Nebenläufige Programmierung

Dieses Skript ist – genau wie die Vorlesung, die es begleitet – ein Experiment. Es hat sicher diverse Fehler und Schwächen, ist garantiert unvollständig, und die Qualität der einzelnen Kapitel wird schwanken.

Dass es dieses Skript überhaupt gibt, und dass wir sogar zumindest zur Zeit darauf recht stolz sind, ist der besondere Verdienst einer Reihe von Studenten, und zwar: Christian Doczkal, Ernst Moritz Hahn, Clemens Hammacher, Arnd Hartmanns, Markus Rabe, Sigurd Schneider, und David Spieler.

Außerdem beigetragen haben, in der einen oder anderen Form: Andreas Abel, Steffen Feidt, Steffen Heil, Nils Jasper, Björn Kunz, Michael Kunz, Daniel Günter Lorig, Gregor-Ulrich Mehlmann, Quan Nguyen, Tobias Orth, Andi Scharfstein, Martin Schreiber, Benjamin Weis und Alexander Wieder.

Wir werden das Skript nach und nach zur Verfügung stellen, und werden auch verbesserte Versionen einzelner Kapitel veröffentlichen, sobald verfügbar. Sie können uns (und sich selbst) dadurch helfen, dass Sie das Material jeweils genau studieren, und uns Fehler und andere Kommentare mitteilen. Fehler gibt es in jedem Fall zu finden, dafür ist in jedem Kapitel gesorgt.

Wir bedanken uns insbesondere auch für die Bereitstellung eines kleinen Topfs von Studiengebühren, durch die wir die Arbeit an diesem Werk und der Konzeption der Vorlesung auf mehrere Schultern verteilen konnten.

Holger Hermanns, Christian Eisentraut

Kapitel 1

Prozesse

Systeme der Informatik berechnen selten eine einzelne Ausgabe auf der Basis einer konkreten Eingabe. Stattdessen kommunizieren sie mit anderen Systemen. Ein Beispiel ist ein Mobiltelefon. Zum einen kommuniziert es mit seinem Benutzer, indem dieser verschiedene Befehle über die Tastatur (oder den Touch-Screen) des Mobiltelefons eingibt. Zugleich kommuniziert es mit Satelliten, die verschiedene Daten übermitteln. Meist besteht das, was wir als ein einzelnes System betrachten, selbst wiederum aus einer Vielzahl von Komponenten, oft *Prozesse* genannt, die miteinander kommunizieren. Diese Komponenten sind größtenteils unabhängig voneinander, teilweise interagieren sie jedoch, um gemeinsam etwas zu erreichen. Man spricht hier von nebenläufigen (engl. *concurrent*) Prozessen. `TODO: redundantes Material aufräumen` Weitere Beispiele sind allgegenwärtig: eine Datenbank von Google, das GPS-Satellitennavigationssystem, ein gewöhnliches Betriebssystem, in dem mehrere Programme gleichzeitig ausgeführt werden können, eine CPU im Zusammenspiel mit einem 3D-Graphikprozessor, ein Pulsmesser und die entsprechende Uhr mit Empfänger am Handgelenk. Man spricht zusammenfassend auch von reaktiven Systemen.

TODO

Die einzelnen Prozesse eines Systems führen verschiedene Aktivitäten aus, die zumeist prozedurale Berechnungen im bekannten Sinne sind. Wir können diese beispielsweise als Prozeduren in SML adäquat repräsentieren. Zudem müssen die Prozesse aber auch Kommunikationsaktivitäten ausführen, um Informationen mit anderen nebenläufigen Prozessen auszutauschen, und sich zu koordinieren.

In der Praxis ist es heutzutage nur noch sehr selten der Fall, dass ein System der Informatik zunächst eine komplette Eingabe liest, dann diese auf einer einzelnen Recheneinheit sequentiell verarbeitet, um dann zum Schluss dem Benutzer die Ergebnisse zu präsentieren. Stattdessen finden in der Regel die Ein- und Ausgaben nicht nur zu Beginn und am Ende der Laufzeit eines Informatiksystems statt, sondern geschehen über die Laufzeit verteilt. Viele dieser Systeme sollen darüberhinaus gar nicht nach endlicher Zeit mit einem Ergebnis terminieren. Es ist viel mehr dem Benutzer überlassen diese zu terminieren, *Runterfahren* sagen der Laie und der Fachmann dazu.

Um eine bestimmte Aufgabe zu erledigen, bedarf es in vielen Fällen der Koordination und Kommunikation zwischen verschiedenen Informatiksystemen, um etwa eine gemeinsame Auf-

gabe zu erfüllen. [1] Ein konkretes Beispiel hierfür ist ein Mobiltelefon. Dies kommuniziert zum einen mit seinem Benutzer, indem dieser verschiedene Befehle über die Tastatur (oder den Touch-Screen) des Mobiltelefon eingibt und über dessen Bildschirm Informationen über den Status des Telefons entnehmen kann. Zugleich kommuniziert es mit Sendemasten (genauer: mit Mobilfunksendeanlagen), um Daten verschiedener Art (Sprache, E-Mails, usw.) zu übertragen.

Fast immer besteht das, was nach außen als ein einzelnes System erscheint, aus einer Vielzahl von Komponenten, oft als Prozesse bezeichnet, die miteinander kommunizieren. Diese Komponenten sind größtenteils unabhängig voneinander und laufen oft gleichzeitig ab, teilweise interagieren sie jedoch, um gemeinsam das nach *außen sichtbare Verhalten* des Gesamtsystems zu bewirken. Man spricht hier von nebenläufigen (engl. *concurrent*) Prozessen.

Weitere Beispiele sind allgegenwärtig:

- verteilte Datenbanken und Suchmaschinen,
- GPS-Navigationssysteme,
- Pulsmesser mit Uhr und Empfänger am Handgelenk,
- Eisenbahnverkehrsleitsysteme,
- Electronic-cash Systeme und Geldautomaten,
- LKW-Mautsysteme auf Autobahnen.

Man spricht zusammenfassend auch von reaktiven Systemen. Ein Prozess in einem reaktiven System interagiert mit (oder reagiert auf) seine Umgebung. Er führt dabei typischerweise verschiedene lokale Aktivitäten aus, die zumeist prozedurale Berechnungen im bekannten Sinne sind. Wir können diese beispielsweise als Prozeduren in SML adäquat repräsentieren. Zudem führen die Prozesse aber auch Kommunikationsaktivitäten aus, wenn sie Informationen mit anderen nebenläufigen Prozessen austauschen.

Beispiel 1. Abbildung 1 zeigt ein Beispielsystem, welches aus einem Handy und einem Sendemast besteht. Wir können dieses System als ein reaktives System betrachten, da ein Handy normalerweise durchgehend angeschaltet ist und über einen Sendemast Kontakt zu den jeweiligen Funknetzen hält und nur dann wirklich aktiv wird, wenn Eingaben vom Benutzer kommen oder beispielsweise eine SMS empfangen wird.

Es ist auch offensichtlich, dass sowohl Handy als auch der Sendemast unabhängig voneinander Aktivtäten ausführen können. So könnte das Handy zum Beispiel vom Benutzer zum *Schnuffel*-Spielen verwendet werden und der Sendemast gleichzeitig mit dem Mobiltelefon von Dieter Schlau kommunizieren. Wenn wir beide Systeme zusammen betrachten, handelt sich also um ein nebenläufiges reaktives System.

[1] Dies passt offensichtlich nicht exakt zum klassischen Berechnungsmodell der Informatik, welches davon ausgeht, dass eine Maschine mit Hilfe eines Programms aus einer Menge von Eingaben eine Menge von Ausgaben erzeugt. Dieses Modell, welches auf John von Neumann und Alan Turing zurückgeht, ist nichtsdestotrotz eine wichtige Grundlage auch für nebenläufige Prozesse.

Beispiel 2. Auch der Treiber eines Graphik-Prozessors in einem High-End PC kann als Teil eines nebenläufigen reaktiven Systems betrachtet werden. Er kommuniziert typischerweise mit dem Betriebssystem, und führt lokale Berechnungen durch, deren Ergebnisse an den Bildschirm(-treiber) weitergereicht werden.

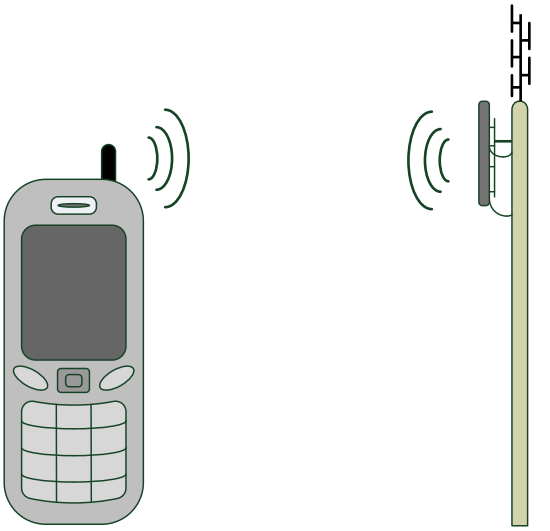

Abbildung 1.1: Handy und Sendemast interagieren.

1.1 Aktionen und Prozesse

Auf unserer Suche nach Erkenntnis betrachten wir keine konkreten Systeme, wie die aus den obigen Beispielen, sondern führen ein allgemeines, abstraktes Konzept ein, das erlaubt, das Verhalten solcher Systeme zu verstehen, ... oder zumindest zu untersuchen.

Zunächst halten wir fest, dass reaktive Systeme aus Prozessen bestehen, und diese Prozesse Aktivitäten ausführen können. Die verschiedenen Aktivitäten eines Prozesses nennen wir im Folgenden *Aktionen*. Wie wir oben schon gesehen haben, gibt es lokale Aktivitäten, bei denen es sich meist um prozedurale Berechnungen handelt, und Kommunikationsaktivitäten. Wenn wir das Verhalten von nebenläufigen Systemen betrachten wollen, dann spielt es im Allgemeinen keine Rolle, wie die lokalen Berechnungen konkret ablaufen. Es ist nur von Bedeutung, welches Verhalten von den anderen Prozessen wahrgenommen werden kann. Wir können uns also vorstellen, dass das Auftreten einer bestimmten Aktion signalisiert, dass ein bestimmter Vorgang gerade passiert und dass dieser Vorgang von anderen Prozessen wahrgenommen werden kann. Wir abstrahieren dabei jedoch vollständig von den tatsächlichen Berechnungen, die zur Durchführung des Vorgangs nötig sind. Deswegen genügt es auch, Aktionen einfach als ein Menge von Namen darzustellen, die keine weitere Struktur haben. Da nun alle Aktionen einen mitteilenden Charakter haben, müssen wir nicht zwischen lokalen (Berechnungs-)Aktivitäten und Kommunikationsaktivitäten unterscheiden.

Die Menge aller Aktionen Act wird neben den soeben beschriebenen *Kommunikationsaktionen* auch eine besondere Aktion τ enthalten. Sie steht für alle Aktivitäten, die intern in einen

Prozess passieren, die jedoch kein anderer Prozess von außen genauer bestimmen kann. Das Auftreten dieser Aktion signalisiert, *dass* eine Aktivität passiert, jedoch nicht, *was* im Inneren des Prozesses passiert. Deswegen gibt es auch nur eine einzige Aktion τ, die verschieden von allen anderen (genau beobachtbaren) Aktionen ist. Während alle anderen Aktionen in unserer Abstraktion reine Kommunikationsaktionen sind, kann man diese Aktion als lokale Aktion betrachten, die eigentlich nur im Inneren des Prozesses von Bedeutung ist. Wir nennen τ deshalb auch die *interne* Aktion.

Bemerkung 1. Sie können sich die interne Aktion in etwa als ein Statuslämpchen an Ihrem Laptop vorstellen, das leuchtet, wenn dieser etwas arbeitet, ohne dass Sie in Erfahrung bringen können, was er gerade konkret treibt.

Definition 1 (*Aktionen*)**.** Sei Com eine unendliche Menge von *Kommunikationsaktionen* und $\tau \notin Com$ die sogenannte *interne* Aktion. Dann ist

$$Act = Com \cup \{\tau\}$$

die Menge *aller* Aktionen.

Wir verwenden A, B, C, A_1, A', ... für Teilmengen von Com und nennen diese *Alphabete*. Eine weitere gängige Bezeichnung für Alphabete ist "Schnittstellen" (engl. *"interfaces"*). Wir verwenden griechische Buchstaben α, β, γ, α_1, α' ... um Kommunikationsaktionen aus Com zu bezeichnen. Lateinische Buchstaben a, b, c, a_1, a', ... stehen für Aktionen aus Act. Es ist sinnvoll, sich diese Unterscheidung gut zu verinnerlichen.

Bemerkung 2. In konkreten Anwendungen haben die Aktionen $\alpha \in Com$ oft eine gewisse Struktur. Zum Beispiel kann durch eine Festlegung

$$\alpha = (\underbrace{ch}_{\text{“channel”}}, \underbrace{ms}_{\text{“message”}})$$

verdeutlicht werden, dass die Aktion α das Schicken der Nachricht ms über den Kanal ch repräsentiert. Solch eine Struktur dient im Allgemeinen nur der Intuition und ist für die Theorie nicht relevant. Später werden wir Aktionen mit etwas mehr Struktur versehen.

Ein nebenläufiges System besteht aus mehreren Prozessen, die zeitgleich nebeneinander existieren und miteinander kommunizieren können. Die Prozesse selbst sind jedoch in der Regel *sequentiell* in ihrer Ausführung. Ein sequentieller Prozess ist ein Prozess, bei dem alle Bearbeitungsschritte strikt aufeinanderfolgen und nichts gleichzeitig geschieht. Wenn wir z.B. ein Mehrprozessorsystem betrachten, so können wir uns jeden einzelnen Prozessor als sequentiellen Prozess vorstellen, der Schritt für Schritt einen Befehl nach dem anderen abarbeitet. Alle Prozessoren zusammen ergeben ein nebenläufiges System. Moderne Prozessoren sind jedoch tatsächlich keine sequentiellen Prozesse: sie führen mehrere Anweisungen parallel aus, um die Ausführung des Programms zu optimieren. Man könnte also auch die Interna eines Prozesses selbst als nebenläufige Systeme modellieren.

Was in einem zu modellierenden System nebenläufig geschieht und was sequentiell, hängt oft davon ab, auf welcher Ebene oder wie detailliert wir es betrachten. Bei fast allen Systeme jedoch sind die grundlegenden Prozesse sequentiell.

1.2 Nebenläufigkeit und Parallelität

Warum verwenden wir den etwas ungewöhnlichen Begriff "*nebenläufig*" für etwas, das man vielleicht als "*parallel*" beschreiben möchte? Tatsächlich arbeitet der Graphikprozessor parallel zu seiner CPU, und die Aktivitäten des Handys werden parallel zu denen des Sendemasten ausgeführt. Trotzdem nennen wir die Aktivitäten jeweils "*nebenläufig*" zueinander. *Nebenläufigkeit* bezeichnet *logisch simultanes* Arbeiten, während "*Parallelität*" *tatsächlich simultanes* Arbeiten meint. Parallelität impliziert immer die Verfügbarkeit von mehreren Prozessoren oder von mehreren physikalisch unabhängigen Komponenten. So können zum Beispiel drei Prozesse auf drei separaten Prozessoren parallel ausgeführt werden, siehe Abbildung 1.2.

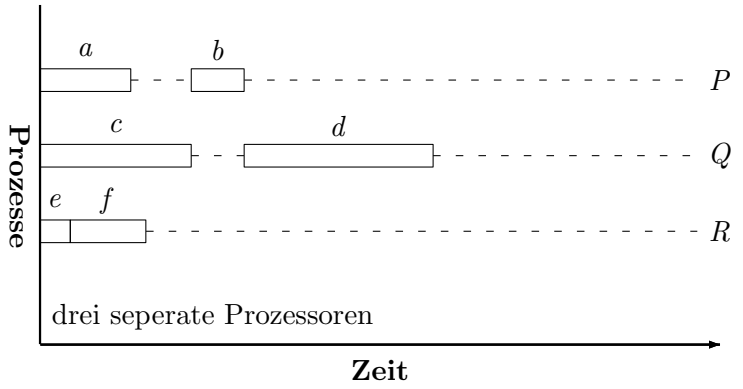

Abbildung 1.2: Ein nebenläufiges, paralleles System aus drei Prozessen, die physikalisch simultan auf drei Prozessoren ablaufen.

Nebenläufigkeit ist eine Verallgemeinerung von Parallelität, die diese Voraussetzung nicht macht. Ein konkretes nebenläufiges System kann mehr als einen Prozessor (oder ähnliche Ausführungseinheit) umfassen, muss aber nicht. Wenn es nur einen gibt, werden mehrere Prozesse darauf abwechselnd ausgeführt, siehe Abbildung 1.3. Der Fachbegriff dafür heißt *Interleaving*, im Deutschen in etwa wie *Verschränken* oder *Verschachteln*.

Insofern umfasst der Begriff *Nebenläufigkeit* auch *konzeptionelle* oder *potentielle* Parallelität, und nicht 'nur' *tatsächliche* Parallelität. Der Begriff der Nebenläufigkeit erlaubt uns, von der konkret verfügbaren Hardware komplett zu abstrahieren. Dies wird uns später erlauben, konzeptionelle Parallelität beim *nebenläufigen Programmieren* auszunutzen, und das Ergebnis, das nebenläufige Programm, bei Bedarf auch auf einem einzelnen Prozessor auszuführen. Das resultierende System ist kein paralleles System, aber sehr wohl ein nebenläufiges.

Aufgabe 1 (*für Archäologen*)**.** Dieter Schlau hat einen verstaubten Laptop mit Windows-XP auf dem Speicher gefunden. Er ist sehr erstaunt, als er feststellt, dass es sich um einen antiquierten Intel U1500 Monoprozessor handelt. Er fragt seine Großmutter, ob in alter Zeit auf solch einem Staubfänger wohl Thunderbird, Powerpoint, und Minesweeper gleichzeitig gelaufen sind bzw. haben. Seine Großmutter erinnert sich noch gut an damals. Mit welchen Schlagworten antwortet sie?

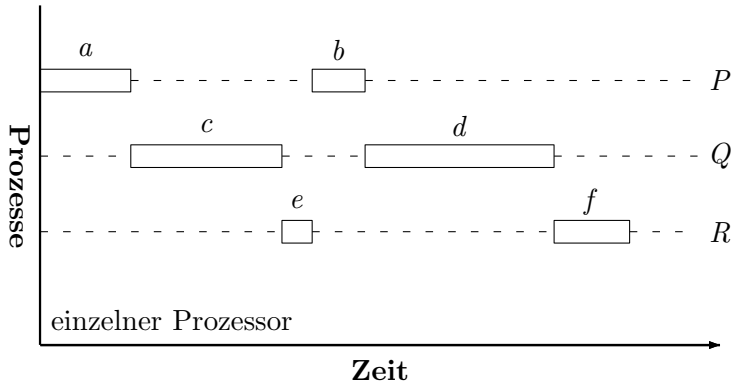

Abbildung 1.3: Ein nebenläufiges, nicht-paralleles System, bei dem drei Prozesse auf einem Prozessor *abwechselnd* ausgeführt werden.

1.3 Beschriftete Transitionssysteme

Die Beschreibung und Analyse der Nebenläufigkeit und Interaktionen eines Systems kann mit Hilfe sogenannter *beschrifteter Transitionssysteme* (engl. Labelled Transition Systems, LTS) erfolgen. Diese stellen zwar eine sehr abstrakte Sicht auf reale Systeme dar, allerdings bilden sie eine theoretisch fundierte Basis für die nebenläufige Programmierung und sind die Grundlage für die meisten weiteren Themen dieses Skriptes. Sie stehen daher im Mittelpunkt dieses Kapitels.

Um einen Prozess vollständig zu beschreiben, müssen wir zunächst die Menge der von ihm ausführbaren Aktionen festlegen. Das Ausführen einer Aktion verändert typischerweise den *Zustand* eines Prozesses.

Beispiel 3. Ein sehr einfaches Beispiel ist ein Streichholz (vgl. Abbildung 1.4). Die einzigen möglichen Aktionen sind

- *strike*, das signalisiert, dass das Streichholz entzündet (oder angestrichen) wird, sowie
- *extinguish*, welches das Erlöschen des Holzes andeutet.

Das Alphabet, also die Menge der Aktionen eines gewöhnlichen Streichholzes ist offensichtlich durch die Menge $\{strike, extinguish\}$ gegeben. Durch jede dieser Aktionen verändert der Streichholz seinen Zustand. Durch das Anzünden geht er vom Anfangszustand ($UNUSED$) in den brennenden Zustand ($BURNING$) über. Beim Erlöschen verändert sich der Zustand zum Endzustand ($EXTINCT$), von dem kein weiteres Verhalten erwartet wird.

Dieter Schlau fragt sich zwischenzeitlich, inwiefern ein Streichholz Teil eines nebenläufigen reaktiven Systems sein kann.

Wie wir sehen, ist zur Festlegung eines Prozesses, neben dem Alphabet, auch die Menge der Zustände, die der Prozess einnehmen kann, relevant, sowie die möglichen Übergänge zwischen

Abbildung 1.4: Ein Streichholz als LTS

seinen Zuständen. Jeder Übergang ist dabei mit der Aktion markiert, die den Zustandswechsel auslöst. Der Anfangszustand wird auch initialer Zustand genannt, und in der graphischen Darstellung durch einen eingehenden Pfeil markiert.

Beispiel 4. Im Folgenden finden Sie einige recht wahllose Beispiele, was durch Zustände und Übergänge ausgedrückt werden kann. Je nach konkretem Hintergrund sind zum Beispiel als Zustände denkbar:

- die momentane Farbe einer Ampel an einer Kreuzung;
- die momentanen Farben aller Ampeln an einer Kreuzung;
- die momentanen Werte aller Programmvariablen und des Programmzählers eines in Ausführung befindlichen Programms;
- der momentane Inhalt eines Datenpuffers einer Streaming-Anwendung;
- der Stand Ihres Studiums;
- der momentane Inhalt eines Einkaufswagens.

Mögliche Übergänge, die sich nicht unbedingt auf die obigen Zustände beziehen, sind beispielsweise:

- der Wechsel zu einer anderen Ampelfarbe;
- die Ausführung eines Befehls;
- das Versenden oder Empfangen einer Nachricht;
- das Landen eines Flugzeugs;
- eine Abbuchung von meinem Konto;
- das Überprüfen eines Passwortes.

Nach diesen einführenden Bemerkungen legen wir nun präzise das formale Modell eines Prozesses fest. Es besteht aus folgenden Bestandteilen: der Menge der möglichen Aktionen (Alphabet), die Menge der Zustände, die Menge der einzelnen Übergänge zwischen den Zuständen durch Aktionen, und außerdem dem initialen Zustand. Die Übergänge nennen wir *Transitionen.*

Definition 2 (*Transitionssystem*)**.** Ein *beschriftetes Transitionssystem* TS ist ein Quadrupel $(S, A, \longrightarrow, s_0)$, wobei

- S die Zustandsmenge,
- $A \subseteq Com$ das (Prozess-)Alphabet,
- $\longrightarrow \subseteq \quad S \times (A \cup \{\tau\}) \times S$ die Transitionsrelation,
- $s_0 \in S$ der initiale Zustand ist.

Dabei können S und A *endlich* oder *abzählbar unendlich* sein. Wir schreiben typischerweise $s \xrightarrow{a} s'$ anstelle von $(s, a, s') \in \longrightarrow$.

Beispiel 5. Für das Beispiel aus Abbildung 1.4, bekommen wir das LTS $(S, A, \longrightarrow, s_0)$, wobei $S = \{UNUSED, BURNING, EXTINCT\}$, $\longrightarrow = \{(UNUSED \xrightarrow{strike} BURNING), (BURNING \xrightarrow{extinguish} EXTINCT)\}$, und $s_0 = UNUSED$. A ist bereits bekannt.

Aufgabe 2 (*leicht, sofern Sie endliche Automaten kennen*)**.** Ein LTS ist sehr ähnlich einem *endlichen Automaten*. Was sind die konkreten Unterschiede?

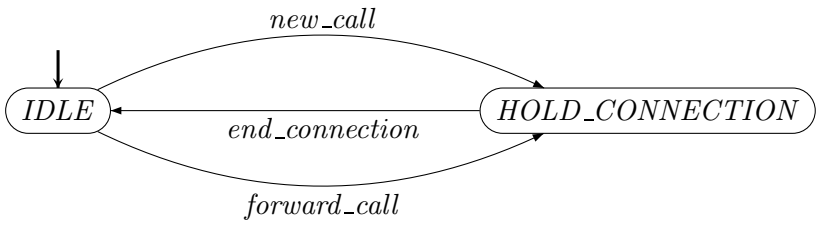

Abbildung 1.5: Einfaches Modell eines Sendemasts

Beispiel 6. Wir wollen nun das Verhalten des Beispieles 1 (Abbildung 1) durch LTS modellieren. Man kann dabei gut sehen, wie wir ein System in erster Näherung modellieren können, ohne uns mit technischen Details eines konkreten Systems auseinandersetzen zu müssen. In Abbildung 1.5 und 1.6 findet sich jeweils ein LTS für den Sendemast und eines für das Mobiltelefon, in graphischer Darstellung. Zunächst werden wir die einzelnen Prozesse jeweils für sich betrachten.

Der Sendemast kann – so wie er hier modelliert wird – höchstens ein Gespräch verwalten. Ein Gespräch kann entweder durch ein Handy begonnen werden (*new_call*) oder von Außen kommend weitergeleitet werden (*forward_call*). Eine offene Verbindung kann mit *end_connection* beendet werden.

Das Handy bietet in unserem Modell eine größere Anzahl an Möglichkeiten. Man kann ein Spiel spielen, man kann angerufen werden und man kann andere Leute anrufen. In jedem Zustand – außer *OFF* – ist es möglich dass das Handy sich abschaltet, zum Beispiel aufgrund einer internen Prüfung des Ladezustandes der Batterie. Diese Transitionen sind mit τ beschriftet, und führen zum Zustand *OFF*.

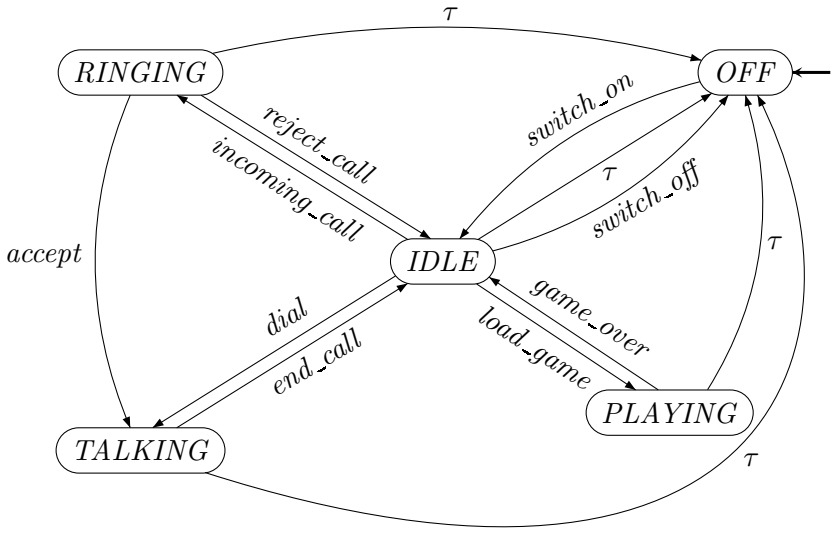

Abbildung 1.6: Einfaches Modell eines Handys

Aufgabe 3 (*leicht*)**.** Betrachten Sie die Modelle und versuchen Sie, das dargestellte Verhalten nachzuvollziehen. Schreiben Sie die dargestellten LTS in der Notation gemäß Definition 2.

Wir werden oft mit den Vorgängern und Nachfolgern von bestimmten Zuständen - gemäß der Transitionsrelation – eines LTS arbeiten. Da es jeweils mehrere davon geben kann, fassen wir diese in Mengen *Pre* und *Post* zusammen. Diese gibt es in einigen Variationen.

Definition 3 (*Vorgänger und Nachfolger*)**.** Sei ein LTS $TS = (S, A, \longrightarrow, s_0)$ gegeben. Sei $s \in S$, $C \subseteq S$ und $a \in Act$.

$$Post\,(s,a) = \left\{ s' \in S \mid s \xrightarrow{a} s' \right\}, \qquad Post(s) = \bigcup_{a \in Act} Post\,(s,a)$$

$$Pre\,(s,a) = \left\{ s' \in S \mid s' \xrightarrow{a} s \right\}, \qquad Pre(s) = \bigcup_{a \in Act} Pre\,(s,a)$$

$$Post(C,a) = \bigcup_{s \in C} Post(s,a), \qquad Post(C) = \bigcup_{s \in C} Post(s)$$

$$Pre(C,a) = \bigcup_{s \in C} Pre(s,a), \qquad Pre(C) = \bigcup_{s \in C} Pre(s)$$

Aufgabe 4 (*leicht*)**.** Definieren Sie $Post(C)$ unter Verwendung von $Post(C,a)$ statt $Post(s)$. Zeigen Sie, dass beide Definitionen äquivalent sind.

Außerdem verwenden wir zwei leicht verschiedene Mengen von Aktionen, die angeben, welche Aktionen in einem bestimmten Zustand als nächstes möglich (Act) sind, beziehungsweise

beobachtet (*Com*) werden könnten.

$$Act(s) = \left\{ a \in A \ \cup \ \{\tau\} \mid \exists s' : s \xrightarrow{a} s' \right\}, \qquad Com(s) = \left\{ \alpha \in A \ \mid \exists s' : s \xrightarrow{\alpha} s' \right\}$$

Aufgabe 5 (*kurz*)**.** Warum heißt das letzte Wort in obigen Satz "könnten"?

Wir nennen Zustand s einen *terminalen* Zustand gdw. $Post(s) = \emptyset$.

Bemerkung 3. Zustand s ist terminal gdw. $Act(s) = \emptyset$.

Aufgabe 6 (*leicht*)**.** Beweisen Sie Bemerkung 3.

Definition 4 (*Erreichbarkeit*)**.** Sei ein LTS $TS = (S, A, \longrightarrow, s_0)$ gegeben. Ein Zustand $s \in S$ heißt *erreichbar* von s' , falls es Zustände $s_1, s_2, \ldots, s_n \in S$ und Aktionen $a_1, \ldots, a_n \in Act$ gibt, so dass

$$s' \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n \text{ und } s_n = s$$

$Reach(s)$ ist die Menge aller von s *erreichbaren* Zustände in TS.

Ein Zustand s ist in $TS = (S, A, \longrightarrow, s_0)$ erreichbar, wenn $s \in Reach(s_0)$. Wir definieren $Reach(TS) = Reach(s_0)$.

Aufgabe 7 (*mittel*)**.** Sei ein LTS $TS = (S, A, \longrightarrow, s_0)$ gegeben. Zeigen Sie, dass

$$Reach(s) = \bigcup_{n \in \mathbb{N}} Post^n(s)$$

wobei

$$Post^0(s) = \{s\} \qquad \text{und} \qquad Post^{n+1}(s) = Post(Post^n(s))$$

Können Sie daraus einen Algorithmus zur Berechnung von $Reach(TS)$ entwerfen, sofern S endlich ist?

1.4 Pfade und Spuren, endlich und unendlich

Ein LTS beschreibt das gesamte mögliche Verhalten eines Prozesses. Wenn wir eine konkrete Ausführung des Prozesses beschreiben wollen, so können wir das tun, indem wir festhalten, welche Zustände ein Prozess bei seiner Ausführung durchlaufen hat und welche Aktionen er dabei jeweils ausgelöst hat. Wir beschreiben dies mit einem *Pfad* durch das LTS.

Definition 5 (*Pfad*)**.** Ein *Pfad* ϱ von TS ist eine *alternierende* Folge von Zuständen und Aktionen, die im initialen Zustand beginnt:

$$\varrho = s_0, a_1, s_1, a_2, \ldots, a_n, s_n, \text{ so dass } s_i \xrightarrow{a_{i+1}} s_{i+1} \quad \forall\, 0 \leq i < n$$

Während ein Pfad eines LTS auch die Identitäten der Zustände eines Prozesses offenlegt, die bei der Ausführung durchlaufen werden, beschränkt sich eine *Spur* ausschließlich auf den von außen beobachtbaren Anteil eines Pfades: die Folge seiner Aktionen.

Definition 6 (*Spur*)**.** Die *Spur* $trace(\varrho)$ des Pfades ϱ ist die Folge seiner Aktionen:

$$trace(s_0, a_1, s_1, a_2, \ldots, a_n, s_n) = a_1\, a_2 \cdots a_n$$

Für $TS = (S, A, \longrightarrow, s_0)$ sei $Paths(TS)$ die Menge aller Pfade in TS. Weiterhin sei $Traces(TS) = \{trace(\varrho) \mid \varrho \in Paths(TS)\}$ die Menge der (gesamten) Spuren aller Pfade von TS.

Beispiel 7. Betrachten wir noch einmal das Beispiel mit dem Handy und dem Sendemast (vgl. Abbildung 1.5 und 1.6). Die Spuren des Modells des Sendemasts bestehen abwechselnd aus zum einen einer der beiden Aktionen *new_call* und *forward_call* sowie zum anderen der Aktion *end_connection*. Die Spuren des Handys lassen sich nur umständlich aufzählen. Eine Beispielspur ist:

$$switch\_on \;\; incoming\_call \;\; reject\_call \;\; dial \;\; \tau \;\; switch\_on$$

Aufgabe 8 (*nebenbei*)**.** Was entspricht der Menge $Traces(TS)$ in der Automatentheorie?

Rein sequentielle Systeme, die aus bestimmten Eingaben eine Ausgabe berechnen, wie z.B. SML-Programme, sind stets so spezifiziert, dass ihre Ausführung nach einiger Zeit terminiert, wenn alles korrekt verläuft. Nebenläufige Systeme hingegen sind häufig so konzipiert, dass sie auf unbestimmte Zeit laufen und es kein beabsichtigtes Ende ihrer Ausführung gibt. Ein Mobiltelefon z.B. soll keineswegs nach fünf Anrufen nicht mehr auf Eingaben reagieren, sondern soll – wenn es zu keinen Problemen wie leergelaufenen Batterien kommt – fortwährend auf Eingaben reagieren. Da solche Systeme also auf unbestimmte Zeit laufen sollen, ist es sinnvoll, ihre Pfade und damit deren Spuren als unendlich aufzufassen. Wir definieren daher:

Definition 7 (*unendlicher Pfad*)**.** Ein *unendlicher Pfad* ϱ von TS ist eine unendliche *alternierende* Folge von Zuständen und Aktionen, die im initialen Zustand beginnt:

$$\varrho = s_0, a_1, s_1, a_2, \ldots \quad so\ dass\ s_i \xrightarrow{a_{i+1}} s_{i+1} \quad \forall\, i \in \mathbb{N}$$

Definition 8 (*unendliche Spur*)**.** Die *unendliche Spur* $trace(\varrho)$ ist die Spur eines unendlichen Pfades ϱ:

$$trace(s_0, a_1, s_1, a_2, \ldots) = a_1\, a_2 \cdots$$

Für $TS = (S, A, \longrightarrow, s_0)$ sei $Paths^{\omega}(TS)$ die Menge aller unendlichen Pfade in TS. Die Menge aller unendlicher Spuren eines LTS bezeichnen wir mit

$$Traces^{\omega}(TS) = \{trace(\varrho) \mid \varrho \in Paths^{\omega}(TS)\}$$

Bemerkung 4. Um die Menge aller endliche Folgen von Objekten einer Menge A darzustellen, schreibt man allgemein A^*. Die Menge aller unendlichen Folgen von Objekten aus A bezeichnet man allgemein mit A^{ω}. Es gilt daher $Traces(TS) \subseteq Act^*$ und $Traces^{\omega}(TS) \subseteq Act^{\omega}$. Man beachte, dass $Traces(TS)$ und $Traces^{\omega}(TS)$ disjunkt sind.

Wir werden im Folgenden sowohl endliche als auch unendlichen Spuren von Transitionssystemen betrachten.

Aufgabe 9 (*empfohlen*)**.** Geben sie ein LTS an, dessen einzige unendliche Spur die Dezimaldarstellung des Bruchs 4/7 ist. Wieviel Zustände hat das LTS mindestens? Warum mindestens? Können Sie diese Aufgabe auch für die Zahl π lösen?

1.5 Isomorphie

Ein LTS dient uns dazu, das beobachtbare Verhalten eines Prozesses formal zu fassen. Diese verhaltensorientierte Sicht motiviert, dass wir uns im Allgemeinen nicht für Zustände interessieren, die nicht erreichbar sind. Außerdem sind für uns die Identitäten der Zustände nicht relevant, ganz im Unterschied zu den Aktionen.

Wir werden deswegen LTS normalerweise nur bis auf Isomorphie unterscheiden.

Definition 9 (*Isomorphie*)**.** Zwei Transitionssysteme $TS = (S, A, \longrightarrow, s_0)$ und $TS' = (S', A', \longrightarrow', s_0')$ sind *isomorph* ($TS \sim_{\text{iso}} TS'$), wenn $A = A'$, und es eine Bijektion β gibt mit

$$\beta : \mathit{Reach}(TS) \to \mathit{Reach}(TS'), \text{ so dass } \beta(s_0) = s_0'$$

und für alle $s_1, s_2 \in \mathit{Reach}(TS)$ und alle $a \in A \cup \{\tau\}$ gilt:

$$s_1 \xrightarrow{a} s_2 \text{ gdw. } \beta(s_1) \xrightarrow{a}{}' \beta(s_2)$$

Es erscheint sinnvoll zwei LTS als gleichwertig zu betrachten, wenn sie isomorph sind. Zu beachten ist, dass sich Isomorphie nur auf *erreichbare Zustände* beschränkt.

Aufgabe 10 (*etwas lästig*)**.** Geben Sie für jedes LTS, welches in diesem Kapitel abgebildet ist, isomorphe LTS mit der Zustandsmenge $S = \{Z, Y, X, W, V, U, T, S, R, Q, P\}$ an.

Wir bezeichnen die Menge aller LTS, die sich mit einem Alphabet A bilden lassen, als Menge der *Prozesse über A*.

Satz 1. Isomorphie ist eine Äquivalenzrelation auf der Menge der Prozesse über Act.

Beweis. Zu zeigen ist, dass $\sim_{\text{iso}}$ auf der Menge der Prozesse über Act reflexiv, transitiv und symmetrisch ist. Wir skizzieren hier nur die Transitivität.

Transitivität Seien $TS = (S, Act, \longrightarrow, s_0)$, $TS' = (S', Act, \longrightarrow', s_0')$ und $TS'' = (S'', Act, \longrightarrow'', s_0'')$ gegeben und gelte $TS \sim_{\text{iso}} TS'$ sowie $TS' \sim_{\text{iso}} TS''$. Es gibt also Bijektionen

$$\beta : \mathit{Reach}(TS) \to \mathit{Reach}(TS') \text{ und } \beta' : \mathit{Reach}(TS') \to \mathit{Reach}(TS'')$$

mit den geforderten Eigenschaften aus Definition 9. Es ist leicht zu zeigen, dass

$$\beta'' : \mathit{Reach}(TS) \to \mathit{Reach}(TS'')$$

die geforderten Eigenschaften für TS und TS'' hat, wobei $\beta''(s) = \beta'(\beta(s))$ für $s \in \mathit{Reach}(TS)$. Damit gilt $TS \sim_{\text{iso}} TS''$.

□

Aufgabe 11 (*empfohlen*)**.** Füllen Sie die Details des Beweises zu Satz 1 aus.

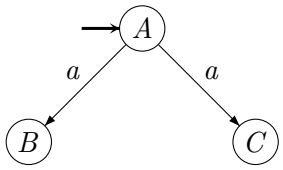

Abbildung 1.7: Interner Nichtdeterminismus

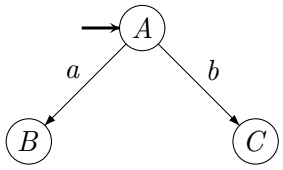

Abbildung 1.8: Externer Nichtdeterminismus

1.6 Nichtdeterminismus

In Transitionssystemen finden wir sehr oft ein Phänomen, das als Nichtdeterminismus bezeichnet wird. Betrachten wir dazu Abbildung 1.7. Der angegebene Prozess kann von Zustand A entweder in Zustand B oder C wechseln. Er kann beides mit der selben Aktion a tun. Wenn also die Aktion a ausgeführt, beziehungsweise von außen beobachtet wird, ist nicht festgelegt, in welchen der beiden Nachfolgezustände der Prozess mit der Aktion a wechselt.

Wenn es in einem System eine Auswahl zwischen zwei Alternativen gibt, und durch nichts festgelegt ist, welche der beiden Alternativen gewählt werden muss, dann nennen wir diese Situation nichtdeterministisch. In dem konkreten Beispiel sprechen wir genauer von *internem Nichtdeterminismus*. Die Entscheidung, welche der beiden Transitionen genommen wird, ist *intern*, sie ist nicht aus der Beobachtung der Aktion a ableitbar.

In der Situation in Abbildung 1.8 liegt kein interner Nichtdeterminismus vor. Dort kann der Prozess zwar ebenfalls vom Zustand A in entweder den Zustand B und C wechseln, jedoch ist diese Entscheidung mittels der zwei unterschiedlichen Aktionen a und b beobachtbar. Hier liegt kein interner Nichtdeterminismus vor. Wir sprechen hier dagegen von *externem Nichtdeterminismus*. Wenn der Prozess sich im Zustand A befindet, und eine Aktion ausführt, ist zunächst einmal nicht determiniert, welche der beiden ausgewählt wird. Allerdings ist die Entscheidung direkt durch das beobachtete Verhalten sichtbar.

Definition 10 (*Nichtdeterminismus*)**.**
Sei ein LTS $TS = (S, A, \longrightarrow, s_0)$ gegeben. Ein Zustand $s \in S$ heißt

- *extern nichtdeterministisch* gdw. $|Act(s)| > 1$, und

- *intern nichtdeterministisch* gdw. es eine Aktion $a \in A$ gibt, so dass $|Post(s, a)| > 1$;

TS ist *deterministisch* gdw. für alle $s \in S$,

$$|Post(s)| \leq 1 \qquad \text{und} \qquad |Act(s)| \leq 1$$

Andernfalls heißt TS *nichtdeterministisch*.

Bemerkung 5. Sofern TS keine intern nichtdeterministischen Zustände umfasst, lässt sich $\longrightarrow$ auch als *partielle* Funktion $(S \times A) \rightharpoonup S$ darstellen. Für deterministische TS kann man eine partielle Funktion $S \rightharpoonup (A \times S)$ verwenden, und für diese gilt $|Traces^{\omega}(TS)| \leq 1$.

Es gibt verschiedene Gründe, warum Nichtdeterminismus bei der Modellierung von Systemen wichtig ist. Externer Nichtdeterminismus ist zum Beispiel notwendig um verschiedene Reaktionen auf die Umgebung vorsehen zu können. Der Sendemast aus Beispiel 7 (Abbildung 1.5) ist im Zustand *IDLE* extern nichtdeterministisch, um auf zwei verschiedene externe Ereignisse (*new_call* und *forward_call*) adäquat reagieren zu können.

Interner Nichtdeterminismus ist nützlich, wenn unser Modell von konkreten Systemdetails abstrahieren soll, die zum Beispiel unwichtig, oder nicht bekannt sind. Ein weiterer Kontext, in dem interner Nichtdeterminismus häufig verwendet wird, ist, dass verschiedene Implementierungsalternativen zu berücksichtigen sind, und noch nicht klar ist, welche Alternative tatsächlich verwendet wird. Außerdem werden wir später sehen, dass sowohl externer als auch interner Nichtdeterminismus durch die nebenläufige Ausführung von Prozessen unvermeidlich entsteht, selbst wenn alle sequentiellen Prozesse deterministisch sind.

Aufgabe 12 (*empfohlen*)**.** Dieter Schlau ist dieses Semester erstmals Übungsgruppenleiter. Als er die Zuteilung der Teilnehmenden zu den Übungsgruppen sieht, ist er schwer enttäuscht: Alle Teilnehmerinnen tummeln sich in einer Gruppe. Leider nicht in seiner.

Er geht sofort zum Assistenten der Vorlesung und will wissen, mit welchem Algorithmus diese Verteilung zustandegekommen sei. Der Assistent antwortet ihm, dass der Algorithmus die Teilnehmenden jeder Übungsgruppe *nichtdeterministisch* auswählt.

Da Dieter sich benachteiligt sieht, fordert er, dass der Algorithmus nochmals ausgeführt wird. Doch egal wie oft er den Algorithmus ausführt, es kommt immer genau die selbe Übungsgruppeneinteilung zustande wie anfangs. Daraufhin meint Dieter, dass dies bei einem *nichtdeterministischen* Algorithmus nicht passieren kann, und der Algorithmus daher nicht *nichtdeterministisch* sein kann.

Hat Dieter Recht? Überlegen Sie, inwiefern man eine Spezifikation, die *Nichtdeterminismus* enthält, konkret (z.B. als Java- oder SML-Programm) implementieren kann.