSA's Default-Annahme eines fairen Schedulings erfüllt. Fügt man jedoch ein wenig Stress hinzu, d. h., lässt man Fahrzeuge schneller ankommen, als sie die Brücke passieren können (Stau), ändert sich dies:

```
||CONGESTEDBRIDGE = ( ONELANEBRIDGE ) <<{left[ID].enter,  right[ID].enter}.
```

Sowohl „Check Progress“ als auch ein prüfender Blick auf den Automaten-Graphen (Abb. 4.2) zeigt die Existenz terminaler Mengen und daher die Verletzung der Fortschritts-Eigenschaften:

```
Finding trace...
Progress violation: PROGLEFT
Trace to terminal set of states:
        right.2.enter
Actions in terminal set:
        right[1..2].{enter, exit}
Progress Check in: 36ms
```

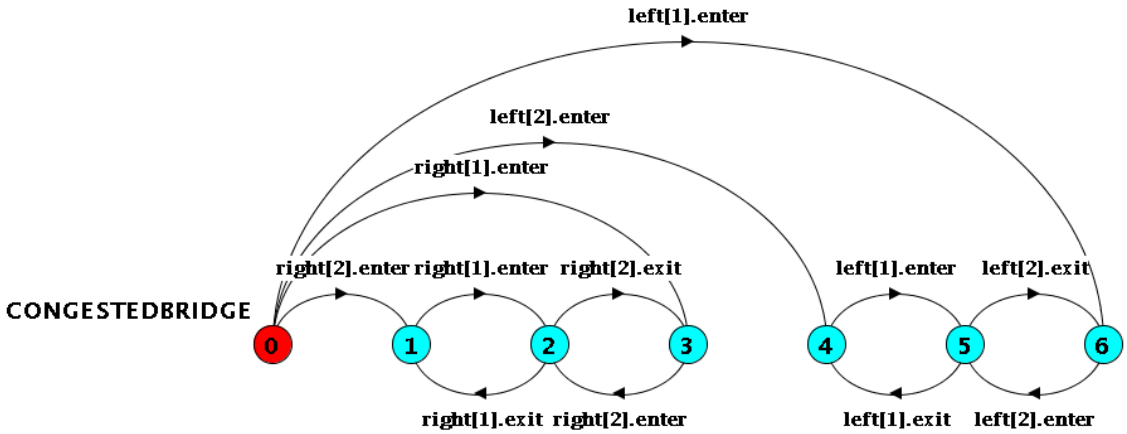

Abbildung 4.2: Single-Lane-Bridge unter Hoch-Last (Stau)

4.1.4 Lösung des Lebendigkeits-Problems, Versuch 1

Das Problem liegt in der Tatsache, dass die Brücke zu wenig Informationen bekommt: sie weiß nur, wer gerade auf der Brücke *ist*, aber nicht, wer noch darauf *möchte*. Deshalb wird das Modell eines Fahrzeugs um eine Aktion `request` erweitert, die den Wunsch, die Brücke zu überqueren, andeutet.

Die Brücke selbst zählt jetzt auch die `requests` mit und berücksichtigt diese beim Eintritt. `CONGESTEDBRIDGE` muss noch so abgeändert werden, dass auch die `request`-Operationen Priorität erhalten; dies ist äquivalent zu der unten stehenden Schreibweise, die anderen Operationen (`exit`) herab zu stufen.

```
CAR = ( request -> enter -> exit -> CAR ).

BRIDGE = BRIDGE[0][0][0][0],
  BRIDGE[l:COUNT][r:COUNT][lreq:COUNT][rreq:COUNT] =
    ( left[ID].request -> BRIDGE[l][r][lreq+1][rreq]
    | when (r == 0 && rreq == 0) left[ID].enter
                  -> BRIDGE[l+1][r][lreq-1][rreq]
    | left[ID].exit -> BRIDGE[l-1][r][lreq][rreq]

    | right[ID].request -> BRIDGE[l][r][lreq][rreq+1]
    | when (l == 0 && lreq == 0) right[ID].enter
                  -> BRIDGE[l][r+1][lreq][rreq-1]
    | right[ID].exit -> BRIDGE[l][r-1][lreq][rreq] ).

||CONGESTEDBRIDGE = ( ONELANEBRIDGE ) >>{left[ID].exit, right[ID].exit}.
```

Nun offenbart sich aber ein anderes Problem: „Check Safety“ meldet einen Deadlock, nämlich genau dann, wenn alle Fahrzeuge einen `request` angemeldet haben. Dies ist äquivalent zum Problem der Dining Philosophers.

4.1.5 Lösung des Lebendigkeits-Problems, Versuch 2

Eine einfache und bewährte Methode, einen Deadlock zu vermeiden ist das Einbringen von Asymmetrie.

Das Brücken-Modell wird also noch um ein Flag erweitert, welche Seite gerade bevorzugt wird (`LeftTurn` oder `RightTurn`). Wenn eine Seite Operationen beendet (also ein `exit`) auftritt, wird die Bevorzugung gewechselt.

```
const LeftTurn = 0
const RightTurn = 1
range WHOIS = LeftTurn..RightTurn

BRIDGE = BRIDGE[0][0][0][0][LeftTurn],
  BRIDGE[l:COUNT][r:COUNT][lreq:COUNT][rreq:COUNT][who:WHOIS] =
    ( left[ID].request -> BRIDGE[l][r][lreq+1][rreq][who]
    | when (r == 0 && (rreq == 0 || who == LeftTurn ) )
        left[ID].enter -> BRIDGE[l+1][r][lreq-1][rreq][who]
    | left[ID].exit -> BRIDGE[l-1][r][lreq][rreq][RightTurn]

    | right[ID].request -> BRIDGE[l][r][lreq][rreq+1][who]
    | when (l == 0 && (lreq == 0 || who == RightTurn))
         right[ID].enter -> BRIDGE[l][r+1][lreq][rreq-1][who]
    | right[ID].exit -> BRIDGE[l][r-1][lreq][rreq][LeftTurn] ).
```

Wie man sieht, hat unser Modell mittlerweile eine Komplexität erreicht, die das „Überprüfen durch Hinschauen“ im Quelltext und manuelle Suche im Automatengraph sehr beschwerlich machen. `ONELANEBRIDGECHECK` hat schon 736 Zustände, `CONGESTEDBRIDGE` immerhin noch 415, so dass beide vom LTSA nicht mehr dargestellt werden können. Aber die maschinelle Verifikation bescheinigt, dass dieses Modell jetzt sicher und lebendig ist und damit das Ziel der Aufgabe darstellt.

4.2 Modellierung in StateMate

Im folgenden Teil wurde versucht, die Single-Lane-Bridge auch unter StateMate zu modellieren. Da eine Überprüfung des Modells nicht möglich ist, wurde versucht, sich möglichst genau an das schon getestete FSP-Modell zu halten.

Bei den hier vorgestellten Modellen tritt der gleiche Fehler auf wie er schon in Abschnitt 2.4.3 (S. 8) beschrieben wurde. Die offenbar einzige Lösung ist auch hier eine Verwendung von Warteschlangen.

Die Modelle wurden auch hier nicht korrigiert, da das Problem erst am Ende des Praktikums offenbar wurde und das Thema meiner Aufgabe das maschinelle Model Checking war, was sich in StateMate sowieso nicht durchführen lässt.

4.2.1 Umsetzung des ersten Entwurfs

Abbildung 4.3 zeigt die Umsetzung des ersten FSP-Modells (vgl. Abschnitt 4.1) in ein StateChart. Der Entwurf lässt sich natürlich nicht 1:1 umsetzen, da in StateMate die harte Synchronisation nicht verfügbar ist. Deshalb müssen die Fahrzeuge den Test, ob sie die Brücke betreten dürfen, jetzt selbst durchführen.

Auch hier tritt das schon beobachtete Phänomen des „Aushungerns“ auf: Sobald ein Fahrzeug z. B. von der linken Seite die Brücke betritt, können von links unendlich viele Fahrzeuge die Brücke betreten und die rechts wartenden nie zum Zuge kommen lassen.

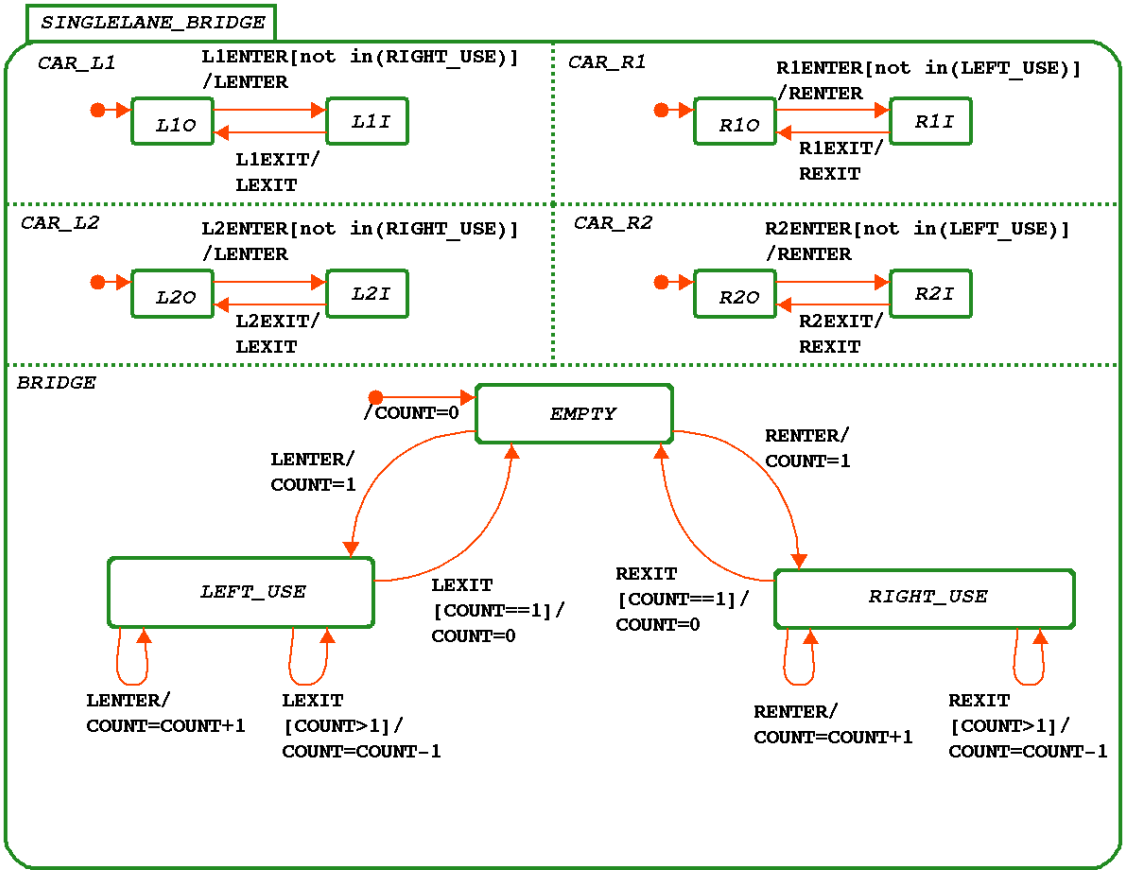

Abbildung 4.3: StateChart des ersten Entwurfes der Single-Lane-Bridge

4.2.2 Umsetzung des endgültigen Entwurfs

Bei der Arbeit mit LTSA wurde dann auch eine endgültige (sichere und lebendige) Lösung gefunden, die ebenfalls in ein StateChart umgesetzt wurde (siehe Abb. 4.4).

Das Chart sieht mittlerweile schon ziemlich umfangreich und unübersichtlich aus. Nach einer Weile Testen (mit immer nur einem Ereignis, siehe Abschnitt 2.4.3!) denke ich, dass das Modell seine Aufgabe erfüllt.

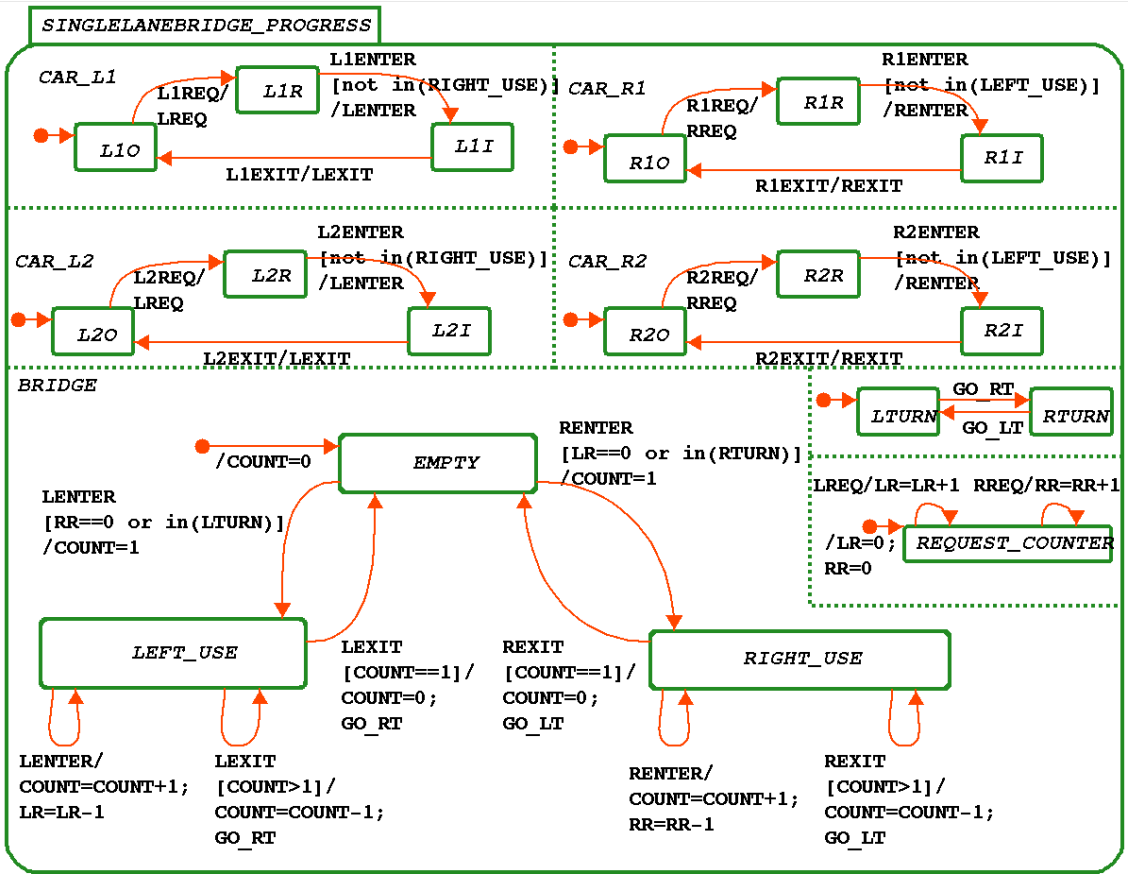

Abbildung 4.4: StateChart der endgültigen Single-Lane-Bridge

Kapitel 5

StateMate vs. LTSA

Die folgende Tabelle entstand am Ende des Praktikums, um einen *rein subjektiven* und *nur auf die Aufgabe bezogenen* Vergleich der beiden Systeme zu gewinnen. Punkte, die der Autor als Vorteil betrachtet, sind mit einem '+' gekennzeichnet, Nachteile mit einem '−'.

Aspekt	**LTSA**	**StateMate**
Einarbeitung	+ klare Programmstruktur (LTSA) − Erlernen der Sprache FSP (Konzepte, Absolvieren der Übungen) recht zeitaufwendig	+ einfache Sprache (UML) − Bedienung, Tutorial − Segfault beim Erstellen eines neuen Projekts
Fähigkeiten der Sprache	+ starke Synchronisation und Spezifikation von Interfaces erlaubt modulare Entwicklung + parametrisierbare Automaten-Struktur − Rechnen nur mit Zuständen möglich; Beschränkung auf DFA + FSP: abgeschlossene Notation − nur eine Transitions-Art; Ereignisse und Aktionen werden nicht unterschieden − nur eine einzige Synchronisations-Methode − keine Hierarchien auf Struktur-Ebene + sehr kompakte Sprache mit präzise definierter Semantik	− Einzel-Prozesse müssen an Umgebung spezifisch angepasst werden, wenig Möglichkeiten der Wiederverwendung − Automaten-Struktur ist (auch statisch) fest + intuitives Rechnen mit externen Variablen → nicht auf DFA beschränkt − StateCharts reichen zur vollständigen Beschreibung nicht aus, DataDictionary muss vorhanden sein + reiche Transitions-Sprache (Events, Actions), Verknüpfung durch Operatoren möglich − nur eine einzige Synchronisations-Methode über `queues`; unpassend für viele Aufgabenstellungen + explizite Hierarchien − Komplexität von UML ohne genaue Semantik

Modell-erstellung	+ sehr schnelle Eingabe (Text) + algebraische Sprache kann durch ein automatisch erzeugtes Diagramm visualisiert werden − bei größeren Automaten geht Übersicht verloren	− langwierige Eingabe durch Zeichnen; viel Zeit geht durch optische Gestaltung verloren + Charts sind intuitiv lesbar, Hierarchie-Bildung kann Lesbarkeit bei komplexen Diagrammen erhöhen
Model checking	+ expliziter Fehlerzustand + automatische Modell-Prüfung auf Sicherheit und Lebendigkeit + Angabe von expliziten Sicherheits- und Lebendigkeitseigenschaften	− kein Fehlerzustand − nur syntaktische Prüfung − erkennt weder terminale Zustände noch terminale Zustandsmengen
Lizenz/Preis	+ Freeware	− kommerziell, sehr teuer

Zusammenfassend lässt sich sagen, dass das Arbeiten mit LTSA *für diese Aufgabe* wesentlich klarer und einfacher war, und Model Checking mit StateMate überhaupt nicht durchführbar war. Dies soll aber auf keinen Fall einer objektiven Bewertung gleichkommen; es gab auch Teilnehmer des Praktikums (mit anderen Problemstellungen), die ganz andere Erfahrungen gemacht haben.

Literaturverzeichnis

[MaKr] Jeff Magee, Jeff Kramer: „Concurrency – State Models & Java programs“; Wiley-Verlag, 1999

[LTSA] Labeled Transition System Analyzer
(`http://www-dse.doc.ic.ac.uk/concurrency/`)

[STM] StateMate -Dokumentation, insbesondere „Modelling reactive systems“

Über dieses Dokument

Dieses Dokument wurde mit dem Textsatzsystem LaTeX 2_ε erstellt. Es kann frei verwendet und kopiert werden. Sollten Sie einen Fehler finden, bitte ich darum, mir eine kurze Nachricht an `martin@piware.de` zu senden. Unter selbiger eMail-Adresse können bei Interesse auch die LaTeX-Quellen angefordert werden.