Mobile Systeme und der π - Kalkül

Schwache Äquivalenz: Beispiele

Patrick Wischnewski
pawi5002@stud.uni-saarland.de
Betreuer: Prof. Gert Smolka

17. Oktober 2004

Wenn man über verschiedene Prozesse spricht möchte man diese miteinander vergleichen, um eine Aussage darüber treffen zu können, ob zwei gegebene Prozesse äquivalent sind, also das gleiche tun. Zu erkennen, ob zwei Prozesse das gleiche tun ist oft sehr schwierig. Weiter benötigen wir einen adäquaten Äquivalenzbegriff auf Prozessen, der definiert wann zwei Prozesse das Gleiche tun. Aus diesem Grund werden wir in diesem Artikel einige Beispiele von Systemen und deren Spezifikation betrachten. Die Spezifikation wird durch Prozesse dargestellt, die nur das Verhalten des Systems gegenüber dem Benutzer darstellen und dabei Implementierungsdetails vernachlässigt. In dem System sind dann alle zur konkreten Umsetzung nötigen Details beschrieben. Ziel ist es dann zu Beweisen, dass System und Implementierung äquivalent sind, also das gleiche tun.

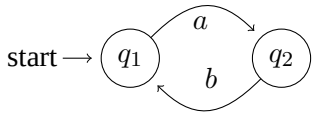
$$System \approx Implementierung$$

Inhaltsverzeichnis

1 Äquivalenz von Systemen

Ziel dieses Abschnittes wird es sein einen angemessenen Äquivalenzbegriff auf Prozessen zu definieren.
Prozesse werden hier mit CCS beschrieben. Mit CCS beschriebene Prozesse lassen sich sehr anschaulich durch Transitionssysteme darstellen. Dabei werden alle Unterausdrücke eines CCS - Ausdrucks als Knoten dargestellt und alle Übergänge der Form $\xrightarrow{\alpha}$ werden durch Kanten dargestellt.

Wir betrachten dazu folgendes Beispiel:



$$S = a.b.S$$

Abbildung: Semaphore

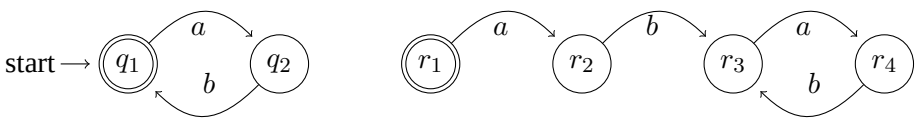
In der Abbildung sieht man die CCS - Beschreibung: $S = a.b.S$ des Systems und der dazu äquivalente Transitionsgraph. Dieses System funktioniert jetzt wie folgt: Man kann hier ein a lesen und dann ein b. Dies kann man beliebig oft tun. Das Lesen der a bzw. b beschreibt hier das Verhalten des Systems dem Benutzer gegenüber.

1.1 Starke Äquivalenz

Definition 1 *Eine Relation S ist eine starke Simulation wenn für alle Paar pSq der Relation gilt: Wenn $p \xrightarrow{\alpha} p'$ dann existiert ein q' mit $q \xrightarrow{\alpha} q'$ und $p'Sq'$. Dabei sind p, q, p' und q' Knoten.*

Definition 2 *Eine binäre Relation S ist eine starke Äquivalenz bzw. starke Bisimulation, wenn S und S^{-1} eine starke Simulation sind.*

Diese Definition meint anschaulich gesehen, dass man in zwei Prozessen immer die gleichen Transitionen durchführen kann.
Betrachte dazu folgendes Beispiel:



$S = a.b.S$ $\qquad$ $P = a.b.S$

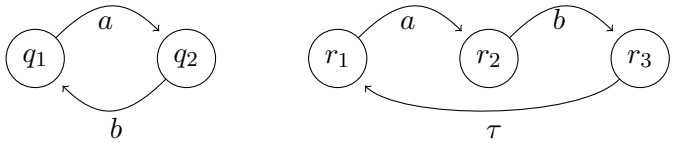
Abbildung: zwei stark äquivalente Prozesse

Man sieht hier, dass man in S und P jeweils die gleichen Transitionen durchführen kann. Man kommt von q_1 mit a nach q_2 und mit a von r_1 nach r_2. ($q_1 \xrightarrow{a} q_2$ und $q_1 \xrightarrow{a} q_2$). Somit entsprechen sich die Zustände q_1 und r_1. Für q_2 und r_2 gilt, dass diese Zustände sich ebenfalls entsprechen, denn $q_2 \xrightarrow{b} q_1$ und $r_2 \xrightarrow{b} r_3$. Genau so sieht man, dass sich q_1 und r_3 entsprechen und q_2 und r_4. Somit gibt es für jede Transition in S eine entsprechende in P und umgekehrt. Somit sind beide Systeme stark äquivalent.

1.2 Schwache Äquivalenz

Mit dem Begriff der starken Äquivalenz hätten wir also einen möglichen Äquivalenz - Begriff gefunden mit dem wir eine Aussage über die Äquivalenz von Prozessen treffen können. Dabei stellt sich die Frage ob dieser Äquivalenzbegriff genügend Systeme äquivalent macht. Oder ob es Prozesse gibt, die man noch als äquivalent ansieht, die nach dieser Definiton aber nicht äquivalent sind?

Betrachten wir dazu folgendes Beispiel:



$S = a.b.S$ $P' = a.b.\tau.P'$

Abbildung: zwei Schwachäquivalente Prozesse

Wir können hier sowohl von q_1 als auch von r_1 ein a lesen und kommen dann nach q_2 bzw. r_2. Hier können wir in beiden Fällen ein b lesen und kommen dann zu den Zuständen q_1 und r_3. Hier ergibt sich jetzt ein Problem. Der Zustand r_3 hat keinen entsprechenden in S. Denn von r_3 kommt man nur mit einer τ - Transition ($\xrightarrow{\tau}$) wieder zu r_1. Nach Definition der starken Bisimulation sind diese beiden Prozesse nicht äquivalent.
Sollten diese Prozesse als äquivalent angesehen werden? Da die τ-Transition nicht beobachtet werden kann, also vor dem Benutzer verborgen bleibt, machen die beiden Systeme vom Standpunkt des Benutzers das Gleiche.
Dies führt uns zu der folgenden Definiton der schwachen Äquivalenz.

Definition 3 *Die Relationen $\Rightarrow$ und $\overset{s}{\Rightarrow}$ sind wie folgt definiert:*

- *$P \Rightarrow Q$ bedeutet, dass es eine Sequenz von null oder mehr Reaktionen $P \rightarrow ... \rightarrow Q$ gibt. Formal: $\Rightarrow \overset{def}{=} \rightarrow^*$*
- *Sei $s = \alpha_1 ... \alpha_n$ Dann bedeutet $P \overset{s}{\Rightarrow} Q$, dass $P \Rightarrow \xrightarrow{\alpha_1} P_1 ... \Rightarrow \xrightarrow{\alpha_n} P_n \Rightarrow Q$ gilt. Formal: $\overset{s}{\Rightarrow} = \Rightarrow \xrightarrow{\alpha_1} \Rightarrow ... \Rightarrow \xrightarrow{\alpha_n} P_n \Rightarrow$*

- *Eine Relation S ist eine schwache Simulation wenn für alle Paar pSq der Relation gilt: Wenn $p \rightarrow p'$ dann existiert ein q' mit $q \Rightarrow q'$ und $p'Sq'$ und wenn $p \stackrel{\alpha}{\rightarrow} p'$ dann existiert ein q' mit $q \stackrel{\alpha}{\Rightarrow} q'$ und $p'Sq'$. Dabei sind p, q, p' und q' Knoten.*

Definition 4 *Eine binäre Relation S ist eine schwache Äquivalenz bzw. schwache Bisimulation, wenn S und S^{-1} eine schwache Simulation sind.*

Die schwache Äquivalenz bezieht sich nur noch auf das beobachtbare Verhalten, also die Reaktionen ohne τ - Transitionen. Mit diesem Äquivalenzbegriff sind die beiden Systeme in der Abbildung oben äquivalent, da sich bei beiden das gleiche Verhalten beobachten läßt.

2 Beispiel: Lotterie

2.1 Lotterie

Wir wollen den Begriff der *Schwachen Äquivalenz* am Beispiel einer Lotterie-Trommel näher untersuchen.
Eine Lotterie-Trommel besteht im wesentlichen aus der Trommel, die dich solange dreht bis sie eine Ball auswirft auf dem eine Nummer steht. Nehmen wir bei dieser Lotterie an, dass ein Ball mehrfach geworfen wird. Er wird nachdem er von der Trommel geworfen wurde wieder in die Trommel zurück gelegt wird. Weiter vereinfachen wir die Lotterie so, dass wir davon ausgehen dass es nur drei Bälle in dem System gibt. Das linke System in der unten stehenden Abbildung zeigt dieses System.
Der Zustand q_1 ist der Zustand in dem sich die Trommel dreht. Bis sie schließlich einen τ-Übergang macht und in einen der anderen Zustände kommt; wobei die anderen Zustände die Zustände sind, in denen ein Ball b_1, b_2 oder b_3 durch den τ-Übergang ausgeworfen wurde und nun darauf warten, dass der ausgeworfene Ball wieder in die Trommel gelegt wird. Dies stellt die einige Interakton des Systems mit der Außenwelt da, also die einzige Stelle des Systems an der man eine Aktion beobachgten kann.

Das zweite System beschreibt, eine mögliche Implementierung. Hier in unserem Beispiel beschreiben die τ-Übergänge zwischen den Knoten L_1,L_2 und L_3 das Drehen der Lotto-Trommel. Wir sehen das sich hier im Gegensatz zu dem linken System eine konkrete Umsetzung der Lotto-Trommel befindet. Ansonsten ist stimmen die beiden Systeme überein.
Jetzt stellt sich also hier die Frage: Sind die beiden Systeme schwach äquivalent; verhalten sie sich nach außen gleich?

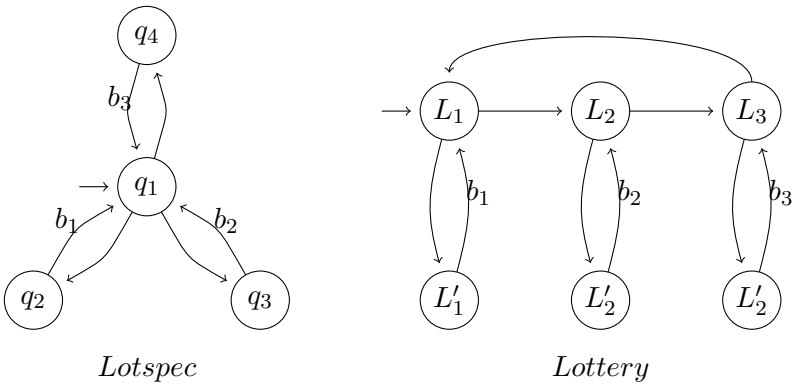


Abbildung: Spezifikation und Implementierung einer Lotterie-Trommel

Wie schon erwähnt entsprechen sich die Zustände q_2 und L'_1, q_3 und L'_2 sowie q_4 und L'_3. Wie sieht es mit dem Verhalten der anderen Zustände aus? Was kann man

hier beobachten?
Man kann bei den Transitionen zwischen den Zuständen L_1, L_2 und L_3, die das "Drehen"der Lotto-Trommel darstellen von außen nichts beobachten, da es sich um τ-Übergänge handelt. Diese können also aus Sicht des Beobachters zu einem Zustand zusammen gefaßt werden. Dann haben wir genau die Situation wie in der Spezifikation. Die beiden Prozesse sind also „Schwach Äquivalent".
Das selbe Argument gilt auch für Lotterien mit mehr als drei Bällen.

Für den Fall $n = 3$ würde man formal wie folgt argumentieren:

Theorem 1 $L_1 \approx Lotspec$

Beweis:
Wir zeigen, dass folgende Relation eine *Schwache Bisimulation* ist:
$S = \{(L_i, Lotspec) | 1 \leq i \leq 3\} \cup \{(L'_i, b_i.Lotspec) | 1 \leq i \leq 3\}$
Wir betrachten dazu das Paar $(L'_1, b_1.Lotspec)$. Wir müssen zeigen, dass zu jeder Transition $\stackrel{\lambda}{\rightarrow}$ oder $\rightarrow$ der ersten Komponente eine entsprechende Transition $\stackrel{\lambda}{\Rightarrow}$ oder $\Rightarrow$ der zweiten Komponente gibt. Aber jede der Komponenten hat nur eine Transition: $L'_1 \stackrel{b_1}{\rightarrow} L_1$ und $b_1.Lotspec \stackrel{b_1}{\rightarrow} Lotspec$. Diese entsprechen sich, denn $(L_1, Lotspec) \in S$. Dies gilt genauso für die Paare $(L'_2, b_2.Lotspec)$ und $(L'_3, b_3.Lotspec)$.

Nun betrachten wir das Paar $(L_1, Lotspec)$. Auch hier müssen wir wieder zeigen, dass es zu jeder Transition der einen Komponente eine entsprechende in der anderen gibt.
$L_1 \rightarrow L_2$ entspricht $Lotspec \Rightarrow Lotspec$
$L_1 \rightarrow L'_1$ entspricht $Lotspec \rightarrow b_1.Lotspec$
Für die andere Komponente gilt:
Für jedes $i \in \{1, 2, 3\}$ gilt: $Lotspec \rightarrow b_i.Lostspec$ entspricht $L_1 \Rightarrow L'_i$
Dies gilt genauso für die Paar $(L_2, Lotspec)$ und $(L_3, Lotspec)$

■

2.2 Fairness

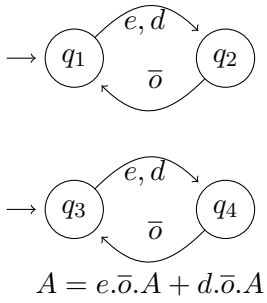
Bei genauerem betrachten der beiden Prozesse in der oberen Abbildung stellt man fest, dass die Implementierung *Lottery* eine mögliche Divergenz enthält. Wie sieht es mit der Äquivalenz aus, wenn die *Lottery* unendlich oft in den Zuständen L_1, L_2 und L_3 bleibt? Nehmen wir an das System *Lotspec* macht einen Übergang von q_1 in einen der Zustände $q_i \in 2, 3, 4$ und die Implementierung *Lottery* divergiert. Dann sind beide Systeme nicht mehr äquivalent.
Die beiden Systeme sind nur mit einer zusätzlichen**Fairness - Annahme** äquivalent, d.h. dass eine mögliche Transition in einem System nicht unendlich oft vermieden wird. Die Definition der schwachen Äquivalenz alleine reicht also nicht aus, um diese Prozesse äquivalent zu machen. Man benötigt zusätzlich noch die Annahme, dass beide System *fair* sind.

3 Beispiel: Job Shop

In einer Fabrik arbeiten zwei Argenten an einem Fliesband zusammen. Sie erhalten Jobs, bearbeiten diese und schicken die bearbeiteten Jobs wieder auf das Band. Dabei gibt es einfache (e) und schwierige (d) Jobs. Sie erhalten Jobs über den Port e bzw. d und schicken bearbeitete durch den Port $\overline{o}$ wieder aufs Band. Diese Agents unterscheiden nicht zwischen einfachen und schweren Jobs. Ein Agent ist also ein einfaches Semaphore, ähnlich dem aus Kapitel 1, was man an der folgenden Defintion sieht:

$$A = e.\overline{o} + d.\overline{o}$$

Die Firma besteht aus zwei Argenten, die parallel und unabhängig von einander arbeiten. Es sind zwei nebeneindander laufende Prozesse wie die folgende Abbildung zeigt.



$$A = e.\overline{o}.A + d.\overline{o}.A$$

Job Shop: $A|A$

Abbildung: Spezifiaktion Job Shop

Betrachten wir nun ein zweites System, indem wir die Agenten durch Jobber ersetzen. Die Jobber unterscheiden jetzt zwischen schweren (d) und einfachen (e) Jobs. Zum Bearbeiten eines schweren Jobs ist eine zusätzliche Ressourse (z.B. ein Werkzeug) erforderlich. Nehmen wir an dass die Jobber für die Verarbeitung eines schweren Jobs einen Hammer benötigen, den sie Aufnehmen (gh) und nach Beendigung des Jobs wieder zurücklegen. (ph). Eine weitere Einschränkung ist, dass es in diesem System nur einen einzigen Hammer gibt. Der Hammer ist wieder ein einfaches Semaphore. Die Implementierung des Systems zeigt die folgende Abbildung, die aus drei nebenläufigen Prozessen besteht, zwei Jobbern und einem Prozess, der den Hammer darstellt:

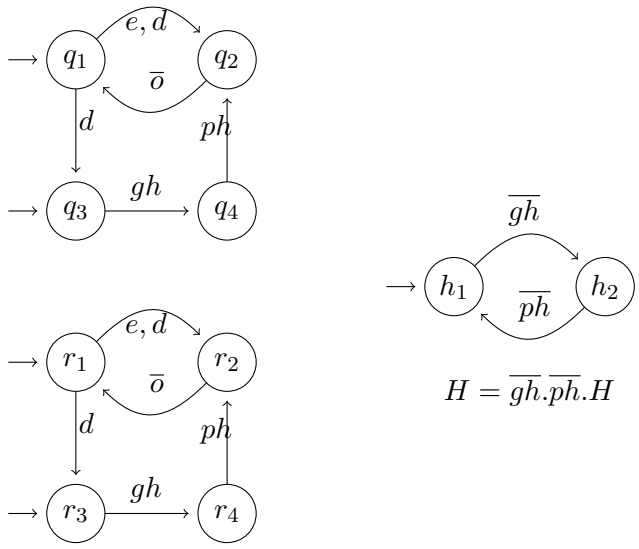


Abbildung: Implementierung Job Shop

Arbeiten diese beiden Systeme äquivalent? Was man sofort sieht ist, wenn nur einfache Jobs zu bearbeiten sind, zeigen beide System das gleiche Verhalten. Im anderen Fall ist es so, dass immer wenn ein Prozess einen einfachen Job und der andere einen schweren Job bekommt, so dass in der Implementierung nie zwei Prozesse gleichzeitig den Hammer benötigen arbeiten Spezifikation und implementierung ebenfalls äquivalent. Wenn in der Implementierung beide Jobber gleichzeitig auf den Hammer zugreifen wollen, muss einer der beiden Prozesse darauf warten, dass der andere den Hammer wieder frei gibt. Aber dieses „Warten“ spielt beim Begriff der *Schwachen Äquivalenz* keine Rolle. Denn es ist nicht spezifiziert in welcher Reihenfolge die Jobber die bearbeiteten Jobs wieder auf das Fliesband legen. Das einzige was spezifiziert ist, ist wann der Beobachter eine $\overline{o}$ Transition beabachten kann, bei welchem der beiden Jobber-Prozesse er dies beobachtet spielt keine Rolle. Also sind Spezifikation und Implementierung äquivalent.

Noch eine kleine Anmerkung: Wie schon oben erwähnt können Jobs in der Implementierung nur in einer bestimmten Reihenfolge abgearbeitet werden. Wohingegen in der Spezifikation jede Reihenfolge auftreten kann.

4 Beispiel: Scheduler

Ein Scheduler kontrolliert n Tasks. Die Tasks können von dem Scheduler gestartet und beendet werden. Um den Task i zu Starten wird die Aktion a_i durchgeführt und zum Beenden die Aktion b_i. Weiter können die Tasks nur in einer bestimmten Reihenfolge gestartet werden. Task $i+1$ darf erst nach Task i gestartet werden (i ist hier modulo n zu verstehen).

Beim Beenden gibt es keine Einschränkungen. Der Task i kann zu jedem Zeitpunkt sofern er gerade läuft durch durchführen der entsprechenden Aktion b_i beendet werden.

Für den Fall $n = 1$, wenn wir genau einen Task haben, ist unser System wieder ein Semaphore ($A = a_1.b_1.A$), mit dem man den Task 1 starten und dann wieder beenden kann; dies kann man dann beliebig oft durchführen.
Für den Fall $n = 2$ beschreibt die untere Abbildung das gewünschte Verhalten. Dabei gibt jeder Zustand an, welcher Task als nächstes gestartet werden kann und welche Tasks gerade Laufen. Um dies in der untern Abbbildung zu verdeutlichen steht bei jedem Knoten ein Tupel, wobei die erste Komponente angibt welcher Task als nächstes gestartet werden kann und die zweite ist die Menge der aktuell laufenden Tasks.

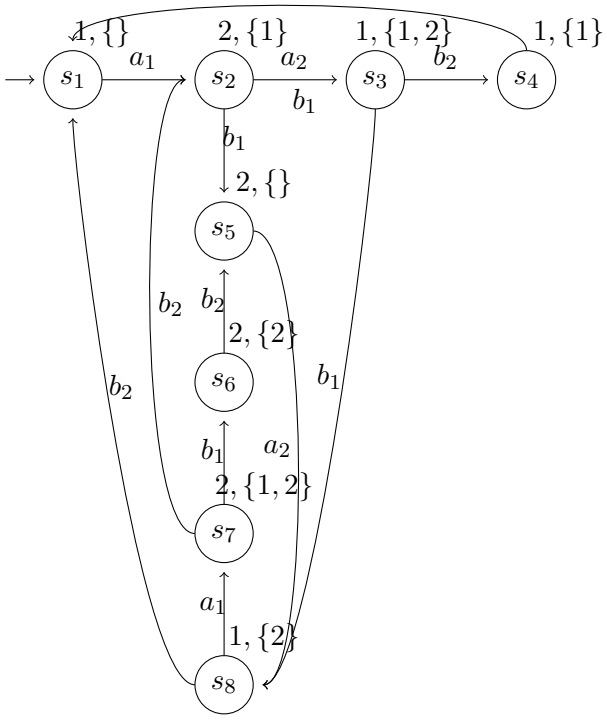
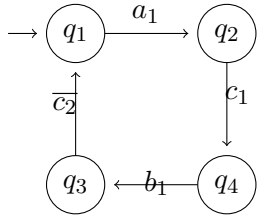


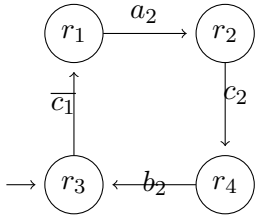
Abbildung: Spezifikation Scheduler

Eine mögliche, aus parallel laufenden Prozessen aufgebaute, Implementierung zeigt die untere Abbildung. Dabei ist die Idee, dass man für jeden zu managenden Task einen eigenen Prozess hat. Jeder Pozeß kann einen Task starten und wieder beednen. Wenn er einen Task gestartet hat, teilt er dies dem nächsten Prozess mit, der als nächstes einen Task starten darf.

Initial befinden sich als alle Prozesse mit Ausnahme des Prozesse, der Task 1 kontrolliert in einem „Warte-Zustand “. Dieser kann den Task 1 starten und dann dem Prozess, der den Task mit der Nummer 2 kontrolliert mitteilen, dass dieser jetzt gestartet werden kann und sich selbst in einen Warte-Zustand versetzten, in dem er darauf wartet, dass der Prozess für den letzten Task ihm mitteilt, dass der erste Task wieder gestartet werden kann.



$A_1 = a_1.c_1.b_1.\overline{c_2}.A_1$



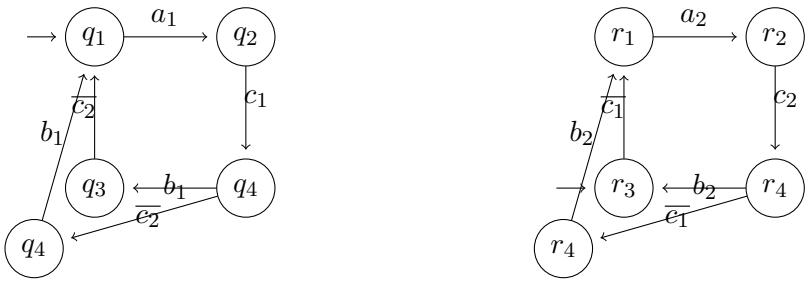
$A_2 = a_2.c_2.b_2.\overline{c_1}.A_2$

Scheduler: $new\ c_1c_2(A_1, \overline{c_1}.A_2)$

Abbildung: Implementierung Scheduler

Sind hier Spezifikation und Implementierung äquivalent? Betrachten wir dazu mal die Eingabe a_1, a_2, in der der Tasks 2 gestartet wird ohne dass 1 vorher beendet wird. Wenn wir also die Aktion a_1 durchführen, führt die Implementierung einen Übergang $q_1 \stackrel{a_1}{\rightarrow} q_2$ durch und dann macht sie einen synchronisierten Übergang mittels c_1, so dass sich die Implementierung im Prozess A_1 im Wartezustand q_4 befindet und im Prozess A_2 im Zustand r_1. Jetzt kann die Transition $r_1 \stackrel{a_2}{\rightarrow} r_2$ durchgeführt werden. In der Spezifiaktion können wir die Transitionen $s_1 \stackrel{a_1}{\rightarrow} s_2 \stackrel{a_2}{\rightarrow} s_3$ durchführen. Bis hier sind die beiden Systeme äquivalent.
In der Spezifikation können wir jetzt entweder die Aktion $s_3 \stackrel{b_1}{\rightarrow} s_8$ oder die Aktion $s_3 \stackrel{b_2}{\rightarrow} s_4$ durchführen. In der Implementierung können wir allerdings in der aktuellen Konfiguration nur ein b_1 lesen. Denn um ein b_2 lesen zu können müßten wir zunächst über c_2 mit A_1 reagieren. Die Implementierung erfüllt hier also **nicht** die Spezifikation.

Wie können wir dieses reparieren? Damit wir an dieser Stelle das richtige Ergebnis bekommen, müßte die Reihenfolge in der b_1 und $\overline{c_2}$ bzw. b_2 und $\overline{c_1}$ auftreten können beliebig sein. So könnten wir in der Situation oben entweder, wie vorher auch schon möglich, ein b_1 lesen oder wir könnten zunächst mittels dem Kanal c_2 eine Reaktion durchführen und dann b_2 lesen. Dies zeigt also genau das Verhalten, dass von der Spezifikation verlangt wurde. Die Änderung verdeutlicht die folgende Abbildung.



$A_1 = a_1.c_1.b_1.\overline{c_2}.A_1 + a_1.c_1.\overline{c_2}.b_1.A_1$ $\quad$ $A_2 = a_2.c_2.b_2.\overline{c_1}.A_2 + a_2.c_2.\overline{c_1}.b_2.A_2$

Scheduler: $new\ c_1c_2(A_1, \overline{c_1}.A_2)$

Abbildung: reperierte Implementierung Scheduler

An diesem Beispiel sieht man, dass es nicht immer einfach ist einen solchen Fehler in der Implementierung zu finden. Denn wenn wir einen Scheduler implementieren wollten der viel mehr Tasks kontrolliert, könnte es in der Praxis so aussehen, dass die Situation, in welcher der Fehler passiert, lange Zeit nicht auftritt, das System also anscheinend das richtige tut. Sollte dann die Situation einmal auftreten ist es sehr schwierig nachzuvollziehen, warum das System sich nicht wie von der Spezifikation gefordert verhält.

5 Beispiel: Buffer

Ein Buffer ist eine Datenstruktur in der man Daten speichern kann, um sie dann wieder auszulesen. Wir betrachte hier beispielhaft, um das System einfach zu halten, einen unären Buffer der Kapazität 2. Das System speichert also, ob es keinen, einen oder zwei Einträge hat. Das Verhalten eines solchen Buffers zeigt die folgende Abbildung.

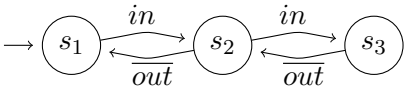


Abbildung: Spezifikation Buffer

Eine mögliche Implementierung mit parallel laufenden Prozessen zeigt die unten stehende Abbildung. Die Idee dabei ist hier, dass man die Prozesse hintereinander hängt und nur der erste die Aktion in durchführen kann und nur der letzte die Aktion $\overline{out}$. Die Prozesse sind über einen Linkport $l, \bar{l}$ miteinander verbunden. Wenn beim Eingang in eine Aktion stattfindet wird die Information, dass Daten in der Struktur sind mittels des Linkports an den anderen Prozess weitergeleitet und der Prozess am Anfang ist wieder bereit eine weitere in Aktion durchzuführen, sofern der Buffer nicht voll ist.

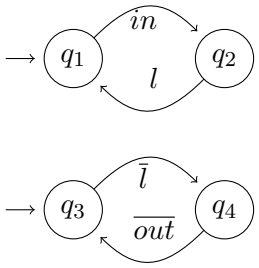


Abbildung: Implementierung Buffer

Hier erfüllt die Implementierung die Spezifikation. Der Buffer läßt sich jetzt leicht auf den Fall n übertragen, indem man in der Mitte noch für jeden Speicherplatz ein Prozess einfügt, der dann mittels Linkports mit seinen beiden Nachbarprozessen verbunden ist. Wenn Daten am Anfang eingefügt werden, wird diese Information mittels der Linkports soweit wie möglich zum Ende durchgereicht.

6 Beispiel: Counter

Im vorherigen Abschnitt haben wir mittels paralleler Prozesse einen unären Buffer mit Kapazität 2 implementiert. In diesem Abschnitt wollen wir einen endlichen Counter aus parallel laufenden Prozessen implementieren, der ebenfalls die Größe 2 hat, mit dem man also bis 2 Zählen kann. Im wesentlichen entspricht die Spezifikation, der des Buffers nur, dass es hier zusätzlich noch einen Test auf Leerheit ($\overline{empty}$) und einen Test gibt, der testet ob der Maximalwert des Counters erreicht wurde ($\overline{full}$). Die Spezifikation zeigt die folgende Abbildung.

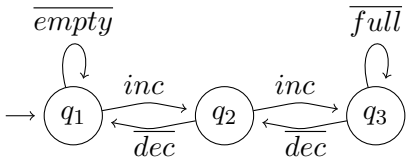


Abbildung: Spezifiaktion Counter

Die Implementierung für nebenläufige Prozesse ist jetzt wegen der Tests nicht mehr ganz so einfach. Wir haben jetzt drei unterschiedliche Prozesse, die mittels Linkports miteinander verbunden sind. Den Prozess A mit Leerheitstest, der den Anfang des Counters darstellt den Prozess B mit Fulltest und einen Prozess C. Die Implementierung wird in folgender Abbildung gezeigt.

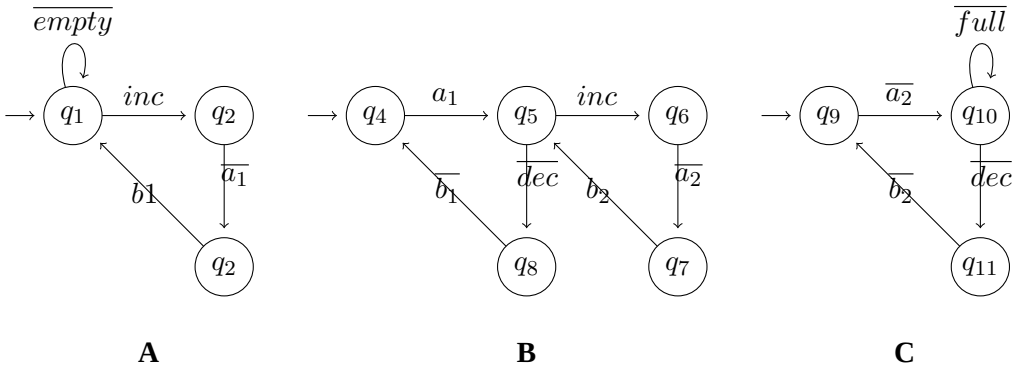


Abbildung: Implementierung Counter

Der Initialzustand des Counters ist wenn sich die Prozesse in den Zuständen q_1, q_4 und q_9 befinden. Hier kann die Aktion $\overline{empty}$ durchgeführt werden oder der Counter mittels inc um eins erhöht werden. Nach der Aktion inc übergibt der Prozess A mittels Linkport a_1 diese Information an den Prozess B weiter, der jetzt seinerseits entweder ein $\overline{dec}$ lesen kann, um diese Information dann mittels Port b_1 an den Prozess A weiter zu leiten oder wieder ein inc lesen kann, um diese Information mittles Port a_2 an den Prozess C weiter zu leiten. Prozess C kann jetzt ein $\overline{full}$ oder ein $\overline{dec}$ lesen.

Literatur

[1] R. Milner, *communicating and mobile systems: the π-calculus*, 1999.