Rheinisch-Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik II
Prof. Dr. Klaus Indermark

Synthese Reaktiver Systeme durch Live Sequence Charts

Seminar: Message Sequence Charts
WS 2004/05

Frank Radmacher
Matrikelnummer: 235573

Betreuung: Benedikt Bollig
Lehrstuhl für Informatik II, RWTH Aachen

Kurzdarstellung

Live Sequence Charts sind eine mächtige Erweiterung von Message Sequence Charts. Wir verfolgen das Ziel, ausgehend von einer Spezifikation bestehend aus Live Sequence Charts, reaktive Systeme zu synthetisieren. Dazu werden wir eine für unser Ziel zweckmäßige Definition von Live Sequence Charts vorstellen und schließlich unsere Problemstellung auf eine spieltheoretische Ebene zurückführen, so daß automatische Codegenerierung möglich ist.

Inhaltsverzeichnis

1 Einleitung **3**

2 Live Sequence Charts **4**
2.1 Beschreibung 4
2.2 Syntax und Semantik 6
2.3 Erfüllbarkeit von LSCs 7
2.4 Cuts 10

3 Liveness und Safety **12**

4 Synthese **15**
4.1 Problemstellung 15
4.2 Das Konsistenzproblem 16
4.3 Algorithmus 18
4.4 Komplexität 21

5 Zusammenfassung und Ausblick **22**

1 Einleitung

Die Konstruktion reaktiver Systeme ist ein großes Problem der Informatik. Besonders die zunehmende Verbreitung verteilter Systeme, deren Entwicklung als sehr fehleranfällig gilt, fordert eine zuverlässige Methodik. In [1, 2] werden vor allem drei Punkte bemängelt, die durch den von uns vorgestellten Ansatz vermieden werden könnten.

- In der Industrie werden zum Entwurf verteilter Systeme derzeitig vor allem Methoden angewendet, die für den Entwurf herkömmlicher sequenzieller Systeme gedacht waren. Wir wollen dagegen ansatzweise eine Sprache, basierend auf *Message Sequence Charts* (MSCs), verwenden und diese so erweitern, daß bequem die gegenseitigen Interaktionen und Reaktionen der einzelnen Prozesse bzw. Instanzen zueinander modelliert werden können.

- Je nach verwendeter Spezifikationssprache ist es schwierig, eine Spezifikation so anzugeben, daß sie den tatsächlichen Sachverhalt wiedergibt. So kann leicht ein unerwünschtes Verhalten in die Spezifikation modelliert werden. Auch dieses Problem läßt sich mindern, indem man eine intuitive Sprache verwendet, wie die graphisch darstellbaren MSCs. Statt eine abstrakte Spezifikationssprache zu verwenden, in der es sehr viel Aufwand bedarf, bis man ein System exakt modelliert hat, halten wir es für sinnvoller, eine exakte szenario-basierte Sprache zu verwenden. Dadurch schließen wir auch die große Lücke, die entsteht, wenn man direkt von Beispielszenarien zu mächtigen Spezifikationssprachen wie Automaten oder temporale Logiken wechselt.

- Heutzutage wird meistens eine nachträgliche *a posteriori* Verifikation durchgeführt, bei der es sehr viel Arbeit erfordert, festzustellen, daß eine Implementierung nicht der vorgegebenen Spezifikation entspricht. Unser Ansatz soll dagegen bereits Korrektheit durch Konstruktion sicherstellen.

In der Softwaretechnik sind zwar *Message Sequence Charts* und use-case Notationen durchaus schon etabliert, jedoch beschreiben diese bloß *Beispiele* von Interaktion und Reaktion des Akteurs und des zu entwickelnden Systems. MSCs eignen sich daher nicht zur Spezifikation solcher Systeme, da sie nicht zwischen möglichem und vorgeschriebenen Verhalten unterscheiden. Genau genommen schweigen sich MSCs über zwei Fragen aus:

1. Weglassen von Nachrichten. Dürfen Nachrichten, die im Chart weggelassen werden, während dessen Ausführung spontan auftreten; oder wird das Auftreten dieser Nachrichten durch das bloße Weglassen verboten?

2. Chart status. Ist das durch einen Chart beschriebene Szenario *obligatorisch* bzw. *universell*, d. h. soll sich das System *immer* wie beschrieben verhalten; oder ist das durch einen Chart beschriebene Szenario als *provisorisch* bzw. *existentiell* zu bezeichnen, d. h. soll der Chart nur aussagen, daß ein bestimmtes Verhalten des Systems *möglich* sein soll?

Um diese Probleme zu vermeiden, führten im Jahr 1998 Damm und Harel in [3] die *Live Sequence Charts* (LSCs) ein, welche eine Erweiterung der MSCs darstellen. Zur Synthese wurden LSCs erstmals in [4] verwendet. Wir werden jedoch in Abschnitt 2 eine einfachere Definition von LSCs geben, die im Wesentlichen die beiden zuvor erwähnten Nachteile von MSCs ausmerzt.

Neben der Syntax und Semantik werden wir in Abschnitt 2 auch die sogenannten *Cuts* einführen, welche uns eine formale Zustandsbeschreibung von LSCs ermöglichen. In Abschnitt 3 werden wir zeigen, daß sich die Erfüllbarkeit von LSCs auf zwei Eigenschaften zurückführen läßt, nämlich auf eine Lebendigkeitsbedingung (*Liveness*) und eine Sicherheitsbedingung (*Safety*). Dadurch wird es uns in Abschnitt 4 gelingen, die Synthese eines reaktiven Systems als Spiel aufzufassen. Das dadurch gegebene *Konsistenzproblem* läßt sich somit durch spieltheoretische Algorithmen lösen. Im letzten Abschnitt werden wir eine Zusammenfassung und einen kurzen Ausblick auf bestehende Probleme geben.

2 Live Sequence Charts

2.1 Beschreibung

Die von uns betrachteten LSCs unterscheiden sich von der ursprünglichen Definition von LSCs in [3]. Wir führen ein vereinfachtes übersichtliches Modell ein, dessen Aussagekraft ausreicht, um die für uns wesentlichen Resultate zu zeigen. Anders als in [3] schenken wir Bedingungen und Modalitäten von Nachrichten keine Beachtung. Der Einfachheit wegen nehmen wir alle Nachrichten als unmittelbar an. Verzögerte Nachrichten können in unserem Modell repräsentiert werden, indem Kommunikationskanäle jeweils als eigene Prozesse bzw. Instanzen modelliert werden.

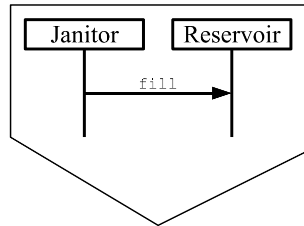
Ein LSC besteht aus mehreren *Instanzen*, die oft auch als *Prozesse* oder *Agenten* bezeichnet werden. Diese werden durch vertikale Linien repräsentiert. Das LSC in Abb. 1(b) besteht z. B. aus den Instanzen `Customer` und `Machine`. Zwischen den Instanzen werden *Nachrichten* (engl. *messages*) wie `askCocoa` ausgetauscht. Oft sprechen wir von diesen Nachrichten auch von *Ereignissen*. Nachrichten werden durch Pfeile repräsentiert. Die Anfangs- und Endpunkte dieser Pfeile bezeichnen wir als *Positionen* (engl. *locations*).

In jeder Instanz sind Positionen zeitlich von oben nach unten geordnet. Diese zeitliche Ordnung von Positionen kann aufgehoben werden, indem diese zu einer *Koregion* zusammengefaßt werden. Dieses wird durch eine vertikale gestrichelte Linie neben den Positionen gekennzeichnet. So können z. B. in Abb. 2(a) die Positionen `askCocoa` und `insertCoin` in beliebiger Reihenfolge stattfinden.

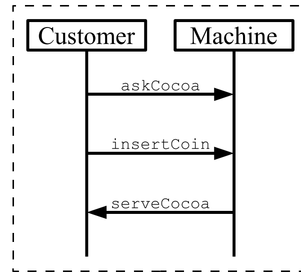
Jedes LSC hat einen der folgenden drei Modi:

- **Initial:** Diese LSCs werden von einer fünfeckigen Box umrandet, wie in Abb. 1(a). Ein Initial LSC muß am Anfang jeder Ausführung des zu entwickelnden Systems erfüllt werden.

- **Existential:** Diese LSCs werden von einem rechteckigen gestrichelten Kasten umrandet, wie in Abb. 1(b). Ein Existential LSC entspricht einem *provisorischen* Sze-

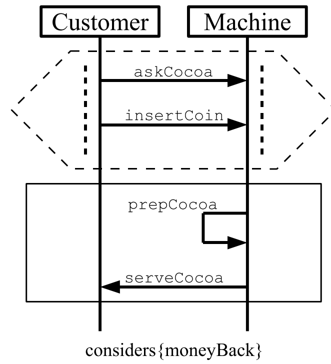


(a)

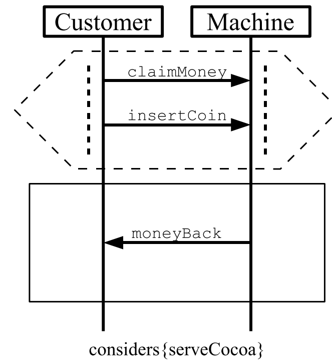


(b)

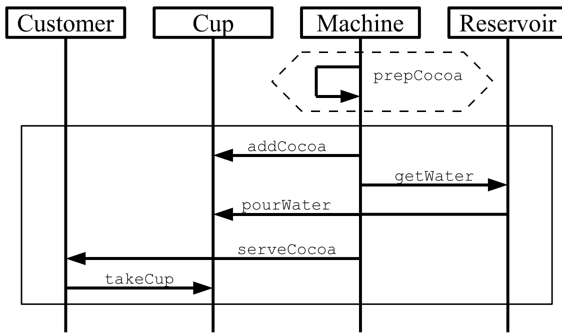
Abbildung 1: Initial und Existential LSCs



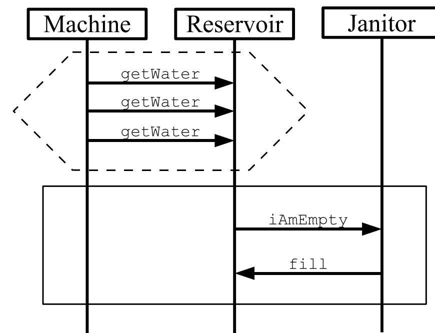
(a)



(b)



(c)



(d)

Abbildung 2: Universal LSCs

nario. Es muß während der Ausführung des LSCs irgendwann mindestens einmal erfüllt werden.

- **Universal:** Diese LSCs bestehen aus zwei Teilen. Der obere Teil, der sogenannte *Prechart*, wird durch eine sechseckige Box begrenzt. Der untere Teil, der sogenannte *Main Chart*, schließt direkt an den Prechart an und wird von einem durchgezogenen Rechteck umrandet. Beispiele für Universal Charts zeigt Abb. 2. Ein Universal Chart entspricht einem *obligatorischen* Szenario. Die semantische Bedeutung ist folgende: Immer wenn das durch den *Prechart* beschriebene Verhalten erfüllt wird, muß anschließend irgendwann das Verhalten des *Main Charts* erfüllt werden. Universal Charts erlauben somit die Beschreibung *reaktiver Systeme*.

In LSCs können Ereignisse als *beschränkt* definiert werden. Diese Ereignisse dürfen nur dann stattfinden, wenn diese explizit im LSC stehen. Nicht beschränkte Ereignisse dürfen jederzeit stattfinden. Beschränkte Ereignisse können in einer Klausel deklariert werden, welche in der Regel mit *restricted* oder *considers* gekennzeichnet wird. So dürfen z. B. in Abb. 2(a) die Ereignisse `pourWater` und `fill` aus Szenarien (c) und (d) zwischen `prepCocoa` und `serveCocoa` auftreten. Andererseits darf das Ereignis `moneyBack` nicht während der Ausführung des LSCs in Abb. 2(a) erfolgen. Standardmäßig sind alle Ereignisse, die in einem LSC vorkommen, beschränkt. So darf in Abb. 2(c) nach dem Ereignis `addCocoa` nicht spontan das Ereignis `serveCocoa` stattfinden, ohne daß zuvor `getWater` und `pourWater` erfolgten.

2.2 Syntax und Semantik

Gegeben sei eine endliche Menge Ag von *Instanzen* und eine endliche Menge $\mathcal{M}$ von *Nachrichtenbezeichnern*. Dann beschreibt das endliche Alphabet $\Sigma = Ag \times \mathcal{M} \times Ag$ die Menge aller möglichen *Ereignisse*. Dabei bedeutet $(a_1, m, a_2) \in \Sigma$, daß a_1 eine Nachricht m an a_2 sendet.

Eine *Spezifikation* ist eine Menge von LSCs, welche wiederum aus sogenannten *Basic Charts* bestehen. Einen Basic Chart definieren wir ähnlich wie einen gewöhnlichen MSC.

Definition 2.1. (Basic Chart)
Ein *Basic Chart* ist ein Tupel

$$C = \langle \Sigma(C), A, (L_a, <_a)_{a \in A}, \phi_C, msg \rangle$$

mit folgenden Bestandteilen:

- $\Sigma(C) \subseteq \Sigma$ sind die durch C *beschränkten* Ereignisse.
- $A \subseteq Ag$ ist die Menge der an C beteiligten *Instanzen* (engl. *Agents*).
- L_a ist die Menge der in a auftretenden Positionen. Die Menge aller in einem Basic Chart C auftretenden Positionen werden wir mit $L_C = \bigcup_{a \in A} L_a$ abkürzen.

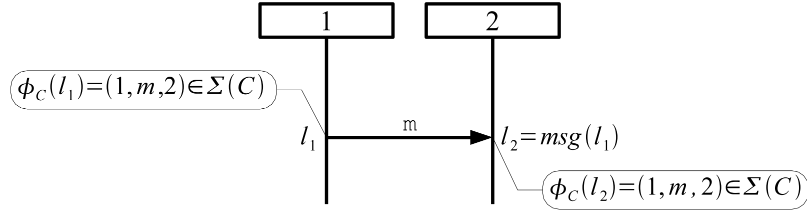


Abbildung 3: Semantische Bezeichnungen in einem Basic Chart

- $<_a \subseteq L_a \times L_a$ ist eine strikte Halbordnung, d. h. sie ist transitiv, irreflexiv und somit auch antisymmetrisch. Sie beschreibt die Reihenfolge, in der die Positionen auftreten können. Wegen möglicher Koregionen ist diese nicht zwingend eine totale Ordnung. Zwei Positionen, die zur selben Koregion gehören, werden verschieden beschriftet.

- $\phi_C : L_C \to \Sigma(C)$ ist eine *Beschriftungsfunktion*, welche die Positionen mit beschränkten Ereignissen beschriftet.

- msg ist eine *Kommunikationsfunktion*, die in der grafischen Darstellung den Pfeilen entspricht. Sie ist eine bijektive Funktion, so daß

  1. der Definitionsbereich $D(msg) \subseteq L_C$ und der Wertebereich $W(msg) \subseteq L_C$ die Menge der Positionen L_C partitioniert,
  2. $\forall l \in D(msg) : \ \phi_C(l) = \phi_C(msg(l))$,
  3. $\forall l \in D(msg) : \ l \in L_{\pi_1(\phi_C(l))}$ und $msg(l) \in L_{\pi_3(\phi_C(l))}$, wobei die Funktion $\pi_j(t)$ die j-te Komponente des Tupels t zurückliefert.

Obwohl die Kommunikation zwischen den Instanzen bereits durch $\Sigma(C)$ und ϕ_C vollständig beschrieben ist, ist die Kommunikationsfunktion msg keinesfalls überflüssig. Sie stellt durch ihre Eigenschaften sicher, daß die Ereignisse in $\Sigma(C)$ in sich konsistent definiert sind. Eine anschauliche Darstellung der Bestandteile von Definition 2.1 liefert Abb. 3.

2.3 Erfüllbarkeit von LSCs

Da wir in einem Basic Chart C nur eine Ordnungsrelation für die einzelnen Instanzen definiert haben, wollen wir im folgenden Ordnungsrelationen auf allen Positionen eines Basic Charts C definieren. Eine Quasiordnung (transitiv und reflexiv, aber nicht zwingend antisymmetrisch) $\preceq$ auf C erhalten wir, indem wir die Halbordnungen der Instanzen dahingehend erweitern, daß mit Pfeilen verbundene Positionen als zeitgleich angenommen werden. Formal ist $\preceq$ auf einem Basic Chart C die kleinste Quasiordnung, für die gilt:

1. $l <_a l'$ für ein $a \in Ag \implies l \preceq l'$
2. $l \in D(msg) \implies l \preceq msg(l)$ und $msg(l) \preceq l$

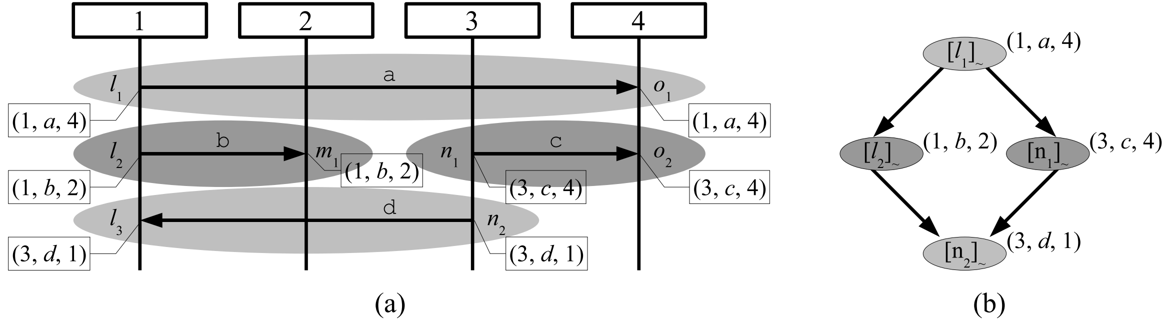


Abbildung 4: (a) Basic Chart C und Äquivalenzklassen $[L_C]_\sim$. (b) zugehörige lpo $\mathcal{L}_C$.

Durch diese Quasiordnung sind Äquivalenzklassen definiert, so daß

$$l \sim l' \iff l \preceq l' \text{ und } l' \preceq l.$$

Ein Basic Chart, in dem alle Positionen, die zur selben Äquivalenzklasse gehören, auch mit den gleichen Ereignissen beschriftet sind, bezeichnen wir als *wohlgeformt.* In der grafischen Darstellung entspricht dieses der Bedingung, daß Pfeile sich nicht kreuzen dürfen. Da wir Nachrichten als unmittelbar annehmen, werden wir im folgenden alle Charts als wohlgeformt annehmen.

In wohlgeformten Charts werden also genau die mit Pfeilen verbundenen Positionen zu einer Äquivalenzklasse zusammengefaßt, so daß diese dann eine strikte Halbordnung $\langle [L_C]_\sim, \prec_C \rangle$ bilden gemäß

$$[l]_\sim \prec_C [l']_\sim \iff l \preceq l' \text{ und } l' \not\preceq l.$$

Dieses entspricht dem Ordnen der *eintretenden* Ereignisse in einem Chart von oben nach unten. Ein wohlgeformter Chart C definiert somit eine *beschriftete strikte Halbordnung* (engl. *labeled partial order* (lpo)). Die Äquivalenzklassen $[L_C]_\sim$ werden durch die Funktion ϕ_C gemäß $\phi_C([l]_\sim) = \phi_C(l)$ mit Ereignissen aus $\phi_C([L_C]_\sim) \subseteq \Sigma(C)$ beschriftet. Diese lpo bezeichnen wir dann mit $\mathcal{L}_C = \langle [L_C]_\sim, \prec_C, \phi_C \rangle$. Für die Menge aller Äquivalenzklassen $[L_C]_\sim$ einer solchen lpo werden wir oft einfach nur $\mathcal{L}_C$ schreiben. Dieses entspricht der Menge aller verschiedener erreichbaren Positionen in einem wohlgeformten Chart. Abb. 4 verdeutlicht, wie man von einem Basic Chart C ausgehend die Äquivalenzklassen $[L_C]_\sim$ und die zugehörige lpo $\mathcal{L}_C$ erhält.

Die strengen Halbordungen auf einem Chart erlauben uns, die Konkatenation $C_1 \cdot C_2$ zweier Basic Charts C_1 und C_2 zu definieren. Diese Operation ist definiert, falls die Mengen ihrer Positionen L_{C_1} und L_{C_2} disjunkt sind. Es ergibt sich die komponentenweise Vereinigung beider Halbordnungen, so daß jede Position bzw. jede zugehörige Äquivalenzklasse aus C_1 kleiner als jede Position bzw. jede zugehörige Äquivalenzklasse aus C_2 ist.

Definition 2.2. (Linearisierung)
Eine *Linearisierung* einer Halbordnung ist eine *lineare Ordnung* (auch *totale Ordnung* genannt), die diese Halbordnung erweitert.

Eine Linearisierung ist also eine ausgewählte Folge von Ereignissen in einem Chart. Beachte, daß eine Linearisierung keine Ereignisse aus $\Sigma(C)$ enthalten kann, die in einer *considers* Klausel aufgeführt sind, da diese Ereignisse keine Äquivalenzklassen aus $\mathcal{L}_C$ beschriften. Nun können wir definieren, wann eine Folge von Ereignissen einen Basic Chart *erfüllt*:

Definition 2.3. ($r \models_B C$)
Eine Folge von Ereignissen $r = e_1 \ldots e_n \in \Sigma^*$ *erfüllt* einen Basic Chart C ($r \models_B C$) genau dann, wenn $r|_{\Sigma(C)}$ eine Linearisierung der lpo $\langle [L_C]_\sim, \prec_C, \phi_C \rangle$ ist. Der Projektionsoperator $|_{\Sigma(C)}$ entfernt aus einer Folge alle Ereignisse, die nicht in $\Sigma(C)$ sind; formal: $\epsilon|_\Sigma = \epsilon$ und $we|_\Sigma = (w|_\Sigma)e$, falls $e \in \Sigma$ bzw. $we|_\Sigma = w|_\Sigma$, falls $e \notin \Sigma$.

Mit Basic Charts $C_{(i)}$ lassen sich LSCs konstruieren. Man muß nur noch zusätzlich angeben, um welchen Typ von LSC es sich handelt. Ein *Initial Chart* hat die Form $\triangleright C$, ein *Existential Chart* die Form $\Diamond C$, und für einen *Universal Chart* schreiben wir $\Box(C_1, C_2)$, wobei C_1 der Prechart und C_2 der Main Chart ist. Für Universal Charts fordern wir, daß $\Sigma(C_1) = \Sigma(C_2)$ ist, d. h. daß die Menge der beschränkten Ereignisse für Prechart und Main Chart per Definition identisch sind.

Nun können wir definieren, wann ein unendlicher Lauf $r = e_1 e_2 \ldots \in \Sigma^\omega$, also ein fortlaufender Strom an Ereignissen, von einem LSC akzeptiert wird:

Definition 2.4. ($r \models_L S$)
Ein unendlicher Lauf $r = e_1 e_2 \ldots$ erfüllt einen LSC S ($r \models_L S$) $\iff$

1. $S = \triangleright C$ und $\exists i : i \geq 1 :\ e_1 \ldots e_i \models_B C$
2. $S = \Diamond C$ und $\exists i, j : 1 \leq i \leq j :\ e_i \ldots e_j \models_B C$
3. $S = \Box(C_1, C_2)$ und $\forall i, j : 1 \leq i \leq j$:

$$(e_i \ldots e_j \models_B C_1) \Longrightarrow (\exists k : j \leq k :\ e_{j+1}, \ldots, e_k \models_B C_2).$$

Beispiel 2.5. Als Beispiel betrachten wir die folgenden Läufe und das Universal LSC aus Abb. 2(a).

$$(\texttt{askCocoa} \cdot \texttt{insertCoin} \cdot \texttt{prepCocoa} \cdot \texttt{serveCocoa})^\omega \tag{1}$$
$$(\texttt{askCocoa} \cdot \texttt{moneyBack})^\omega \tag{2}$$
$$(\texttt{askCocoa} \cdot \texttt{insertCoin} \cdot \texttt{prepCocoa} \cdot \texttt{moneyBack} \cdot \texttt{serveCocoa})^\omega \tag{3}$$
$$\texttt{askCocoa} \cdot \texttt{insertCoin} \cdot \texttt{prepCocoa} \cdot (\texttt{iAmEmpty})^\omega \tag{4}$$

Die Läufe (1) und (2) sind jeweils ein Modell vom LSC in Abb. 2(a). Im ersten Beispiellauf wird der Prechart durch `askCocoa · insertCoin` linearisiert, gefolgt von einer Linearisierung des Main Charts. Im zweiten Beispiellauf wird der Prechart niemals erfüllt. Die Läufe (3) und (4) sind *keine* Modelle des LSCs. In beiden Läufen wird zunächst der Prechart erfüllt, jedoch tritt in (3) das beschränkte Ereignis `moneyBack` im Main Chart auf, und in (4) tritt niemals das Ereignis `serveCocoa` auf.

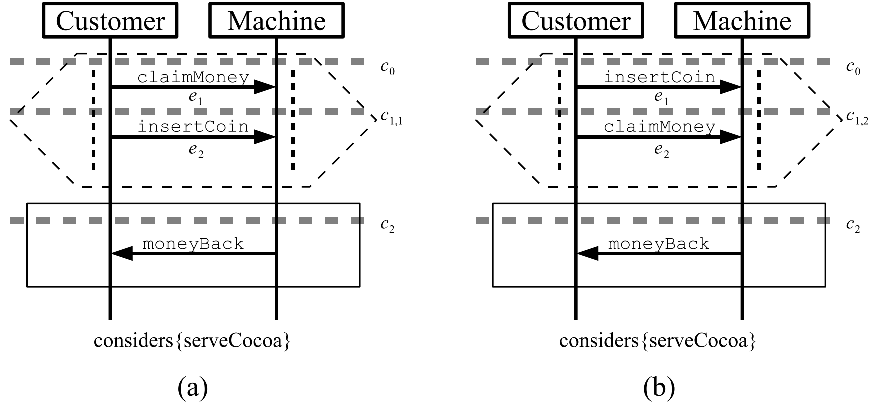


Abbildung 5: Cuts in einem LSC

Eine Spezifikation $\mathcal{S}$ ist eine Menge von LSCs. Sie kann Initial, Existential und Universal Charts enthalten und somit in die drei Mengen $\mathcal{S}_{\triangleright}$, $\mathcal{S}_{\diamond}$ und $\mathcal{S}_{\Box}$ partitioniert werden. Da wir das Ziel haben, aus einer solchen Spezifikation $\mathcal{S}$ ein Programm zu synthetisieren, sind wir insbesondere an den Läufen interessiert, die alle LSCs aus $\mathcal{S}$ erfüllen. Die Menge aller Läufe, welche die Spezifikation erfüllen, fassen wir zu einer Menge R zusammen.

Definition 2.6. ($R \models_S \mathcal{S}$)
Eine Menge unendlicher Läufe R *ist Modell von* $\mathcal{S}$ ($R \models_S \mathcal{S}$) $\iff$

1. $\forall S \in \mathcal{S}_{\triangleright} \cup \mathcal{S}_{\Box} : \forall r \in R : r \models_L S$
2. $\forall S \in \mathcal{S}_{\diamond} : \exists r \in R : r \models_L S.$

2.4 Cuts

Die Semantik eines LSCs hängt jeweils maßgeblich von der Linearisierung der lpo $\mathcal{L}_S$ ab. Da unser Algorithmus solche Linearisierungen handhaben muß, benötigen wir einen Mechanismus, um diese zu erkennen. Dazu führen wir die sogenannten *Cuts* ein.

Ein *Cut* ist die Menge der Positionen, die „schon erreicht“ wurden. Intuitiv kann ein Cut als Zustand eines Charts gesehen werden. Formal ist ein *Cut* in einer lpo $\mathcal{L}_C = \langle [L_C]_\sim, \prec_C, \phi_C \rangle$ eine Teilmenge von Äquivalenzklassen $c \subseteq \mathcal{L}_C$, die *unter Vorgänger abgeschlossen* ist: $\forall l \in c : \forall l' \in \mathcal{L}_C : l' \prec l \Longrightarrow l' \in c$.

In einem Cut c können wir mit einem von einer lpo $\mathcal{L}_C$ beschrifteten Ereignis $e \in \Sigma(C)$ einen weiteren Cut c' erreichen. Dann schreiben wir

$$c \xrightarrow{e} c' \iff \exists l \in [L_C]_\sim : l \notin c, c' = \{l\} \cup c \text{ und } \phi_C(l) = e.$$

Beispiel 2.7. Abb. 5 veranschaulicht Cuts in dem LSC aus Abb. 2(b). Sukzessive können vom Cut $c_0 = \emptyset$ aus weitere Cuts erreicht werden. In dem Chart aus Abb. 2(b) sind von $c_0 = \emptyset$ aus sogar verschiedene Cuts erreichbar, abhängig von der gewählten Linearisierung des Precharts. So gilt (a) $c_0 \xrightarrow{e_1} c_{1,1} \xrightarrow{e_2} c_2$ sowie (b) $c_0 \xrightarrow{e_2} c_{1,2} \xrightarrow{e_1} c_2$.

Den Zusammenhang zwischen Linearisierung und Cut können wir allgemeiner wie folgt formulieren:

Lemma 2.8. *(Linearisierung durch Cuts)*
Sei $\mathcal{L}_C = \langle [L_C]_\sim, \prec_C, \phi_C \rangle$ *eine lpo,* $e_1, \ldots, e_n \in \Sigma(C)$*. Dann gilt* $\emptyset \xrightarrow{e_1} c_1 \xrightarrow{e_2} c_2 \ldots c_{n-1} \xrightarrow{e_n} c_n = \mathcal{L}_C \iff e_1 \cdot \ldots \cdot e_n$ *ist eine* Linearisierung *von* $\mathcal{L}_C$*.*

Dieses Lemma legt nahe zu sagen, daß eine endliche Folge von Ereignissen w einen Cut c *generiert*. Allerdings wollen wir Cuts von beliebigen Suffixen von w generieren lassen.

Definition 2.9. (Generierung von Cuts)
Ein endliches Wort $w \in \Sigma^*$ *generiert* einen Cut c in einer lpo $\mathcal{L}_C$ genau dann, wenn ein Suffix von $w|_{\Sigma(C)}$ eine Linearisierung von c ist.

Die Menge aller Cuts in einer lpo $\mathcal{L}_C$, die von einem Wort $w \in \Sigma^*$ generiert wird, werden wir mit $gen(w, \mathcal{L}_C)$ bezeichnen. Diese Menge ist induktiv wie folgt definiert:

Definition 2.10. ($gen(w, \mathcal{L}_C)$)

$$\begin{aligned} gen(\epsilon, \mathcal{L}_C) &= \{\emptyset\} \\ gen(w \cdot e, \mathcal{L}_C) &= \{\emptyset\} \cup \begin{cases} \{c' \mid \exists c \in gen(w, \mathcal{L}_C), e \in \Sigma(C) : c \xrightarrow{e} c'\} & \text{falls } e \in \Sigma(C) \\ \{c \mid c \in gen(w, \mathcal{L}_C)\} & \text{falls } e \notin \Sigma(C) \end{cases} \end{aligned}$$

Diese Definition von $gen(w, \mathcal{L}_C)$ deckt alle Cuts ab, die von Definition 2.9 erfaßt werden. Denn für die Wahl des leeren Suffixes ist stets $\emptyset \in gen(w, \mathcal{L}_C)$, und wir erreichen von einem Cut c sukzessive alle möglichen weiteren Cuts c' durch $c \xrightarrow{e} c'$, falls $e \in \Sigma(C)$, und bleiben beim bereits erreichten Cut c, falls $e \notin \Sigma(C)$. Somit gilt:

Lemma 2.11. *Für alle* $w \in \Sigma^*$ *und alle lpos* $\mathcal{L}_C$ *gilt:* $c \in gen(w, \mathcal{L}_C) \iff c$ *wird von* w *in* $\mathcal{L}_C$ *generiert.*

Für einen Chart C fassen wir für sämtliche Wörter die Mengen aller generierten Cuts zusammen zu $Gen(C) = \{gen(w, \mathcal{L}_C) \mid w \in \Sigma^*\}$. Beachte, daß diese Menge endlich ist, da jeder LSC nur endlich viele Positionen enthält. Somit gibt es auch nur endlich viele verschiedene Folgen erreichbarer Cuts. Diese Eigenschaft erlaubt uns in Abschnitt 4 einen Spielgraphen auf einem endlichen Zustandsraum zu konstruieren. Daher ist die Anzahl der Elemente von $Gen(C)$ für uns von Bedeutung. In [1] wurde diese wie folgt abgeschätzt:

Satz 2.12. *Für jede lpo* $\mathcal{L}_C$ *eines wohlgeformten Charts* C *gilt*

$$|Gen(\mathcal{L}_C)| = 2^{O(n \log n)},$$

wobei n *die Anzahl der Äquivalenzklassen bzw. der Positionen in* C *ist.*

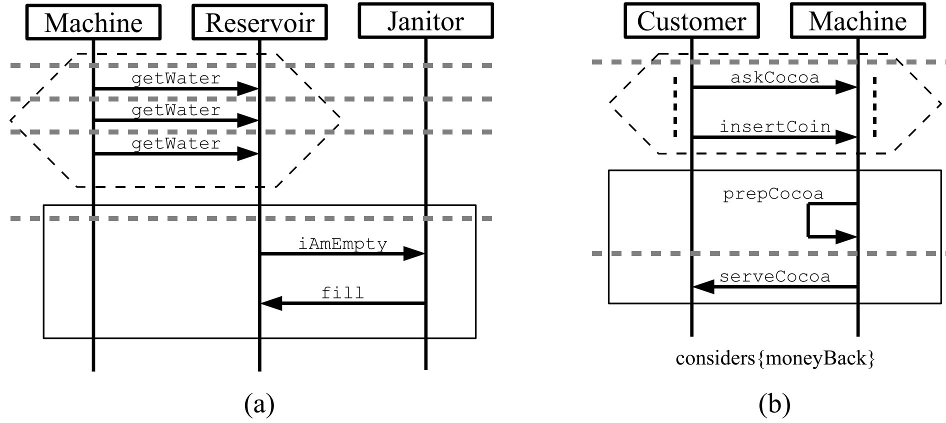


Abbildung 6: Generierte Cuts in LSCs

3 Liveness und Safety

In diesem Abschnitt zeigen wir, daß sich universelle Szenarien durch zwei Eigenschaften ausdrücken lassen: *Liveness* und *Safety*. Liveness ist eine Lebendigkeitsbedingung, die das Auftreten „guter Ereignisse“ verlangt. Safety ist eine Sicherheitsbedingung, die das Auftreten „schlechter Ereignisse“ verbietet. Uns geht es darum, die Erfüllbarkeit eines LSCs nicht anhand unendlicher Läufe, sondern anhand der endlichen Ereignismenge entscheiden zu können. Die Erfüllbarkeit auf die Bedingungen Liveness und Safety zurückzuführen hat vor allem den Vorteil, daß sich genau *eine* Liveness Bedingung und *eine* Safety Bedingung pro Ereignis in Σ ergeben. Dadurch können wir die Verantwortung für die Einhaltung dieser Bedingungen einem bestimmten Ereignis und somit auch einer bestimmten Instanz zuordnen.

Wir werden uns im folgenden auf die für reaktive Systeme wesentlichen Universal LSCs beschränken. Initial LSCs können als Spezialfall von Universal LSCs betrachtet werden, indem diese durch ein einzelnes Ereignis `start` aktiviert werden, welches künstlich an den Anfang jedes Laufs hinzugefügt wird. Die Handhabung von Existential LSCs scheint hingegen etwas aufwendiger zu sein. Laut [2] könnten Existential LSCs jedoch durch ein ähnliches Verfahren, wie das am Ende von [1] beschriebene, berücksichtigt werden.

Beispiel 3.1. Zunächst betrachten wir die folgende Folge von Ereignissen:

$$w = \texttt{getWater} \cdot \texttt{pourWater} \cdot \texttt{getWater} \cdot \texttt{pourWater} \cdot \texttt{getWater} \cdot \texttt{pourWater}$$

Die von w im LSC aus Abb. 2(d) generierten Cuts haben wir in Abb. 6(a) als dicke gestrichelte Linien dargestellt. Dieses Beispiel verdeutlicht, daß von einer Folge von Ereignissen mehrere *verschiedene* Cuts generiert werden können, da jedes Suffix von w betrachtet werden muß. In diesem Universal LSC wird auch ein Cut c erreicht, der schon im Main Chart liegt. In diesem Fall gilt $\mathcal{L}_{C_1} \subseteq c \subset \mathcal{L}_{C_1 \cdot C_2}$. Wir sagen dann, daß der Universal Chart durch einen Cut *aktiviert* wurde, oder daß der Chart *aktiv* ist.

Betrachten wir nun folgende Folge von Ereignissen und das LSC aus Abb. 2(a):

$$w' = \texttt{insertCoin} \cdot \texttt{askCocoa} \cdot \texttt{prepCocoa}$$

Wie Abb. 6(b) zeigt, wird nach dieser Folge ein Cut im Main Chart erreicht, d. h. der Chart ist aktiv. Daher muß in jeder Fortführung von w' irgendwann das Ereignis `serveCocoa` auftreten, um das Universal LSC zu erfüllen. Wir sagen dann, dieses Ereignis ist *notwendig* (engl. *required*). Für dieses Ereignis gilt also eine Lebendigkeitsbedingung.

Genauso dürfen in einer Fortführung von w' gewisse Ereignisse *nicht* auftreten bevor das Ereignis `serveCoca` aufgetreten ist. Dieses sind genau die in dem LSC beschränkten Ereignisse, die jedoch nicht als nächstes Ereignis im Chart auftauchen, also: {`insertCoin`, `askCocoa`, `prepCocoa`, `moneyBack`}. Wir sagen dann, diese Ereignisse sind *verboten* (engl. *forbidden*). Für diese Ereignisse gilt also eine Sicherheitsbedingung.

Wir werden zeigen, daß Läufe, in denen *notwendige* Ereignisse stets irgendwann auftreten und *verbotene* Ereignisse niemals auftreten, gerade die Modelle von Universal LSCs sind. Im folgenden werden wir diese zwei Eigenschaften formalisieren.

Definition 3.2. (Verbotene Ereignisse, Safety)
Betrachten wir eine Spezifikation $\mathcal{S}$ und einen endlichen Lauf $w \in \Sigma^*$. Dann *verbietet* $\mathcal{S}$ das Ereignis $e \in \Sigma$ hinter w ($\mathit{forbid}(w, e)$) genau dann, wenn es ein Szenario $\Box(C_1, C_2) \in \mathcal{S}$ gibt, so daß ein Cut $c \in \mathit{gen}(w, C_1 \cdot C_2)$ existiert, der die folgenden Bedingungen erfüllt:

1. $\mathcal{L}_{C_1} \subseteq c \subset \mathcal{L}_{C_1 \cdot C_2}$, d. h. der Chart ist aktiv;
2. e ist beschränkt durch $\Box(C_1, C_2)$;
3. $\nexists c' : c \xrightarrow{e} c'$.

Ein Lauf $e_1 e_2 \ldots \in \Sigma^\omega$ heißt *e-sicher* genau dann, wenn $\forall i : 1 \leq i : \mathit{forbid}(e_1 \ldots e_i, e) \Longrightarrow e \neq e_{i+1}$.

Definition 3.3. (Notwendige Ereignisse, Liveness)
Eine Spezifikation $\mathcal{S}$ *benötigt* das Ereignis $e \in \Sigma$ nach einem Wort $w \in \Sigma^*$ ($\mathit{require}(w, e)$) genau dann, wenn es ein Szenario $\Box(C_1, C_2) \in \mathcal{S}$ gibt, so daß ein Cut $c \in \mathit{gen}(w, C_1 \cdot C_2)$ existiert, der die folgenden Bedingungen erfüllt:

1. $\mathcal{L}_{C_1} \subseteq c \subset \mathcal{L}_{C_1 \cdot C_2}$, d. h. der Chart ist aktiv;
2. e ist beschränkt durch $\Box(C_1, C_2)$;
3. $\exists c' : c \xrightarrow{e} c'$.

Ein Lauf $e_1 e_2 \ldots \in \Sigma^\omega$ ist *e-lebendig* genau dann, wenn $\forall i : 1 \leq i : \mathit{require}(e_1 \ldots e_i, e) \Longrightarrow \exists k : i < k : e = e_k$.

Satz 3.4. (Liveness + Safety = LSCs)
Für jede Spezifikation $\mathcal{S}$, die nur aus Universal LSC besteht, und jeder Menge aus unendlichen Läufen R gilt

$$R \models_S \mathcal{S} \iff \forall r \in R : \forall e \in \Sigma : \; r \text{ ist } e\text{-sicher und } e\text{-lebendig}.$$

Beweis:

Der Beweis der $\Rightarrow$-Richtung erfolgt durch Anwendung der Definitionen 2.6, 2.4, 2.3, durch Lemma 2.8 und unter Verwendung der Definitionen von Liveness und Safety.

$$\begin{aligned}
& R \models_S \mathcal{S} \\
\Longrightarrow\ & \forall\Box(C_1, C_2) \in \mathcal{S} : \forall r \in R : r \models_L \mathcal{S} \\
\Longrightarrow\ & \forall\Box(C_1, C_2) \in \mathcal{S} : \forall r \in R : \forall i, j : 1 \leq i \leq j : \\
& (e_i \dots e_j \models_B C_1) \Longrightarrow (\exists k : j \leq k :\ e_{j+1} \dots e_k \models_B C_2) \\
\Longrightarrow\ & \forall\Box(C_1, C_2) \in \mathcal{S} : \forall r \in R : \forall i, j : 1 \leq i \leq j : \\
& \Box(C_1, C_2) \text{ ist aktiv } \Longrightarrow \exists k \geq j :\ e'_1 \dots e'_n = e_{j+1} \dots e_k|_{\Sigma(C_2)} : \\
& \emptyset \xrightarrow{e'_1} c_1 \xrightarrow{e'_2} \dots \xrightarrow{e'_n} \mathcal{L}_{C_2}
\end{aligned}$$

Betrachten wir nun ein beliebiges Präfix w eines beliebigen Laufs $r = e_1 e_2 \ldots \in R$, einen zugehörigen Cut $c \in gen(w, C_1 \cdot C_2)$ und ein beliebiges Ereignis $e \in \Sigma$. Nun können wir zwei Fälle unterscheiden: Falls ein Chart durch c aktiviert wurde und e durch $\Box(C_1, C_2)$ beschränkt ist, kann man leicht zeigen, daß r stets e-safe und e-live ist, da in diesem Fall gilt:

$$\exists k \geq j : e'_1 \dots e'_n = e_{j+1} \dots e_k|_{\Sigma(C_2)} : \emptyset \xrightarrow{e'_1} c_1 \xrightarrow{e'_2} \dots \xrightarrow{e'_n} \mathcal{L}_{C_2}$$

Wurde andererseits kein Chart durch c aktiviert oder ist $e \notin \Sigma(C_1 \cdot C_2)$, so ist r wegen den ersten beiden Bedingungen in den Definitionen 3.2 und 3.3 trivialerweise e-sicher und e-lebendig.

Die $\Leftarrow$-Richtung zeigen wir durch Widerspruch. Angenommen $r \in R$ ist e-sicher und e-lebendig, aber $r \not\models_L \mathcal{S}$. Aus letzterem folgern wir, daß ein Szenario $S = \Box(C_1, C_2)$ existiert, so daß Indizes i, j mit folgenden Eigenschaften existieren:

- $r_i \dots r_j|_{\Sigma(S)}$ ist eine Linearisierung von C_1
- $\nexists k \geq j : e_1 \dots e_n = r_{j+1} \dots r_k|_{\Sigma(C_2)} : \emptyset \xrightarrow{e_1} c_1 \xrightarrow{e_2} \dots \xrightarrow{e_n} \mathcal{L}_{C_2}$

Da $\Box(C_1, C_2)$ ein wohlgeformter Chart ist, wird bei jedem Ereignis der Folge $r_{j+1} \dots r_k$ jeweils höchstens ein Cut erreicht. Ebenso gilt für jedes k, daß durch $r_j \dots r_{k+1}|_{\Sigma(C_2)}$ entweder kein neuer Cut erreicht wird oder aber ein neuer Cut erreicht wurde, der größer ist, d. h. der eine Obermenge des vorherigen Cuts ist. Wähle nun das kleinste $k*$, so daß gilt:

- Durch $r_j \dots r_{k*}|_{\Sigma(C_2)}$ wird der Cut $c* \subset \mathcal{L}_{C_2}$ erreicht.
- $\forall k \geq k*$: Das Erreichen eines Cuts c durch $r_j \dots r_k|_{\Sigma(C_2)}$ impliziert $c = c*$.

Offensichtlich muß ein solches $k*$ existieren, da einerseits $r \not\models_L \mathcal{S}$ und andererseits eine Folge von Cuts monoton wachsend über eine *endliche* Menge (hier: $\mathcal{L}_{C_2}$) ist. Da

$c* \subset \mathcal{L}_{C_2}$, muß ein Ereignis e existieren, so daß $c* \xrightarrow{e} c' \subseteq \mathcal{L}_{C_2}$. Nun führen wie eine Fallunterscheidung durch:

$\exists k > k* : r_k \in \mathcal{L}_{C_2}$: Da in diesem Fall $\forall e : c* \xrightarrow{e} c' \subseteq \mathcal{L}_{C_2} \Longrightarrow e \neq r_k$, ist der Lauf r nicht e-sicher. Andererseits hätten wir einen erreichbaren Cut gefunden, der größer als $c*$ ist.

$\forall k > k* : r_k \notin \mathcal{L}_{C_2}$: In diesem Fall gilt insbesondere $\forall k > k* : r_k \neq e$. Das impliziert jedoch, daß der Lauf r nicht e-lebendig ist.

□

4 Synthese

4.1 Problemstellung

Eine Spezifikation, wie die aus Abb. 2, beschreibt sowohl das Verhalten des zu entwickelnden Systems (engl. system under development (SUD)) als auch das Verhalten der Umgebung (engl. Environment). So gehören offensichtlich zum System die beiden Instanzen `Machine` und `Reservoir`, zur Umgebung zählen wir hingegen die Instanzen `Customer`, `Janitor` und `Cup`. In der Softwaretechnik müssen wir diese beiden Typen von Instanzen jedoch unterscheiden, da wir nur das Verhalten des zu entwickelnden Systems implementieren können, nicht aber das Verhalten der Umgebung.

Mit Satz 3.4 haben wir eine Lebendigkeits- und eine Sicherheitsbedingung für jedes Ereignis eingeführt, das in einer LSC Spezifikation auftaucht. Da nun jedes Ereignis durch die Instanz kontrolliert wird, die es sendet, können wir sagen, daß die Bedingungen *Liveness* und *Safety* entweder gegebene Annahmen sind, falls die Instanz zu der Umgebung gehört, oder aber Garantien, die das zu entwickelnde System liefern muß, falls die Instanz zum System gehört.

Der LSC aus Abb. 2 spezifiziert beispielsweise, daß der Wart `Janitor` das `Reservoir` auffüllt, wenn es leer ist. Dieses ist also eine Annahme über die Umgebung. Ebenso wird spezifiziert, daß der Automat `Machine` dem Kunden `Customer` sein Geld zurückgibt, falls dieser eine Münze einwirft und die Geldrückgabe betätigt. Dieses ist also eine Garantie, welche die Implementierung des Systems liefern muß.

Es kann natürlich auch vorkommen, daß eine LSC Spezifikation unerfüllbar ist. Für solche Spezifikationen existiert natürlich keine Implementierung des Systems. Allerdings kann es auch bei erfüllbaren Spezifikation vorkommen, daß eine bestimmte Folge erlaubter Ereignisse der Umgebung zu einem Zustand führt, in dem eine weitere Ausführung unmöglich ist. In einem solchen Fall entsteht dann ein *Deadlock*. Wir werden dieses an einem Beispiel erläutern:

Beispiel 4.1. Betrachten wir wieder die durch Abb. 2 gegebene Spezifikation. Offensichtlich ist diese Spezifikation erfüllbar, z. B. durch den Lauf

$$(\texttt{insertCoin} \cdot \texttt{claimMoney} \cdot \texttt{moneyBack})^{\omega}.$$

Allerdings führt folgendes Präfix zu einem Deadlock:

`insertCoin` · `claimMoney` · `askCocoa` · `prepCocoa` · `addCocoa` · `getWater` · `pourWater`

In diesem Fall fordert das Szenario aus Abb. 2(a), daß `serveCocoa` eintritt und verbietet das Eintreten von `moneyBack`. Abb. 2(b) fordert das Gegenteil, was zum Deadlock führt. Beachte, daß das Verhalten von `Customer` nicht die Annahmen über ihn verletzt.

Wir sehen also, daß selbst bei erfüllbaren Spezifikationen das sture Einhalten der Annahmen über die Umgebung nicht zwangsläufig zu einer Implementierung führen muß. Ein bestimmtes Verhalten der Umgebung entsprechend den spezifizierten Annahmen kann also in unserem Beispiel jede Implementierung zum Absturz bringen.

Da wir jedoch jenseits unserer Spezifikation keine weiteren Annahmen über die Umgebung machen wollen, wollen wir solche Spezifikationen als fehlerhaft erkennen. Spezifikationen, die unerfüllbar sind oder die durch das spezifizierte Verhalten der Umgebung in einen nicht akzeptierenden Lauf gezwungen werden können, nennen wir *inkonsistent*. Wie wir am Beispiel aus Abb. 2 gesehen haben, sind solche Probleme keinesfalls immer offensichtlich, sondern ergeben sich erst durch Interaktion und Überlagerung verschiedener Szenarien. Unser Ziel ist es, diese Inkonsistenzen algorithmisch zu erkennen. Unser Algorithmus wird dabei einen konstruktiven Weg einschlagen, d. h. falls eine LSC Spezifikation konsistent ist, liefert er eine mögliche Implementierung.

4.2 Das Konsistenzproblem

Angenommen die Menge aller Instanzen Ag ist partitioniert in die Mengen E und S, also in die Instanzen der Umgebung und des Systems. Die Ereignismenge Σ partitionieren wir entsprechend in zwei disjunkte Mengen Σ_E und Σ_S, wobei ein Ereignis stets der sendenden Instanz zuzuordnen ist. Zunächst legen wir fest, welche Bedingungen eine Spezifikation $\mathcal{S}$ an die Umgebung stellt.

Definition 4.2. ($\mathbf{WB}(\Sigma_E)$)
Eine Umgebung E heißt *wohlverhaltend* (engl. well-behaving) in einem Lauf r genau dann, wenn

$$\forall e \in \Sigma_E : \; r \text{ ist } e\text{-sicher und } e\text{-lebendig.}$$

Die Menge aller Läufe, in welcher die Umgebung wohlverhaltend ist, bezeichnen wir mit $\mathbf{WB}(\Sigma_E)$.

Wir werden nun das wechselseitige Verhalten von Umgebung und System als ein *unendliches Spiel* [5] auffassen. Dabei nehmen wir an, daß die Umgebung (Spieler 1) dem System (Spieler 0) eine Folge von Ereignissen vorgibt. Falls es sich dabei um eine wohlverhaltende Folge handelt, so muß das System im darauffolgenden Zug ebenfalls entsprechend der Spezifikation mit einer Folge von Ereignissen antworten. Die LSC-Spezifikation ist nun gerade dann konsistent, wenn das System eine Gewinnstrategie hat. Eine Gewinnstrategie liefert direkt eine mögliche Implementierung des spezifizierten Systems.

Formal ist eine Strategie eine Funktion $f : \Sigma^* \to {\Sigma_S}^*$. Das Spiel ist rundenbasiert. Die Umgebung gibt einen endlichen Lauf r vor. Das System antwortet mit $f(r)$ entsprechend

der Strategie f. Nachdem diese Sequenz abgearbeitet ist, liefert die Umgebung eine neue Sequenz r'. Das System antwortet nun mit $f(r \cdot f(r) \cdot r')$.

Durch unser spieltheoretisches Modell können wir reaktive Systeme auf natürliche Weise modellieren. Wir erlauben sogar der Umgebung, mehrere Ereignisse simultan an das System zu übermitteln. Allerdings müssen wir annehmen, daß das System unendlich schneller arbeitet als die Umgebung, d. h. daß der Eingabepuffer des Systems nicht von Ereignissen überflutet wird oder aber Nachrichten bei vollem Puffer verworfen werden.

Das *Resultat* (engl. *outcome*) einer Strategie ist die Menge aller Läufe, die eine Strategie f hervorbringen kann, also formal

$$Out(f) = \{u_0 w_0 u_1 w_1 \dots \mid \forall i \geq 0 : \; u_i \in {\Sigma_E}^+ \text{ und } w_i = f(u_0 w_0 \dots u_i)\}.$$

Nun wollen wir formal beschreiben, wann eine Spezifikation konsistent ist. Dabei sagen wir für $\Sigma' \subseteq \Sigma$, w ist Σ'-sicher bzw. Σ'-lebendig, falls w für alle $e \in \Sigma'$ e-sicher bzw. e-lebendig ist.

Definition 4.3. (Konsistenz)
Eine Spezifikation $\mathcal{S}$, die nur aus universellen Szenarien besteht, ist *konsistent* bezüglich $\mathcal{S}$ genau dann, wenn

$$\exists f : \forall w \in Out(f) : \begin{cases} w \text{ ist } \Sigma_E\text{-sicher} & \Longrightarrow \quad w \text{ ist } \Sigma_S\text{-sicher} \\ w \text{ ist } \Sigma_E\text{-lebendig} & \Longrightarrow \quad w \text{ ist } \Sigma_S\text{-lebendig.} \end{cases}$$

Falls eine solche Strategie f existiert, garantiert Satz 3.4, daß jeder durch f hervorgebrachte Lauf, der die Annahmen über die Umgebung erfüllt, auch die Spezifikation erfüllt. Für eine zu einer konsistenten Spezifikation $\mathcal{S}$ gehörigen Strategie f gilt also stets

$$Out(f) \cap \mathbf{WB}(\Sigma_E) \models_S \mathcal{S}.$$

In Definition 4.3 schließen wir von der Sicherheit der Umgebung auf die Sicherheit des Systems und von den Annahmen über die Lebendigkeit auf die Lebendigkeit des Systems. Man mag sich fragen, warum wir die Definition für Konsistenz nicht wie folgt ansetzen:

$$\Sigma_E\text{-sicher} \wedge \Sigma_E\text{-lebendig} \Longrightarrow \Sigma_S\text{-sicher} \wedge \Sigma_S\text{-lebendig}$$

Für wohlverhaltende Läufe ist diese Bedingung äquivalent zu der aus Definition 4.3, da die linken Seiten der Implikationen stets erfüllt sind. Diese Definition erlaubt jedoch dem System sich beliebig zu verhalten, auch wenn die Umgebung nur eine der Bedingungen erfüllt. Insbesondere darf das System Sicherheitsbedingungen mißachten, falls die Umgebung eine Lebendigkeitsbedingung nicht erfüllt. Betrachten wir hierzu ein kleines Beispiel.

Beispiel 4.4. Wie wollen eine Steuerung für eine Hebebühne bauen. Die Steuerung kann den Motor in drei Zustände versetzen: hoch, runter und neutral. Es wird erwartet, daß die Hebebühne nach unten fährt, wenn der Motor in den Zustand „runter“ versetzt wird. Dieses ist eine Lebendigkeitsbedingung der Umgebung. Weiterhin wird gefordert, daß

sich der Motor im Zustand „neutral“ befindet, wenn die Bremsvorrichtung die Hebebühne blockiert. Dieses ist offensichtlich eine Sicherheitsbedingung. Stellen wir uns nun folgenden Lauf vor. Die Steuerung wird zunächst auf Position „hoch“ gestellt und die Hebebühne fährt hoch. Dann wird diese mit der Bremsvorrichtung blockiert und anschließend wird die Steuerung auf Position „runter“ gesetzt. Dieser Lauf verletzt die Lebendigkeitsbedingung der Umgebung, da die Hebebühne nicht runter fahren kann, wenn sie blockiert wird. In obiger Bedingung ist die linke Seite der Implikation verletzt, und das System muß in diesen Fall weder Lebendigkeits- noch Sicherheitsbedingungen erfüllen. So könnte eine Implementierung z. B. zulassen, daß der Motor in Bewegung gesetzt wird, obwohl er durch die Bremsvorrichtung blockiert wird. Nach Definition 4.3 muß das System jedoch auch in diesem Fall weiterhin seine Sicherheitsbedingung erfüllen. Definition 4.3 ist somit die stärkere Bedingung. Sie unterscheidet klar zwischen Ereignissen, die endlich falsifiziert werden können (Safety), und Ereignissen, die nur durch unendliche Läufe falsifiziert werden können (Liveness).

Dieses Beispiel verdeutlicht auch, daß es durchaus sinvoll sein kann, der Sicherheitsbedingung bei der Definition der Konsistenz ein größeres Gewicht einzuräumen als der Lebendigkeitsbedingung. So könnte man zulassen, daß sich das System beliebig verhalten darf, falls die Umgebung Safety verletzt. Wenn jedoch nur Liveness verletzt wird, sollte sich das System dennoch an seine Sicherheitsbedingungen halten. Für dieses Verhalten kann man die Konsistenz wie folgt modellieren:

$$\Sigma_E\text{-sicher} \Longrightarrow (\Sigma_S\text{-sicher} \wedge (\Sigma_E\text{-lebendig} \Longrightarrow \Sigma_S\text{-lebendig}))$$

Diese Definition von Konsistenz stellt also eine Mischung zwischen den beiden zuvor diskutierten Definitionen dar. Wir werden später sehen, daß der in der Originalliteratur [1] skizzierte Algotithmus auf dieser Definition von Konsistenz beruht, obwohl diese Definition in der Originalliteratur keine Erwähnung findet.

4.3 Algorithmus

In diesem Abschnitt wollen wir zeigen, wie man eine universelle LSC Spezifikation $\{S_1, \ldots, S_n\}$ algorithmisch auf Konsistenz prüfen kann. Bevor wir einen Spielgraphen definieren können, benötigen wir eine Transitionssystem.

Definition 4.5. ($\gamma \overset{e}{\Longmapsto} \gamma'$)
Sei $S = \Box(C_1, C_2)$ ein Universal LSC und $Gen(S) = Gen(C_1 \cdot C_2)$. Dann gelte für alle $\gamma, \gamma' \in Gen(S)$ und alle $e \in \Sigma$ die Transition $\gamma \overset{e}{\Longmapsto} \gamma'$, wenn

- e ist entweder nicht durch S beschränkt, d. h. $e \notin \Sigma(S)$, und es gilt $\gamma = \gamma'$,
- oder e ist durch S beschränkt, d. h. $e \in \Sigma(S)$, und es gilt
  - $\gamma' = \{\emptyset\} \cup \{c' \mid \exists c \in \gamma : c \overset{e}{\longrightarrow} c'\}$
  - und $\forall c \in \gamma :\ \mathcal{L}_{C_1} \subseteq c \subset \mathcal{L}_{C_1 \cdot C_2} \Longrightarrow \exists c' : c \overset{e}{\longrightarrow} c'$

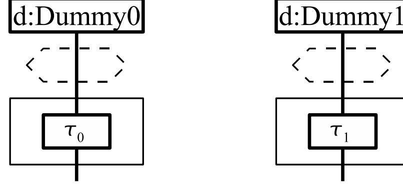


Abbildung 7: Fairness Szenarien.

Man kann leicht zeigen, daß

$$\forall w \in \Sigma^* : \forall \gamma \in Gen(C_1 \cdot C_2) : \{\emptyset\} \overset{w}{\Longmapsto} \gamma \iff w \text{ ist } \Sigma\text{-sicher und } \gamma = gen(w, C_1 \cdot C_2).$$

Um den Spielerwechsel in unserem rundenbasierten Spiel zu modellieren fügen wir zwei zusätzliche „Dummy“-Ereignisse τ_0 und τ_1 ein. Das Ereignis τ_0 markiert das Ende einer Folge von Ereignissen des Systems (Spieler 0), τ_1 steht am Ende einer Ereignisfolge der Umgebung (Spieler 1). Also wird ein Lauf $u_0 w_0 u_1 w_1 \ldots$ mit $u_i \in {\Sigma_E}^+$ und $w_i \in {\Sigma_E}^+$ als $u_0 \tau_1 w_0 \tau_0 u_1 \tau_1 w_1 \tau_0 \ldots$ kodiert. Wir fügen auch der LSC Spezifikation zwei entsprechende Universal LSCs mit leerem Prechart gemäß Abb. 7 hinzu. So müssen τ_0 und τ_1 unendlich oft auftreten. Dieses entspricht einer *Fairnessbedingung*, so daß beide Spieler immer wieder am Zug sind. Alle Ereignisse in Σ_E und Σ_S tragen fortlaufende Indexnummern. τ_0 bzw. τ_1 bezeichnen wir als das $[|\Sigma_S| + 1]$-te bzw. $[|\Sigma_E| + 1]$-te Ereignis.

Unser Spielgraph sieht nun wie folgt aus:

$$G_{\mathcal{S}} = \langle V, V_0, \Delta, \Omega \rangle,$$

wobei

- V die Menge der Knoten ist, die wir hier definieren durch

  $V = (\{0,1\} \times \Sigma \times Gen(S_1) \times \ldots Gen(S_n) \times [|\Sigma_S| + 1] \times [|\Sigma_E| + 1]) \cup \{sink_0, sink_1\}.$

  Ein Knoten der Form $(i, e, \gamma_1, \ldots, \gamma_n, c_0, c_1)$ hat dabei folgende Bedeutung:

  - i kennzeichnet, daß Spieler i am Zug ist ($i = 0$: System, $i = 1$: Umgebung).
  - e ist das zuletzt aufgetretene Ereignis.
  - $\gamma_j (1 \leq j \leq n)$ ist die Menge der Cuts, die derzeitig in S_j erreicht wurden.
  - c_0 ist der Index des Ereignisses aus Σ_S, das als nächstes auf Liveness geprüft werden muß. Dabei erfüllt ein Ereignis mit dem Index c_0 seine Lebendigkeitsbedingung, wenn es als nächstes Ereignis auftritt oder aber nicht notwendig (engl. required) ist. Dann wird c_0 auf $c_0 + 1$ gesetzt. Die Ereignisse werden zyklisch auf Liveness geprüft. Hat das Ereignis $[|\Sigma_S| + 1]$ seine Bedingung erfüllt, so wird c_0 in diesem Zug auf 1 zurückgesetzt.
  - c_1 ist das gleiche wie c_0, jedoch für Ereignisse aus Σ_E.

  Der Senkezustand $sink_i$ bedeutet, daß Spieler i eine seiner Sicherheitsbedingungen verletzt hat.

- V_0 sind die Knoten, an denen Spieler 0 am Zug ist:

$$V_0 = \{sink_1\} \cup \{(i, e, \gamma_1 \dots \gamma_n, c_0, c_1) \in V \mid i = 0\}$$

- Δ ist die Transitionsrelation, welche die folgenden möglichen Züge enthält:
  - Sie enthält Züge der Form

    $$\Delta((i, e, \gamma_1 \dots \gamma_n, c_0, c_1), (i', e', \gamma_1' \dots \gamma_n', c_0', c_1'))$$

    so daß für alle $j \in \{1, \dots, n\}$ die Transition $\gamma_j \overset{e'}{\Longmapsto} \gamma_j'$ gilt, d. h. daß für ein ausgewähltes Ereignis e' Safety erfüllt sein muß. Ganz formal müssen diese Züge folgenden Bedingungen genügen:
    1. $i' = 1 - i \iff e' = \tau_i$.
    2. $e' \in \Sigma_S \iff i = 0$.
    3. $\gamma_j \overset{e'}{\Longmapsto} \gamma_j', (1 \leq j \leq n)$.
    4. $c_0 = |\Sigma_S| + 1 \Longrightarrow c_0' = 1$. Andererseits, sei $a \in \Sigma_S$ das c_0-te Symbol. Falls $e' = a$ oder a für $\gamma_1, \dots, \gamma_n$ nicht notwendig (engl. required) ist, dann $c_0' = c_0 + 1$, sonst $c_0' = c_0$.
    5. Für c_1 bzw. c_1' entsprechend c_0 bzw. c_0', jedoch für die Umgebung E.
  - Falls für Spieler i die Transition $\gamma_j \overset{e'}{\Longmapsto} \gamma_j'$ nicht existiert, d. h. eine Sicherheitsbedingung verletzt wird, so enthält Δ die Transition

    $$\Delta((i, e, \gamma_1 \dots \gamma_n, c_0, c_1), sink_i).$$

  - Es ist nicht möglich einen Senkezustand zu verlassen:

    $$\Delta(sink_i, sink_i)$$

- $\Omega \subseteq V^\omega$ ist die Gewinnbedingung für das System, also die Menge aller unendlichen Läufe, für die Spieler 0 gewinnt. Diese definieren wir mit Hilfe einer sogenannte *Streett Bedindung*. Eine Streett Bedingung ist eine Menge von Paaren von Zustandsmengen: $\{(E_1, F_1), \dots, (E_k, F_k)\}$. Diese Bedingung akzeptiert einen Lauf, wenn für alle $i \in \{1 \dots k\}$ gilt: Wenn E_i unendlich oft besucht wird, dann wird auch F_i unendlich oft besucht. Hier verwenden wir eine Streett Bedingung mit nur einem Paar:

  $$\Omega = \text{Streett}(\{(F_E, F_S)\}) := \{r \in V^\omega \mid \inf(r) \cap F_E \neq \emptyset \Rightarrow \inf(r) \cap F_S \neq \emptyset\}$$

  mit

  - $F_E = \{(i, e, \gamma_1 \dots \gamma_n, c_0, c_1) \mid c_1 = |\Sigma_E| + 1\} \cup \{sink_0\}$
  - $F_S = \{(i, e, \gamma_1 \dots \gamma_n, c_0, c_1) \mid c_0 = |\Sigma_S| + 1\} \cup \{sink_1\}$

Laut [1] entspricht dieses der Definition von Konsistenz nach Definition 4.3. Denn immer wenn unendlich oft ein Zustand aus F_E erreicht wird, so daß Liveness für die Umgebung erfüllt ist, muß das System sich ebenfalls so verhalten. Weiterhin muß das System seine Sicherheitsbedingung erfüllen, wenn die Umgebung ihre Sicherheitsbedingung erfüllt. Denn das Erreichen von $sink_0$ führt dazu, daß die Streett Bedingung niemals erfüllt werden kann. Verletzt jedoch die Umgebung ihre Sicherheitsbedingung, so erreicht das Spiel den Zustand $sink_1$. Dadurch ist die Streett Bedingung stets erfüllt, d. h. das System darf sich in diesem Fall beliebig verhalten. Diese Gewinnbedingung entspricht also abweichend von Definition 4.3 der Konsistenzbedingung

$$\Sigma_E\text{-sicher} \Longrightarrow (\Sigma_S\text{-sicher} \wedge (\Sigma_E\text{-lebendig} \Longrightarrow \Sigma_S\text{-lebendig})).$$

Wir haben jedoch bereits in Abschnitt 4.2 anhand von Beispiel 4.4 diskutiert, daß diese Definition durchaus sinnvoll ist. Wir wollen es deshalb in dieser Ausarbeitung bei der hier definierten Gewinnbedingung Ω belassen.

Unter Berücksichtigung der in der Gewinnbedingung verwendeten Definition von Konsistenz, ist eine universelle LSC Spezifikation $\mathcal{S}$ also genau dann konsistent, wenn

$$\exists f : f \text{ ist eine Gewinnstrategie für Spieler 0 auf } G_{\mathcal{S}} \text{ von } (1, e, \{\emptyset\}, \ldots, \{\emptyset\}, 1, 1)$$

für ein beliebiges $e \in \Sigma$. Eine solche Gewinnstrategie f liefert uns zugleich eine mögliche Implementierung für das System.

Um nun eine solche Gewinnstrategie f algorithmisch zu finden, wollen wir das Spiel $G_{\mathcal{S}}$ in ein *Paritätsspiel* (engl. *parity game*) [5] umformen. Ein Spielgraph eines Paritätsspiels ist ein Spielgraph mit einer *Färbung* $\Omega : V \to \{1, \ldots, k\}$ für ein $k \in \mathbb{N}$, die jedem Knoten eine *Farbe* zuordnet. Spieler 0 gewinnt das Paritätsspiel $r = e_1 e_2 \ldots \in \Sigma^\omega$, wenn die höchste Farbe, die unendlich oft besucht wird, gerade ist, d. h. $\max(\inf(\Omega(r)))$ ist gerade.

Unser Spiel können wir leicht in ein Paritätsspiel transformieren, indem wir folgende Färbung wählen:

- $\Omega(v) = 2$, falls $v \in F_S$;
- $\Omega(v) = 1$, falls $v \in F_E \backslash F_S$;
- $\Omega(v) = 0$, sonst.

Nun können wir auf die vorhandenen Algorithmen zurückgreifen, um Paritätsspiele zu lösen. Algorithmen sind z. B. in [5] beschrieben.

4.4 Komplexität

Das Konsistenzproblem ist sowohl co-NP-schwer als auch NP-schwer. Die entsprechenden Reduktionen von 3-SAT auf das negative und das positive Konsistenzproblem finden sich in [1].

Gewinnstrategien für Paritätsspiele mit nur 3 Farben lassen sich in polynomineller Zeit bezüglich der Anzahl der Knoten finden. Die Anzahl aller Folgen erreichbarer Cuts haben wir bereits durch Satz 2.12 mit $|Gen(\mathcal{L}_C)| = 2^{O(n \log n)}$ abgeschätzt, wobei n die Anzahl der Äquivalenzklassen bzw. der Positionen im Chart ist. Somit können wir die Anzahl der Knoten in unserem Spielgraph $G_{\mathcal{S}}$ und somit die Laufzeit unseres Algorithmus abschätzen durch

$$O(|\Sigma|\ |\Sigma_S|\ |\Sigma_E|\ 2^{s\, n \log n}),$$

wobei s die Anzahl der Szenarien in einer Spezifikation und n die maximale Anzahl an Positionen in einem Chart ist.

5 Zusammenfassung und Ausblick

Wir haben Live Sequence Charts als eine mächtige Spezifikationssprache für reaktive Systeme vorgestellt. Insbesondere Universal LSCs sind für die Spezifizierung reaktiver Systeme geradezu prädestiniert. Wir haben die Syntax und Semantik von LSCs erläutert. Durch die Halbordnungen sind oft eine Vielzahl verschiedener Folgen von Ereignissen möglich. Um diese formal in den Griff zu bekommen, haben wir die sogenannten Cuts eingeführt.

Es ist uns gelungen, die Erfüllbarkeit eines Universal LSCs auf zwei Eigenschaften pro Ereignis zurückzuführen: Liveness und Safety. Dadurch kann die Verantwortung für die Einhaltung dieser Bedingungen einer bestimmten Instanz zugeordnet werden. Somit kann klar zwischen der Verantwortung der Umgebung und der Verantwortung des Systems unterschieden werden.

Spezifikationen können inkonsistent sein. In einem solchen Fall ist es trotz einer wohlverhaltenden Umgebung unmöglich, eine Implementierung für das System zu finden. Die Unterscheidung zwischen Liveness und Safety hat uns auch dazu gebracht, verschiedene Varianten von Definitionen der Konsistenz gegenüberzustellen.

Zum Schluß haben wir einen Algorithmus vorgestellt, der solche Inkonsistenzen ausmerzen kann. Dazu haben wir das Konsistenzproblem als ein unendliches rundenbasiertes Spiel modelliert. Die Umformung in ein Paritätsspiel erlaubt uns die Verwendung effizienter Algorithmen. Der Algorithmus liefert eine Implementierung, falls die Spezifikation konsistent ist, oder aber einen Sabotageplan, falls sie inkonsistent ist.

In der Originalliteratur [1] wird noch der Aspekt der *Mercifulness* berücksichtigt. Es kann in manchen Fällen möglich sein, daß unser Algorithmus eine Strategie liefert, die implementierbar ist, weil sie die Umgebung daran hindert gewisse Ereignisse unendlich oft auszuführen. Daher wird in [1] der Algorithmus um *Mercifulness* erweitert, so daß Strategien, die dieses unerwünschte Verhalten nicht aufweisen, bevorzugt werden. Wir wollen hier aber nicht näher darauf eingehen.

Weiterhin haben wir nur Universal LSCs behandelt, die für eine LSC Spezifikation eines reaktiven Systems wesentlich sind. Initial LSCs können jedoch gewissermaßen als Spezialfall von Universal LSCs gesehen werden. Die Handhabung von Existential LSCs sollte hingegen mit einer ähnlichen Erweiterung, wie die der *Mercifulness* aus [1], möglich sein.

Literatur

[1] Yves Bontemps, Pierre-Yves Schobbens, and Christof Löding. Synthesizing open reactive systems from scenario-based specifications. *Fundamenta Informaticae*, XX:1–31, 2004.

[2] Yves Bontemps and Pierre-Yves Schobbens. Synthesizing open reactive systems from scenario-based specifications. In Felice Balarin and Johan Lilius, editors, *Proc. of the 3rd ACSD'03*, pages 41–50, Guimarães, Portugal, June 2003. IEEE Computer Science Press.

[3] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[4] David Harel and Hillel Kugler. Synthesizing state-based object systems from LSC specifications. *Int. J. of Foundations of Computer Science*, 13(1):5–51, 2002. (also appeared in CIAA 2000, LNCS 2088, pp. 1–33, Springer, 2001).

[5] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games.* Number 2500 in Lecture Notes in Compter Science. Springer-Verlag, 2002.