Übungen zur Vorlesung
Testautomatisierung
Sommersemester 2004

AG BS
Jan Peleska, Stefan Bisanz,
Aliki Tsiolakis

Blatt 4

Revision: 1.10

Test-Suite zum Testen eines zustandsbasierten Systems

Diese Aufgabenserie behandelt erneut die Komponente *PLS (Passenger Lighted Signs)*. Als Erweiterung der vorherigen Serien wird nun die Teilfunktion *PRAM (Pre-recorded Announcement)* berücksichtigt. Damit handelt es sich bei PLS nicht mehr um ein rein kombinatorisches, sondern um ein *zustandsbasiertes* System.

Die Komponente hat (zusätzlich zur Beschreibung aus Aufgabenserie 2) weitere Ausgabekanäle: **pramOn[0..31]** steuert die Ausgabe der Lautsprecheransage an der entsprechenden Adresse. Dafür wird angenommen, dass an jeder Adresse zusätzlich zu den Anzeigen ein Lautsprecher angeschlossen sein kann. An welcher Adresse tatsächlich ein Lautsprecher freigeschaltet ist, wird durch den zusätzlichen Parameter **paramPramAssigned[0..31]** gesteuert. Dieser funktioniert genau so, wie die anderen Zuordnungstabellen aus Serie 2.

Folgender neuen Anforderung muss PLS gerecht werden: Immer wenn die Anschnallzeichen aktiviert oder deaktiviert werden (also bei jedem *Wechsel*) soll an den entsprechenden Adressen eine allgemeine Ansage (z.B. "Achtung!") erfolgen. Dabei wird **pramOn[i]** von PLS auf `true` gesetzt, um die Ansage zu starten, und nach einer definierten Zeit t_{pram} wieder auf `false` gesetzt, um die Ansage zu beenden. Entsprechend sollen Ansagen für die *return-to-seat-signs* aktiviert werden. Ausgangszeichen sind davon nicht betroffen. Somit müssen zweierlei Zustände berücksichtigt werden:

- Der aktuelle Zustand der Anzeigen (aktiviert/deaktiviert).
- Der Zustand der Lautsprecheransagen.

Alle weiteren Anforderungen (s. Aufgabenserie 2) behalten ihre Gültigkeit.

Die *Spezifikation* des zustandsbasierten PLS besteht im wesentlichen aus vier flachen parallelen Statecharts: FSB (s. Abb. 1), RTS (s. Abb. 2) und EXIT (s. Abb. 3) bestehen jeweils aus zwei Zuständen, in denen gespeichert wird, ob die jeweiligen Anzeigen gerade aktiviert sind oder nicht.

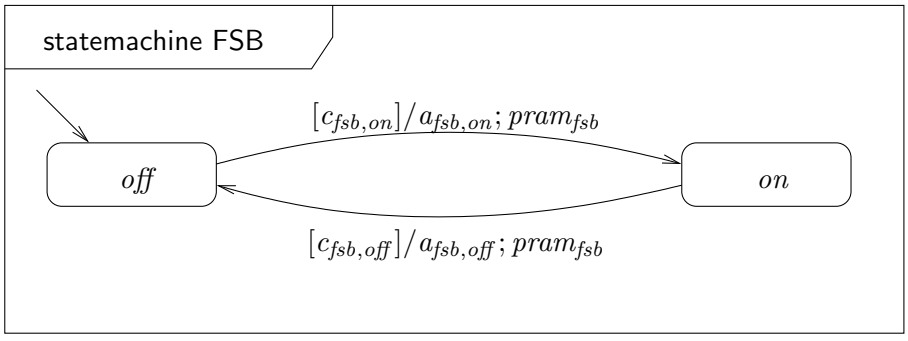

Abbildung 1: Statechart FSB.

Die Zustandsübergänge dazwischen werden über Wächter[1] gesteuert, die direkt aus der kombinatorischen Spezifikation abgeleitet sind:

$$c_1 \quad = \quad paramDecompActivation \wedge (pressureLow1 \vee pressureLow2)$$

[1] auch *guards* oder *conditions* genannt

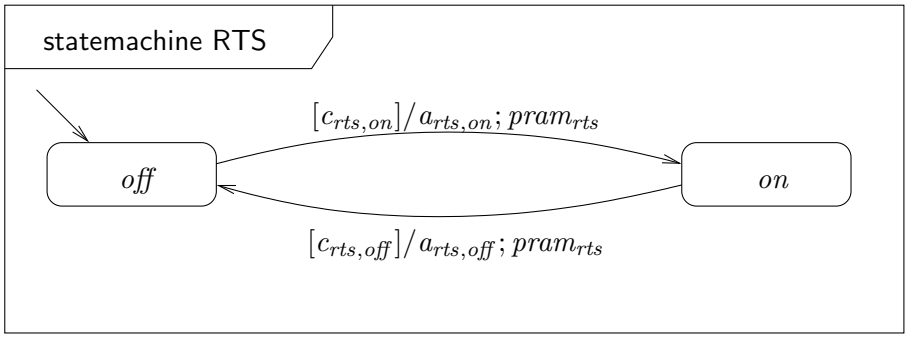

Abbildung 2: Statechart RTS.

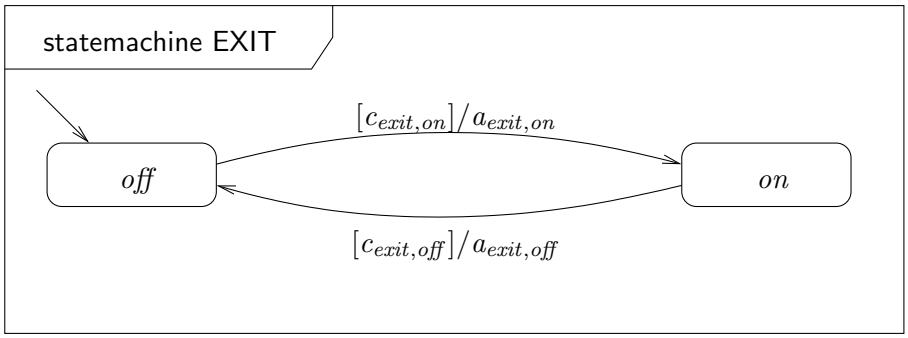

Abbildung 3: Statechart EXIT.

$$
\begin{aligned}
c_2 &= (\neg paramDecompActivation \vee (\neg pressureLow1 \wedge \neg pressureLow2)) \\
&\quad \wedge fsbSwitchOn \\
c_3 &= (\neg paramDecompActivation \vee (\neg pressureLow1 \wedge \neg pressureLow2)) \\
&\quad \wedge \neg fsbSwitchOn \wedge \neg fsbSwitchAuto \\
c_4 &= (\neg paramDecompActivation \vee (\neg pressureLow1 \wedge \neg pressureLow2)) \\
&\quad \wedge \neg fsbSwitchOn \wedge fsbSwitchAuto \\
&\quad \wedge paramAutoSolutionType = SOLUTION1 \\
&\quad \wedge (ldgDownLocked \vee slatActive1 \vee slatActive2) \\
c_5 &= (\neg paramDecompActivation \vee (\neg pressureLow1 \wedge \neg pressureLow2)) \\
&\quad \wedge \neg fsbSwitchOn \wedge fsbSwitchAuto \\
&\quad \wedge paramAutoSolutionType = SOLUTION1 \\
&\quad \wedge \neg ldgDownLocked \wedge \neg slatActive1 \wedge \neg slatActive2 \\
c_6 &= (\neg paramDecompActivation \vee (\neg pressureLow1 \wedge \neg pressureLow2)) \\
&\quad \wedge \neg fsbSwitchOn \wedge fsbSwitchAuto \\
&\quad \wedge paramAutoSolutionType = SOLUTION2 \wedge ldgDownLocked \\
c_7 &= (\neg paramDecompActivation \vee (\neg pressureLow1 \wedge \neg pressureLow2)) \\
&\quad \wedge \neg fsbSwitchOn \wedge fsbSwitchAuto \\
&\quad \wedge paramAutoSolutionType = SOLUTION2 \wedge \neg ldgDownLocked
\end{aligned}
$$

$c_1 \ldots c_7$ entsprechen den Bedingungen über die Eingaben und Parameter aus den Anforderungen R001. . . R007, $c_2 \ldots c_7$ jeweils zusätzlich jener aus R008 (s. Aufgabenserie 2). Daraus lassen sich für jede Anzeigensorte je zwei Bedingungen zusammensetzen, welche dem Aktivieren und dem Deaktivieren

entsprechen:

$$
\begin{aligned}
c_{fsb,on} &= c_1 \vee c_2 \vee c_4 \vee c_6 \\
c_{fsb,off} &= c_3 \vee c_5 \vee c_7 \\
c_{exit,on} &= c_1 \\
c_{exit,off} &= c_2 \vee c_3 \vee c_4 \vee c_5 \vee c_6 \vee c_7 \\
c_{rts,on} &= c_2 \vee c_4 \vee c_6 \\
c_{rts,off} &= c_1 \vee c_3 \vee c_5 \vee c_7
\end{aligned}
$$

Jeder Zustandsübergang führt dann die entsprechende Aktion aus, um die entsprechende Ausgabe zu setzen (d.h. die zugehörigen Anzeigen zu aktivieren bzw. zu deaktivieren):

$$
\begin{aligned}
a_{fsb,on} &= \forall i \in \{0..31\} \bullet fsbSignOn[i] = paramFsbSignAssigned[i] \\
a_{fsb,off} &= \forall i \in \{0..31\} \bullet fsbSignOn[i] = false \\
a_{exit,on} &= \forall i \in \{0..31\} \bullet exitSignOn[i] = paramExitSignAssigned[i] \\
a_{exit,off} &= \forall i \in \{0..31\} \bullet exitSignOn[i] = false \\
a_{rts,on} &= \forall i \in \{0..31\} \bullet rtsSignOn[i] = paramRtsSignAssigned[i] \\
a_{rts,off} &= \forall i \in \{0..31\} \bullet rtsSignOn[i] = false
\end{aligned}
$$

Weiterhin erzeugen die Automaten FSB und RTS jeweils ein *Signal* $pram_{fsb}$ bzw. $pram_{rts}$, welches an das Statechart PRAM (s.u.) gesendet wird, um die Lautsprecheransage zu aktivieren. Im Automaten EXIT geschieht dieses nicht.

Der Automat PRAM (s. Abb. 4) verwaltet die Aktivierung/Deaktivierung der Lautsprecheransagen. Sofern alle Lautsprecheransagen deaktiviert sind (*off*), wartet er auf die eingehenden Signale $pram_{fsb}$ bzw. $pram_{rts}$, um beim Eintreffen die Lautsprecheransagen überall dort zu aktivieren, wo Anzeigen der zugehörigen Sorte *und* Lautsprecher angeschlossen sind (Übergang zu *fsbActive* bzw. *rtsActive*). Dabei wird ein *Timer* gesetzt, welcher bestimmen soll, wann die Ansage beendet wird. Dieses geschieht beim Empfang des Ablaufsignals. Wird die Ansage zuvor erneut angestossen, so wird der Timer neu gesetzt und somit die Ansagedauer verlängert. Der Zustand *bothActive* kodiert die Situation, daß sich die Aktivierungen für FSB und RTS überlagern. So können beim Übergang von *fsbActive* bzw. *rtsActive* nach *off alle* Ansagen beendet werden, beim Übergang von *bothActive* nach *fsbActive* bzw. *rtsActive* werden nur die zugehörigen Ansagen deaktiviert. Die enthaltenen Aktionen der Transitionen sind folgendermassen definiert:

$$
\begin{aligned}
a_{pram,fsbon} &= \forall i \in \{0..31\} \bullet pramOn[i] = paramFsbSignAssigned[i] \wedge paramPramAssigned[i] \\
a_{pram,rtsbon} &= \forall i \in \{0..31\} \bullet pramOn[i] = paramRtsSignAssigned[i] \wedge paramPramAssigned[i] \\
a_{pram,on} &= \forall i \in \{0..31\} \bullet pramOn[i] = \\
&\quad paramPramAssigned[i] \wedge (paramFsbSignAssigned[i] \vee paramRtsSignAssigned[i]) \\
a_{pram,off} &= \forall i \in \{0..31\} \bullet pramOn[i] = false
\end{aligned}
$$

Zur Erstellung der Test-Suite wird wie üblich ein Projektrahmen für den RT-Tester als Archiv zur Verfügung gestellt (`ta-project3.tgz`).

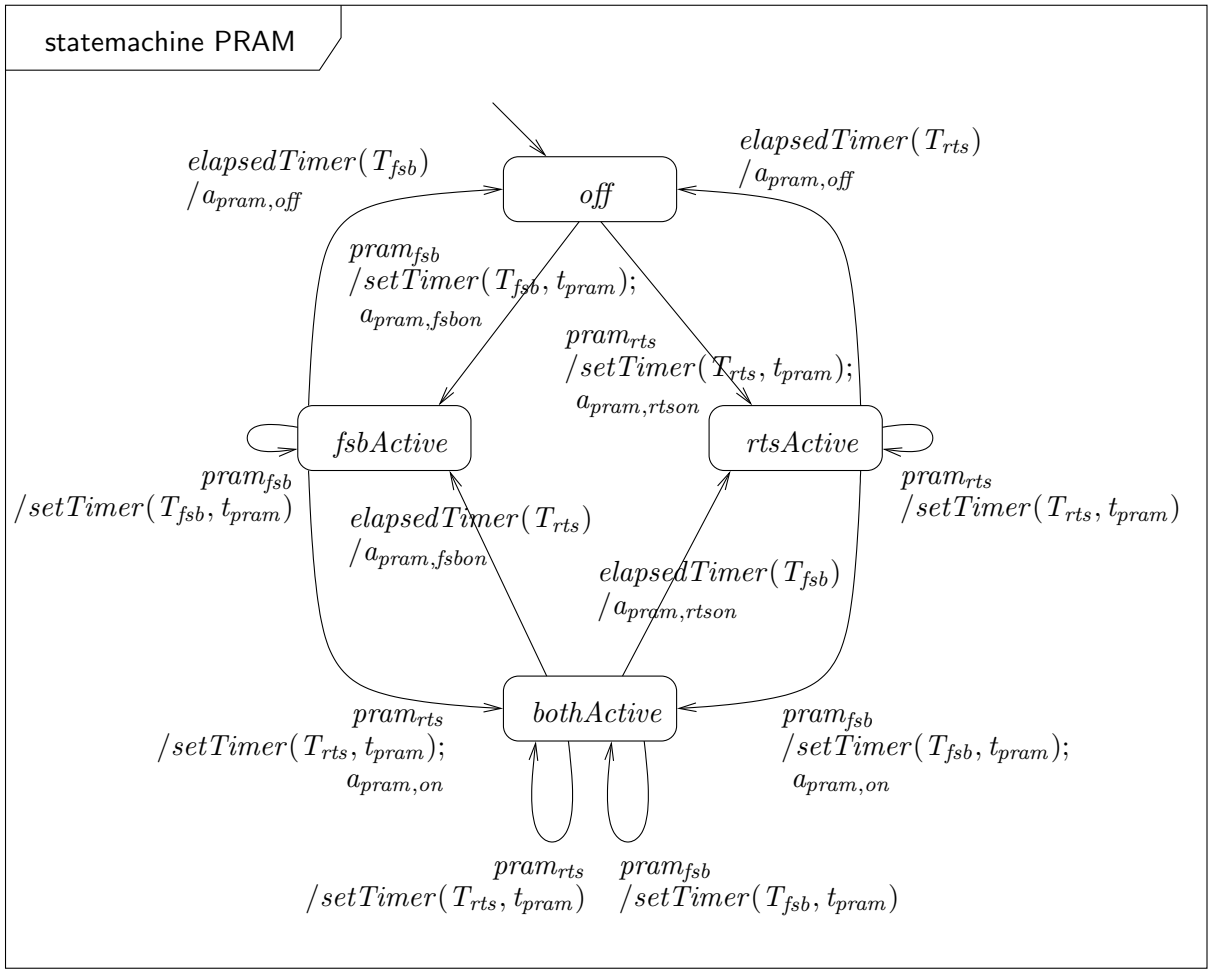

Abbildung 4: Statechart PRAM.

Aufgabe 1: Implementierung von flachen Statecharts **50%**

Grundlage des zustandsbasierten Testens ist die Implementierung eines abstrakten Datentyps *Statechart* in Form der folgenden C-Datenstrukturen (und den zugehörigen Operationen):

struct scstatechart Ein komplettes Statechart.

struct scstate Ein Zustand (State, Location).

struct sctransition Eine Transition.

struct scsignal Ein Signal (Ereignis, Event).

struct scguard Ein Wächter (Guard, Condition).

struct scaction Eine Aktion.

struct sctimer Ein Timer.

struct sctimerlist Datenstruktur für die Verwaltung von mehreren Timern.

Dabei sind `struct sctimer` und `struct sctimerlist` strenggenommen kein direkter Bestandteil des Datentyps; sie sind allerdings eng damit verknüpft, indem sie direkt durch spezielle Aktionen aus dem Statechart angestossen werden und spezielle Signale beim Ablauf in das Statechart schicken.

Teilaufgabe 1: Definition der Operationen

Implementieren Sie die spezifizierten Operationen zu den Datenstrukturen `struct scstatechart`, `struct scstate`, `struct sctransition`, `struct scsignal`, `struct scguard` und `struct scaction`. Die Deklarationen sind in den entsprechenden C-Header-Dateien im Verzeichnis `sclib` in `ta-project3` enthalten. Die zugehörigen Datenstrukturen selbst sind darin schon definiert. (`struct sctimer` und `struct sctimerlist` sind schon komplett definiert, d.h. sowohl die Datenstruktur als auch die Operationen.)

Teilaufgabe 2: Test der Implementierung

Testen Sie Ihre Implementierung (unformal), indem Sie eine geeignete Methode `int main (int, char**);` bereitstellen, welche die Implementierung des Datentyps prüft. Dokumentieren Sie deren Ausgaben.

Aufgabe 2: Testen des zustandsbasierten Systems 50%

Mit Hilfe der in Aufgabe 1 erstellten Bibliothek `sclib` kann das zustandsbasierte System nun getestet werden. Für jedes der vier parallelen Statecharts wird ein separater Checker erstellt, der das jeweilige Teilverhalten prüft. Die Vorgehensweise pro Checker ist wie folgt:

Innerhalb des Checkers wird das zugehörige Statecharts simuliert. Dafür wird die Datenstruktur `struct scstatechart` instanziiert und initial der Startzustand betreten. Danach soll darauf gewartet werden, dass eine ausgehende Transition möglich wird und diese dann sofort (wg. der "urgency") feuern. Weil PLS ein deterministisches System ist, kann angenommen werden, dass immer nur eine Transition zur Zeit freigeschaltet ist.[2] Nachdem die Transition gefeuert hat, geht es im Folgezustand entsprechend weiter.

Somit liefert die Simulation das *Sollverhalten* in Abhängigkeit der tatsächlichen Eingaben bei der Testausführung. Immer, wenn eine Transition in der Simulation feuert, muss nun geprüft werden, ob das *tatsächliche Verhalten* des Testlings dem entspricht. Da das *beobachtbare Verhalten* von PLS gerade den Ausgaben **fsbSignOn[]**, **rtsSignOn[]**, **exitSignOn[]** und **pramOn[]** entspricht, muss der Checker nun die zugehörigen Ausgaben des Testlings und die erwarteten Ausgaben gemäß der Simulation vergleichen, d.h. der fsb-Checker prüft die Ausgaben **fsbSignOn[]**, etc. Dabei muss dem Testling (wie auch bei den kombinatorischen Systemen) eine Reaktionszeit (von 100ms) eingeräumt werden.

Anpassung des Testvorgehens: Weil die Spezifikation in statemachine PRAM nicht vollständig deterministisch ist (s.o.), wird der Test nun folgendermaßen durchgeführt: Während die Simulation in einem frisch betretenen Zustand auf die passende Reaktion des Testlings wartet, wird nun *parallel dazu* geprüft, ob die Simulation schon einen weiteren Schritt machen könnte (d.h. ob eine Transition freigeschaltet ist). Ist das der Fall, dann wird der aktuelle Testschritt *abgebrochen* und liefert somit kein Resultat. Der nächste Simulationsschritt wird sofort durchgeführt. Daraufhin wird ein neuer Testschritt begonnen, der die Ausgaben von Simulation und Testling vergleicht, etc. Somit ist es gleichgültig, ob die Simulation von statemachine PRAM den Pfad *off* → *fsbActive* → *bothActive* oder *off* → *rtsActive* → *bothActive* beschreitet, weil schließlich nur das Gesamtresultat beider Transitionen betrachtet wird.[3]

[2] Tatsächlich ist PLS in der vorliegenden Modellierung *nicht* deterministisch, weil die Reihenfolge der Ereignisse $pram_{fsb}$ und $pram_{rts}$ unterschiedlich sein kann, wenn beide (quasi) gleichzeitig auftreten. Diese Reihenfolge bestimmt insbesondere, welcher der Pfade *off* → *fsbActive* → *bothActive* und *off* → *rtsActive* → *bothActive* in statemachine PRAM simuliert wird. Das muss nicht zwangsläufig damit übereinstimmen, welche Ausführung der Testling implementiert. Daraus folgt eine *Anpassung des Testvorgehens*, s. im folgenden.

[3] *Beachte:* Dieses ist nur ein Workaround für die spezielle PLS-Spezifikation. Er erlaubt allerdings, den relativ einfachen Prüfalgorithmus für deterministische Systeme anstelle eines deutlich aufwändigeren für nicht-deterministische Systeme anzuwenden.

Für die Generierung der Testdaten wird die Kombinationsüberdeckung aus Aufgabenserie 2 verwendet. Die Parameter werden nicht variiert, d.h. es ist nur eine Testprozedur nötig. Diese ist komplett, d.h. Generator und SUT, in `test1` von `ta-project3` vorgegeben.

Weiterhin ist eine Abstrakte Maschine `timeservice` im Verzeichnis `specs` vorgegeben, welche die Verwaltung der Timer realisiert. Dort werden insbesondere die für die Spezifikation von PLS benötigten Timer T_{fsb} und T_{rts} bereitgestellt.

Teilaufgabe 1: Erstellung der Testprozedur

Erstellen Sie die vier separaten Checker gemäß der obigen Beschreibung. Ein Rahmen `stateChecker` existiert im Verzeichnis `specs`.

Beachten Sie bei der Initialisierung der Statecharts, dass manche Elemente initialisiert sein müssen, bevor die einzelnen Checker ihre weiteren Initialisierungsschritte durchführen. Teilen Sie die Initialisierung derart auf, dass die erste Phase in einem der Checker im Abschnitt `@INIT:` und die zweite Phase am Anfang des jeweiligen Abschnitts `@PROCESS:` durchgeführt werden.

Geben Sie am Ende der jeweiligen Checker-Ausführung die *State Coverage* sowie die *Transition Coverage* für das zugehörige Statechart aus. Die Datenstruktur aus `sclib` ist dafür schon vorbereitet, so dass für die State Coverage pro Zustand die Anzahl der Besuche sowie für die Transition Coverage pro Transition die Häufigkeit des Feuerns direkt ausgelesen werden kann.

Dokumentieren Sie die Testfälle in den Testprozeduren, so dass diese nach der Testdurchführung mit in den Log-Dateien erscheinen. Alle relevanten Informationen zum Test müssen daraus hervorgehen. Das schließt beim Test zustandsbasierter Systeme sowohl den (angenommenen) Zustand des Testlings als auch die jeweils feuernde Transition ein.

Teilaufgabe 2: Durchführung der Testprozeduren

Führen Sie die in Teilaufgabe 1 erstellten Testprozeduren mit dem RT-Tester 6.0 aus.

Abgabe: Bis Montag, 12. Juli 2004, im Tutorium.

Geben Sie alle Aufgabenlösungen sowohl (1) *ausgedruckt* (in MZH8210, MZH8170) als auch (2) *elektronisch* (http://www.informatik.uni-bremen.de/~alien/upload.html) ab.

In allen Dokumenten und Dateien die Namen aller Gruppenmitglieder nicht vergessen!

Übersicht der Abgaben:

Aufg.	Schriftliche Abgabe	Elektronische Abgabe
1	alles	alles (eingebettet in das RTT-Projekt, s. 2)
2	pro Testprozedur: *Real-Time Test Specification*-Dateien (`specs/*.rts`), Konfigurationsdatei (`conf/*.conf`), alle evtl. geänderten Dateien mit neuen Stubs, Typen o.ä. (Ausdruck gerne auch mit a2ps)	Das gesamte Projekt inklusive der Log-Dateien in allen Testprozeduren sowie des `code-coverage`-Verzeichnisses

Die elektronische Abgabe bitte als Archiv (.tar.gz, .tar.bz2 oder .zip), inkl. einer .ps- oder .pdf-Version der gedruckten Abgabe!