Temporallogische Spezifikationen und CSP
Studienarbeit

Andreas Schäfer

Betreuerin: Dr. Heike Wehrheim

15. Oktober 2000

Fachbereich Informatik
Abteilung Semantik
Prof. Dr. E.-R. Olderog

Carl v. Ossietzky
UNIVERSITÄT
OLDENBURG

Inhaltsverzeichnis

1 Einleitung

CSP und lineare temporale Logik sind zwei unterschiedliche Formalismen zur Spezifikation reaktiver Systeme. Im Gegensatz zu transformationellen Systemen, die zu einer Eingabe eine Ausgabe berechnen und danach terminieren, laufen reaktive Systeme unendlich weiter und ihre Aufgabe besteht darin, mit ihrer „Umwelt“ zu „interagieren“, anstatt ein Ergebniss zu berechnen. Ein Betriebssystem ist also z.B. ein reaktives System.
In CSP kann sowohl das System als auch die gewünschte Eigenschaft beschrieben werden. Dabei orientiert sich die Beschreibung stärker an einer Implementierung, während durch lineare temporale Logik viele gewünschte Eigenschaften sehr intuitiv und abstrakt beschrieben werden können. In dieser Studienarbeit wird ein Verfahren entwickelt, das zu einer Reihe von Spezifikation in temporaler Logik eine äquivalente CSP-Spezifikation konstruieren kann. Für die CSP-Spezifikation kann dann durch ein Computerprogramm für ein CSP-System überprüft werden, ob das System die spezifizierte Eigenschaft besitzt.

2 CSP

Die Sprache CSP (*Communicating Sequential Processes*) wurde von C.A.R. Hoare 1985 in [1] eingeführt. Sie dient der Beschreibung von Systemen, die aus mehreren parallel ablaufenden Prozessen bestehen können. Dabei wird von den internen Abläufen der einzelnen Prozesse abstrahiert und nur die Kommunikation von Prozessen untereinander und mit der Umgebung betrachtet. Einzelne Kommunikationen (auch als Events bezeichnet) können zu Kanälen zusammengefasst werden.
Das Alphabet Σ eines Prozesses ist die Menge der möglichen Kommunikationen. Die folgenden CSP-Konstrukte werden benötigt und sollen kurz informell vorgestellt werden. Großbuchstaben bezeichnen dabei Prozessnamen und Kleinbuchstaben mögliche Kommunikationen.
CSP-Prozesse werden durch Gleichungen definiert, die auch rekursiv sein können. In [1] und [2] sind Syntax und Semantik detailliert beschrieben, hier werden nur die wichtigsten Operatoren an Hand einiger Beispiele vorgestellt.

STOP Der Prozess **STOP** kann keine Kommunikation mehr ausführen und ist nicht terminiert.

Präfix Der Prozess Q mit $Q = a \rightarrow P$ kann zuerst ein a kommunizieren und sich dann wie P verhalten.

Externe Auswahl Für den Prozess $Q = (a \rightarrow P_1)\Box(b \rightarrow P_2)$ kann sich die Umgebung oder ein parallel laufender Prozess aussuchen, ob Q ein a

kommunizieren und sich wie P_1 verhalten soll oder ein b kommunizieren und sich wie P_2 verhalten soll.

Interne Auswahl Bei $Q = (a \rightarrow P_1) \sqcap (b \rightarrow P_2)$ bestimmt der Prozess Q selbst nichtdeterministisch, ob er der Umgebung anbietet a oder b auszuführen und sich dann wie P_1 bzw. wie P_2 zu verhalten.

Die Operatoren für externe bzw. interne Auswahl können auch in verallgemeinerter Form benutzt werden. Dabei steht dann eine (u.U. auch unendliche) Menge von Wahlmöglichkeiten zur Verfügung. In dieser Arbeit wird aber stets von Prozessen ausgegangen, bei denen nur *endlich* viele Wahlmöglichkeiten bestehen.

Rekursion Rekursive Definitionen sind ebenfalls möglich. $CLK = \mathbf{tick} \rightarrow CLK$ ist ein Prozess, der immer **tick** kommuniziert.

Parallele Komposition Um zu modellieren, dass zwei Prozesse parallel ablaufen und dabei über einem Alphabet von Kommunikationen synchronisiert sind, wird der $||$-Operator verwendet.
P mit $P = (P_1 \;{}_{\{a,b\}}||_{\{a,b,c\}}\; P_2)$ ist ein Prozess, bei dem P_1 und P_2 parallel ablaufen. Die beiden Prozesse sind dabei über der Schnittmenge der beiden Alphabete rechts und links des Operators synchronisiert. Sie können also nur beide gleichzeitig die Kommunikation a oder b ausführen. Die Kommunikationen, die nur in einem Alphabet auftauchen, können unabhängig vom anderen Prozess durchgeführt werden. P_2 kann also c unabhängig von P_1 ausführen.

Hiding Eine Menge von Kommunikationen kann mit dem Hiding-Operator „\“ vor der Umgebung versteckt werden. In der CSP-Semantik wird dann jede dieser Kommunikationen durch die spezielle Aktion τ ersetzt. Eine solche τ-Aktion kann unabhängig von der Umgebung durchgeführt werden und parallele Prozesse können darauf auch nicht synchronisiert werden. Ein Prozess entscheidet nichtdeterministisch, ob τ ausgeführt werden soll, oder nicht. In $Q = ((a \rightarrow \mathbf{STOP}) \square (b \rightarrow \mathbf{STOP})) \setminus b$ wird b durch τ ersetzt. Q kann sofort mit τ zu **STOP** übergehen oder warten und der Umgebung die Kommunikation von a anbieten. Der Prozess verhält sich äquivalent zu $Q' = (a \rightarrow \mathbf{STOP}) \sqcap \mathbf{STOP}$.

2.1 Semantik

Die Semantik eines CSP-Prozesses kann durch *traces*, *failures* und *divergences* beschrieben werden.

traces Ein Trace eines CSP-Prozesses ist eine Folge von Kommunikationen, die er nacheinander ausführen kann. Die Menge $traces$ ist die Menge aller dieser Folgen. Jeder Prozess hat den speziellen leeren Trace $\langle\rangle$.
Z.B. hat der Prozess

$$Q = (a \to Q)\Box(b \to \mathbf{STOP})$$

die Traces

$$traces(Q) = \{\langle\rangle, b, a, ab, aa, aab, aaa, aaab, \ldots\}.$$

$$Q' = (a \to Q') \sqcap (b \to \mathbf{STOP})$$

hat dieselben Traces wie Q, weil Q' sich Q' in jedem Schritt entscheiden kann, ob es der Umgebung a anbietet oder b.

failures Ein Failure eines CSP-Prozesses P ist ein Paar (s, X) und bedeutet, dass der Prozess nachdem er den Trace s ausgeführt hat, genau die Kommunikationen aus X *verweigern* und keine τ-Transitionen ausführen kann (*stable failure*). Die Menge aller Paare ist $failures(P)$.
Für Q und Q' sind die *failures* unter der Voraussetzung eines Alphabetes $\Sigma = \{a, b\}$:

$$failures(Q) = \{(\langle\rangle, \emptyset), (a, \emptyset), (b, \{a, b\}), \ldots\}$$

$$failures(Q') = \{(\langle\rangle, \{a\}), (\langle\rangle, \{b\}), (a, \{a\}), (a, \{b\}), (b, \{a, b\}), \ldots\}.$$

Q kann nach jedem a immer noch a oder b kommunizieren. Nach b verhalten sich Q und Q' wie **STOP** , können also alles verweigern. Q' kann sich aber nach jedem a entscheiden, ob es ein a oder ein b erwartet. Wenn b erwartet wird, wird a verweigert und umgekehrt. Falls ein Prozess keine Kommunikation mehr zuläßt, ist er in einen *Deadlock* geraten.
STOP ist der Prototyp eines Deadlock-Prozesses.

divergences Ein Prozess divergiert, falls er nach Abarbeitung eines Traces unendlich viele τ Transitionen durchführen kann. Der Prozess läuft dann intern weiter ohne mit der Umgebung zu kommunizieren. Die Menge der Traces, nach denen ein CSP-Prozess divergieren kann, wird mit $divergences$ bezeichnet.

Berechnung der *traces,failures,divergences* Mengen Diese Mengen werden induktiv über den Aufbau der Prozessterme bestimmt. Von den in [1] und [2] vorgestellten Regeln werden hier nur die folgenden benötigt. In den Regeln wird der zweistellige $\restriction$-Operator verwendet, der aus einem Wort alle Zeichen entfernt, die nicht in der angebenen Menge enthalten sind.

Regel 2.1 **($traces(P\ {}_X\|_Y\ Q)$).**

$$traces(P\ {}_X\|_Y\ Q) = \{s \in (X \cup Y)^* | s \restriction X \in traces(P) \wedge s \restriction Y \in traces(Q)\}$$

Der Prozess $P\ {}_X\|_Y\ Q$ hat alle die Wörter s als Traces, die Trace von P sind, wenn nur Zeichen aus X betrachetet werden und Trace von Q sind, wenn nur Zeichen aus Y betrachtet werden.

Regel 2.2 **($traces(P \setminus X)$).**

$$traces(P \setminus X) = \{s \restriction X | s \in traces(P)\}$$

Die Traces von $traces(P \setminus X)$ sind die Traces von P beschränkt auf die Kommunikationen aus X.

Regel 2.3 **($failures(P\ {}_X\|_Y\ Q)$).**

$$\begin{aligned} failures(P\ {}_X\|_Y\ Q) \quad = \quad & \{(s, (\underbrace{(A \cap B)}_{\text{von P und Q verweigert}} \cup \underbrace{((X \cap Y) \cap (A \cup B))}_{\substack{\text{von P oder Q verweigert,}\\ \text{aber darüber synchronisiert}}} \mid \\ & s \in (X \cup Y)^* \wedge \\ & (s \restriction X, A) \in failures(P) \wedge \\ & (s \restriction Y, B) \in failures(Q))\} \end{aligned}$$

Bei paralleler Komposition werden also die Events verweigert, die von beiden Prozessen verweigert werden. Für die Events über denen die Prozesse synchronisiert sind, reicht es schon, wenn ein Prozess den Event verweigert.

Regel 2.4 **($failures(P \setminus X)$).**

$$failures(P \setminus X) = \{(s \restriction (\Sigma \setminus X), A) | (s, A \cup X) \in failures(P)\}$$

Wenn ein Prozess P alle Kommunikationen aus $X \cup A$ verweigert, dann verweigert $P \setminus X$ alle Kommunikationen, die zusätzlich zu denen aus X – also die aus A– von P verweigert werden in einem stabilen Zustand.

Regel 2.5 **($divergences(P\ {}_X\|_Y\ Q)$).**

$$\begin{aligned} divergences(P\ {}_X\|_Y\ Q) \quad = \quad & \{s\hat{\ }t | t \in (X \cup Y)^* \wedge \\ & s \restriction X \in traces(P) \wedge \\ & s \restriction Y \in traces(Q) \wedge \\ & (s \restriction X \in divergences(P) \vee \\ & s \restriction Y \in divergences(Q))\} \end{aligned}$$

Der Prozess $P\ {}_X\|_Y\ Q$ divergiert also, falls einer der beiden Prozesse P oder Q divergiert.

Regel 2.6 ($divergences(P \setminus X)$).

$$\begin{aligned} divergences(P \setminus X) &= \{(s \restriction (\Sigma \setminus X))\hat{\ }t | t \in (\Sigma \setminus X)^* \wedge \\ &\quad (s \in divergences(P) \vee \\ &\quad (\forall n \in \mathbb{N}\ \exists u \in X^* : \#u > n \wedge (s\hat{\ }u) \in traces(P)))\} \end{aligned}$$

Der Prozess $P \setminus X$ divergiert also falls P divergiert oder falls eine unendliche Folge von Kommunikationen aus X durchgeführt werden kann.

2.2 Refinement(Verfeinerung)

Verfeinerung (Refinement) ist ein zentrales Konzept von CSP. Ein Prozess Q verfeinert einen Prozess P falls sich P nur so „verhalten" kann wie Q. Diese Beziehung wird durch $P \sqsubseteq Q$ beschrieben. Damit ist es möglich, die wesentlichen Eigenschaften eines Systems durch eine einfache CSP-Spezifikation zu beschreiben. Für eine spätere Implementierung kann dann über die Verfeinerungsrelation geprüft werden, ob die Implementierung die Vorgaben der Spezifikation erfüllt. Der Test kann automatisch durch das Tool FDR durchgeführt werden.
CSP bietet drei verschieden starke Verfeinerungsrelationen, die auf den $traces$–, $failures$– und $divergences$-Mengen basieren.

Trace-Refinement Ein Prozess Q ist ein Trace-Refinement von P, falls alle Traces von Q auch für P möglich sind.

$$P \sqsubseteq_T Q \text{ gdw. } traces(Q) \subseteq traces(P)$$

Dieser schwächste Verfeinerungsbegriff erlaubt zum Beispiel die Spezifikation von Sicherheitseigenschaften, wie das Verbieten bestimmter Events oder Eventfolgen. Dadurch wird aber nicht ausgesagt, dass der Prozess überhaupt etwas tut; so ist **STOP** Trace-Refinement jeden Prozesses, da der einzige Trace von **STOP** der leere Trace $\langle\rangle$ ist, der von jedem anderen Prozess ausgeführt werden kann.

Beispiel 2.1 (Trace-Refinement). Die Spezifikation soll beschreiben, dass nur die Kommunikationen a und b möglich sind.

$$Spec = (a \rightarrow Spec) \,\Box\, (b \rightarrow Spec)$$

erzeugt alle Traces, die nur as und bs enthalten. Für die Implementierungen

$$P_1 = a \rightarrow P_1$$

und

$$P_2 = (a \rightarrow P_2) \,\Box\, (c \rightarrow \mathbf{STOP})$$

gilt dann

$$Spec \sqsubseteq P_1 \quad \text{aber} \quad Spec \not\sqsubseteq P_2,$$

da P_2 auch den Trace c hat, den die Spezifikation nicht besitzt.

Failures-Refinement Bei Failures-Refinement ($\sqsubseteq_F$) wird zusätzlich zum Trace-Refinement berücksichtigt, welche Kommunikationen die Prozesse verweigern können, d.h.

$$P \sqsubseteq_F Q \text{ gdw. } traces(Q) \subseteq traces(P) \text{ und } failures(Q) \subseteq failures(P)$$

Beispiel 2.2 (Failures-Refinement). Eine Spezifikation, die beschreibt, dass immer die Kommunikation von a möglich ist, kann so aussehen:

$$Spec = (b \to Spec \sqcap c \to Spec) \,\Box\, a \to Spec$$

Spec kann nur b und c verweigern und hat alle Folgen aus as bs und cs als Traces.

Failures-Divergences-Refinement Die schärfste Verfeinerungsrelation ($\sqsubseteq_{\mathrm{FD}}$) verlangt auch, dass Q nur dann divergiert, wenn die Spezifikation P auch divergiert.

$$\begin{array}{lll} P \sqsubseteq_{\mathrm{FD}} Q & \text{gdw.} & traces(Q) \subseteq traces(P) \text{ und} \\ & & failures(Q) \subseteq failures(P) \text{ und} \\ & & divergences(Q) \subseteq divergences(P) \end{array}$$

2.3 FDR

Mit dem Tool FDR (Failures-Divergences-Refinement) von *Formal Systems Europe* können die drei Verfeinerungsrelationen automatisch überprüft werden. Dazu müssen die CSP-Prozesse in einer „maschinenlesbaren“ Notation eingegeben werden, die in [5] beschrieben ist. Die Notation bildet die CSP-Operatoren durch ASCII-Zeichen nach.

CSP-Notation	FDR-CSP-Notation
$P \,\Box\, Q$	`P [] Q`
$P \sqcap Q$	`P |~| Q`
$a \to P$	`a -> P`
$P \;{}_{\{a,b,c\}}\Vert_{\{d,e,f\}}\; Q$	`P [{a,b,c}||{d,e,f}] Q`

Außerdem wird die Deklaration von möglichen Kommunikationen und Kanälen durch das Schlüsselwort `channel` verlangt. Mit der Anweisung `assert P [FD= Q` wird FDR mitgeteilt, dass die Beziehung $P \sqsubseteq_{FD} Q$ überprüft werden soll.

Beispiel 2.3 (Geldautomat). In Anlehnung an die Beispiele aus [2] kann ein Geldautomat in CSP wie folgt modelliert werden.
Events des Automaten:

- **cardIn** Karte einführen

- **pinIn** PIN eingeben
- **reqIn** Geldbetrag eingeben
- **chkPin** PIN überprüfen
- **pinOK** PIN in Ordnung
- **pinNOK** PIN nicht in Ordnung
- **cardOut** Karte ausgeben
- **dispMoney** Geld auszahlen

Beschreibung in CSP:

$$\begin{aligned} ATM &= \mathbf{cardIn} \rightarrow \mathbf{pinIn} \rightarrow \mathbf{reqIn} \rightarrow \mathbf{chkPin} \rightarrow \\ &\quad ((\mathbf{pinOK} \rightarrow \mathbf{cardOut} \rightarrow \mathbf{dispMoney} \rightarrow ATM) \\ &\quad \Box(\mathbf{pinNOK} \rightarrow ATM)) \end{aligned}$$

Der Benutzer gibt die Karte in den Automaten ein und tippt die PIN und den gewünschten Geldbetrag. Danach überprüft der Automat die PIN und gibt die Karte und das Geld aus, falls die PIN in Ordnung ist. Die PIN soll von einem Zentralrechner überprüft werden, der auch in CSP spezifiziert ist:

$$ZR = \mathbf{chkPin} \rightarrow (\mathbf{pinOK} \rightarrow ZR \sqcap \mathbf{pinNOK} \rightarrow ZR)$$

Ob eine PIN in Ordnung ist, entscheidet der Rechner in diesem einfachen Modell durch Nichtdeterminismus. Die Zusammenarbeit des Geldautomaten mit dem Zentralrechner kann dann durch parallel Komposition dargestellt werden:

$$ATMZR = ATM\ {}_{\{\mathbf{chkPin},\mathbf{pinOK},\mathbf{pinNOK}\}}\|_{\{\mathbf{chkPin},\mathbf{pinOK},\mathbf{pinNOK}\}}\ ZR$$

Für den Benutzer erfolgt die Überprüfung der PIN unsichtbar, er arbeitet mit :

$$ATMUSER = ATMZR \setminus \{\mathbf{chkPin}, \mathbf{pinOK}, \mathbf{pinNOK}\}.$$

Ein Geldautomat soll im allgemeinen eine Reihe von Eigenschaften besitzen. Eine sinnvolle Sicherheitseigenschaft ist, dass ein Automat die Karte behalten soll, falls die PIN falsch ist. Andererseits soll er die Lebendigkeitseigenschaft erfüllen, Geld auszugeben, falls die PIN in Ordnung ist. Diese Eigenschaften können sehr einfach durch temporallogische Formeln beschrieben werden.

3 Lineare Temporale Logik

Temporale Logik erweitert die Aussagenlogik durch eine Reihe von Operatoren, mit denen der zeitliche Ablauf beschrieben werden kann. Sie wurde von A. Pnueli eingeführt, um das Verhalten von reaktiven Systemen zu beschreiben. In [3] wird die Propositionale Temporale Logik PTL mit Zukunfts- und Vergangenheitsoperatoren eingeführt und ein Beweissystem vorgestellt, mit dem nachgewiesen werden kann, dass ein Programm das temporallogisch beschriebene Verhalten besitzt. Hier wird nur PTL mit Zukunftsoperatoren betrachtet.

3.1 Informelle Beschreibung

Atomare Formeln sind neben **true** und **false** auch *Propositionen*, Eigenschaften von denen zu jedem Zeitpunkt feststeht, ob das Programm die Eigenschaft besitzt oder nicht. Für Programme kann $x = 4$ eine Proposition sein, die beschreibt, dass eine Variable x den Wert 4 besitzt. Ein einzelner Buchstabe a kann für einen CSP-Prozess eine Proposition sein, um zu beschreiben, dass a die letzte durchgeführte Kommunikation ist.

Boolsche Operatoren können wie in der Aussagenlogik beliebige Formel verbinden.

Temporallogische Operatoren Die temporale Logik erweitert die Aussagenlogik um die drei einstelligen Operatoren *eventually* ($\Diamond$), *always* ($\Box$) und *next* ($\bigcirc$) und um den zweistelligen *until*-Operator ($\mathcal{U}$).

- $\Diamond$ Der *eventually*-Operator beschreibt, dass irgendwann in der Zukunft die Formel dahinter gilt. Die Forderung, dass der Geldautomat *irgendwann* die Karte wieder herausgibt kann z.B. durch

  $$\Diamond\mathbf{cardOut}$$

  ausgedrückt werden.

- $\Box$ Der zu *eventually* duale *always*-Operator ist erfüllt, falls die Formel dahinter in Zukunft *immer* erfüllt ist.

  $$\Box\,(\mathbf{pinIn} \rightarrow \Diamond\mathbf{chkPin})$$

  formalisiert, dass der Geldautomat immer wenn eine PIN-Nummer eingegeben wird, die Nummer irgendwann danach auch überprüft wird.

- $\bigcirc$ Der *next*-Operator fordert, dass die Formel dahinter im nächsten Zeitpunkt erfüllt ist. Damit kann die Eigenschaft „Zuerst wird die Karte eingesteckt und danach die PIN eingegeben“ durch

  $$\mathbf{cardIn} \wedge \bigcirc\mathbf{pinIn}$$

beschrieben werden.

$\mathcal{U}$ Eine Formel $\phi_1 \mathcal{U} \phi_2$ mit dem *until*-Operator ist erfüllt, falls irgendwann ϕ_2 gilt und vorher immer ϕ_1. Für den Geldautomaten kann man damit durch

$$(\neg\textbf{dispMoney}\ \mathcal{U}\ \textbf{pinOK}) \vee (\Box\neg\textbf{dispMoney})$$

beschrieben werden, dass kein Geld ausgezahlt wird, bis feststeht, dass die PIN in Ordnung ist.

Durch Formeln der propositionalen temporalen Logik werden Sprachen über dem Alphabet der möglichen Wahrheitswerte der Propositionen beschrieben. Diese Sprachen sind regulär und können somit von einem endlichen Automaten (oder Büchi-Automaten) erkannt werden. In [4] wird eine Konstruktion eines erkennenden Automaten zu einer PTL-Formel vorgestellt.

3.2 Beispiel

Zu dem in Abschnitt 2.3 in CSP beschriebenen Geldautomaten soll eine Sicherheitseigenschaft und eine Lebendigkeitseigenschaft in PTL spezifiziert werden.

Beispiel 3.1 (Sicherheitseigenschaft). Wenn der Benutzer die EC-Karte einsteckt, soll ein Geldautomat kein Geld ausgeben, bis feststeht, dass die PIN in Ordnung ist oder ein neuer Benutzer eine Karte einführt. Diese Tatsache kann durch die Formel

$$\Box(\textbf{cardIn} \rightarrow ((\neg\textbf{dispMoney} \wedge \neg\textbf{cardIn})\ \mathcal{U}\ (\textbf{pinOK} \vee \textbf{cardIn})))$$

beschrieben werden.

Beispiel 3.2 (Lebendigkeitseigenschaft). Durch die Formel

$$\Box(\Diamond\textbf{cardIn})$$

wird die einfache Lebendigkeitseigenschaft „Der Automat ist immer irgendwann wieder bereit einen Kunden zu bedienen“ beschrieben.

4 PTL Spezifikationen für CSP-Prozesse

PTL-Formeln können auch dazu benutzt werden CSP-Prozesse zu spezifizieren. Dabei können viele Eigenschaften durch PTL-Formeln sehr viel intuitiver ausgedrückt werden, als durch einen anderen CSP-Prozess. In dieser Studienarbeit werden PTL-Formeln benutzt, um das Auftreten von Ereignissen zu beschreiben, atomare Propositionen sind daher die auftretenden Kommunikationen eines CSP-Prozesses. Interpretiert werden die Formeln über endlichen und unendlichen Traces, d.h. Wörten aus Σ^* und Σ^ω.

Definition 4.1 (Erfüllbarkeit für unendliche Traces). Sei ϕ eine PTL Formel, $s = s(0)s(1)\ldots \in \Sigma^\omega$ ein *unendlicher Trace*, dann gilt:

Konstanten

$$s \models \textbf{true} \text{ immer}$$

$$s \models \textbf{false} \text{ nie}$$

Propositionen Sei $a \in \Sigma$, dann gilt:

$$s \models a \quad \text{gdw.} \quad \exists t \in \Sigma^\omega : s = at$$

Operatoren Seien ϕ, ψ PTL-Formeln, dann gilt:

$\neg$	$s \models \neg\phi$	gdw.	$s \not\models \phi$
$\wedge$	$s \models \phi \wedge \psi$	gdw.	$s \models \phi$ und $s \models \psi$
$\vee$	$s \models \phi \vee \psi$	gdw.	$s \models \phi$ oder $s \models \psi$
$\rightarrow$	$s \models \phi \rightarrow \psi$	gdw.	falls $s \models \phi$ dann $s \models \psi$
$\bigcirc$	$s \models \bigcirc\phi$	gdw.	$s(1)s(2)\ldots \models \phi$
$\Diamond$	$s \models \Diamond\phi$	gdw.	$\exists\ k \in \mathbb{N}\ s(k)s(k+1)\ldots \models \phi$
$\Box$	$s \models \Box\phi$	gdw.	$\forall\ k \in \mathbb{N}\ s(k)s(k+1)\ldots \models \phi$
$\mathcal{U}$	$s \models \phi\ \mathcal{U}\ \psi$	gdw.	$\exists\ k \in \mathbb{N}\ [s(k)s(k+1)\ldots \models \psi$ und $\forall\ 0 \leq j < i : s(j)s(j+1)\ldots \models \phi$

Definition 4.2 (Erfüllbarkeit für endliche Traces). Sei ϕ eine PTL Formel, $s = s(0)s(1)\ldots s(n) \in \Sigma^*$ ein *endlicher Trace*, dann gilt:

Konstanten

$$s \models \textbf{true} \text{ immer}$$

$$s \models \textbf{false} \text{ nie}$$

Propositionen Sei $a \in \Sigma$, dann gilt:

$$s \models a \quad \text{gdw.} \quad \exists t \in \Sigma^* : s = at$$

Operatoren Seien ϕ, ψ PTL-Formeln, dann gilt:

$\neg$	$s \models \neg\phi$	gdw.	$s \not\models \phi$
$\wedge$	$s \models \phi \wedge \psi$	gdw.	$s \models \phi$ und $s \models \psi$
$\vee$	$s \models \phi \vee \psi$	gdw.	$s \models \phi$ oder $s \models \psi$
$\rightarrow$	$s \models \phi \rightarrow \psi$	gdw.	falls $s \models \phi$ dann $s \models \psi$
$\bigcirc$	$s \models \bigcirc\phi$	gdw.	$s(1)s(2)\ldots s(n) \models \phi$ und $n \geq 1$
$\Diamond$	$s \models \Diamond\phi$	gdw.	$\exists\ k \in \mathbb{N}, k \leq n : s(k)s(k+1)\ldots s(n) \models \phi$
$\Box$	$s \models \Box\phi$	gdw.	$\forall\ k \in \mathbb{N}\ k \leq n \Rightarrow s(k)s(k+1)\ldots s(n) \models \phi$
$\mathcal{U}$	$s \models \phi\ \mathcal{U}\ \psi$	gdw.	$\exists\ k \in \mathbb{N}, k \leq n : [s(k)s(k+1)\ldots s(n) \models \psi$ und $\forall\ 0 \leq j < i : s(j)s(j+1)\ldots s(n) \models \phi$

Definition 4.3 (Maximale Traces). Ein endlicher Trace s heißt maximaler Trace, falls nach Ausführung des Traces jede Kommunikation verweigert werden kann.

Definition 4.4 (Erfüllbarkeitsrelation für CSP-Prozesse). Sei P ein nicht divergierender CSP-Prozess und ϕ eine PTL-Formel, dann gilt:

$$P \vDash \phi \quad \text{gdw.} \quad \text{Für alle unendlichen und alle maximalen Traces } \; s \text{ gilt } s \vDash \phi$$

Die Einschränkung auf nicht divergierende Prozesse ist notwendig, um im nächsten Schritt die Gültigkeit der Formel automatisch überprüfen zu können.

5 PTL Modelchecking mit FDR

Für CSP-Prozesse und die Klasse von PTL-Formeln, die durch einen deterministischen Büchi-Automaten erkannt werden, kann mit FDR über *failures-divergences-refinement* getestet werden, ob ein Prozess eine Formel dieser Klasse nach Definition 4.4 erfüllt. Dazu wird zuerst ein deterministischer Automat konstruiert, der alle unendlichen und maximalen Traces akzeptiert, die die Formel erfüllen. Dieser Automat wird dann durch eine Menge von CSP-Prozessen simuliert, die dann parallel zu dem zu testenden CSP-Prozess ablaufen. Wenn sich der Automat in einem Endzustand befindet, zeigen das die entsprechenden CSP-Prozesse durch ein spezielles Event der Umgebung an.

Konstruktion PTL-Formel $\rightsquigarrow$ Automat Sei ϕ eine PTL-Formel. Ähnlich zu [4] kann ein erkennender –nicht unbedingt deterministischer– Automat konstruiert werden. Für eine Reihe von Formeln existiert aber auch ein erkennender *deterministischer* Automat

$$\mathcal{A}_\phi = (Q_\mathcal{A}, \Sigma, \Delta, q_0, \underbrace{B}_{\text{Büchi-Zustände}}, \underbrace{F}_{\text{Endzustände}})$$

für den gilt:

$$\begin{aligned} s = s(0)s(1)\ldots s(n) \vDash \phi \quad & \text{gdw.} \quad \text{es einen endl. Lauf von } \mathcal{A}_\phi \text{ auf } s \\ & \phantom{\text{gdw.}} \quad \text{gibt, der in einem Endzustand endet.} \\ s = s(0)s(1)\ldots \vDash \phi \quad & \text{gdw.} \quad \text{es einen unendl. Lauf von } \mathcal{A}_\phi \text{ auf } s \\ & \phantom{\text{gdw.}} \quad \text{gibt, der mindestens einen Büchi-Zustand} \\ & \phantom{\text{gdw.}} \quad \text{unendlich oft durchläuft.} \end{aligned}$$

Bemerkung 5.1. Ein deterministischer Automat soll für jedes Eingabezeichen *genau* einen Übergang besitzen.

Konstruktion Automat $\rightsquigarrow$ CSP-Prozesse

Idee Ein so konstruierter Automat kann in CSP durch eine Menge von Prozessen simuliert werden. Zu der Menge $Q_{\mathcal{A}} = \{q_0, \ldots, q_n\}$ der Zustände des Automaten $\mathcal{A}$ wird eine Menge von CSP-Prozessen $Q_0, \ldots, Q_n$ konstruiert, die die Transitionen von $\mathcal{A}$ simulieren. Wenn sich der Automat $\mathcal{A}$ in einem Büchi-Zustand befindet soll dies der entsprechende CSP-Prozess der Umgebung dadurch mitteilen, dass er eine *neue* Kommunikation **acc** (**acc** $\notin \Sigma$) mit der Umgebung durchführt, bevor der Zustand wieder verlassen wird. Befindet sich der Automat in einem Endzustand soll der CSP-Prozess **acc** kommunizieren können ohne den Zustand zu wechseln oder vorher eine andere Kommunikation durchführen zu müssen. Die CSP-Prozesse sollen ein unendliches oder endliches „Wort" dadurch akzeptieren, dass sie unendlich oft **acc** kommunizieren können.

Konstruktion

$$
\begin{array}{lll}
Q_i = \Box\{a \to Q_j | q_i \xrightarrow{a} q_j \in \Delta\} & \text{falls} & q_i \notin B \cup F \\
Q_i = \Box\{a \to \mathbf{acc} \to Q_j | q_i \xrightarrow{a} q_j \in \Delta\} & \text{falls} & q_i \in B \setminus F \\
Q_i = \Box\{a \to Q_j | q_i \xrightarrow{a} q_j \in \Delta\} \Box \mathbf{acc} \to Q_i & \text{falls} & q_i \in F \setminus B \\
Q_i = \Box\{a \to \mathbf{acc} \to Q_j | q_i \xrightarrow{a} q_j \in \Delta\} \Box \mathbf{acc} \to Q_i & \text{falls} & q_i \in F \cap B
\end{array}
$$

Seien ausserdem

$$Q^F_{\mathbf{acc}} = \{Q_i \in \{Q_0, \ldots, Q_n\} | q_i \in F\},$$

$$Q^B_{\mathbf{acc}} = \{Q_i \in \{Q_0, \ldots, Q_n\} | q_i \in B\},$$

$$Q_{\mathbf{acc}} = Q^F_{\mathbf{acc}} \cup Q^B_{\mathbf{acc}}$$

die Mengen der Endzustandsprozesse.

Bemerkung 5.2. Da der Prozess Q_0 parallel mit dem zu testenden Prozess P laufen soll, muss in den Büchi- und Endzustandsprozessen zuerst die normale Kommunikation a erfolgen, bevor **acc** kommuniziert werden kann. Wäre umgekehrt Q_i definiert durch

$$Q_i = \Box\{\mathbf{acc} \to a \to Q_j | q_i \xrightarrow{a} q_j \in \Delta \text{ und } q_i \in B \setminus F\}$$

könnte Q_i nach Kommunikation von **acc** nicht mehr mit dem parallel laufenden Prozess synchronisiert werden, da dann nur noch *eine* Kommunikation möglich wäre.

Modelchecking Um mit FDR testen zu können, ob ein Prozess unendlich oft **acc** kommunizieren kann, wird der Prozess INFACC als Spezifikation definiert:

$$\text{INFACC} = \mathbf{acc} \to \text{INFACC}$$

Mit diesen Definitionen kann dann für einen CSP-Prozess P mit Alphabet Σ wie folgt geprüft werden, ob P die PTL-Spezifikation ϕ erfüllt. Dazu wird P_{Test} definiert durch

$$P_{\text{Test}} = (P \mathbin{{}_{\Sigma}\|_{\Sigma \cup \{\mathbf{acc}\}}} Q_0) \setminus \Sigma.$$

Es gilt dann der Satz:

Satz 5.1. Sei P ein nicht divergierender CSP-Prozess mit endlichem Verzweigungsgrad und ϕ eine PTL-Formel, zu der ein *deterministischer* Automat $\mathcal{A}_\phi = (Q_\mathcal{A}, \Sigma, \Delta, q_0, B, F)$ konstruiert werden kann. Dann kann P_{Test} wie angegeben konstruiert werden, so dass:

$$\text{INFACC} \sqsubseteq_{\text{FD}} P_{\text{Test}} \quad \text{gdw.} \quad P \vDash \phi.$$

Beweis Für den Beweis werden folgende Lemmata benötigt:

Lemma 5.2. Aus der Konstruktion der CSP-Prozesse $Q_0, \ldots, Q_n$ folgt sofort für $q, q' \in Q_\mathcal{A}$ und $Q, Q' \in \{Q_0, \ldots, Q_n\}$:

1. $q \xrightarrow{a} q' \in \Delta$ gdw. $Q \xRightarrow{a} Q'$ für $q \notin B, Q \notin Q^B_{\mathbf{acc}}$
2. $q \xrightarrow{a} q' \in \Delta$ gdw. $Q \xRightarrow{a\mathbf{acc}} Q'$ für $q \in B, Q \in Q^B_{\mathbf{acc}}$

Lemma 5.3. Sei s ein *unendlicher* Trace von P und $s \nvDash \phi$, dann kann P_{Test} divergieren.

Beweis Gelte $s \nvDash \phi$. Da $\mathcal{A}$ deterministisch ist, existiert ein unendlicher nicht-akzeptierender Lauf $\rho = \rho(0), \rho(1), \ldots$ von $\mathcal{A}$ mit

$$\forall i \in \mathbb{N} : \rho(i) \in Q_\mathcal{A}$$

und

$$\forall i \in \mathbb{N} : \rho(i) \xrightarrow{s(i)} \rho(i+1)$$

und

$$\exists f \in \mathbb{N} : \forall i > f : \rho(i) \notin B,$$

d.h. nach Abarbeitung des endlichen Präfix $s(0) \ldots s(f)$ wird kein Büchi-Zustand mehr durchlaufen. Nach Lemma 5.2 existiert dann eine Ausführung ρ' des CSP-Prozesses Q_0 mit $\rho' = \rho'(0), \rho'(1), \ldots$,

$$\rho'(0) = Q_0$$

und

$$\forall i \in \mathbb{N} : \rho'(i) \in \{Q_0, \dots, Q_n\}$$

und

$$\forall i \in \mathbb{N} : \rho'(i) \overset{s(i)}{\Rightarrow} \rho'(i+1) \text{ oder } \rho'(i) \overset{s(i)\mathbf{acc}}{\Rightarrow} \rho'(i+1)$$

und

$$\exists f \in \mathbb{N} : \forall i > f : \rho'(i) \notin Q^B_{\mathbf{acc}}$$

daraus folgt für das feste f nach Lemma 5.2 1. :

$$\forall i > f : \rho'(i) \overset{s(i)}{\Rightarrow} \rho'(i+1),$$

Somit existiert ein Trace s' von Q_0 mit den Eigenschaften:

- $s' \setminus \{acc\} = s$
- $\exists f \in \mathbb{N} : \forall i > f : s'(i) \in \Sigma.$

Da s ein Trace von P und s' ein Trace von Q_0 ist, ist s' nach CSP-Semantik ein Trace von $(P\ {}_{\Sigma}||_{\Sigma\cup\{\mathbf{acc}\}}\ Q_0)$ (Regel 2.1). Dass heißt, **acc** muss nach Abarbeitung von $s'(0)\dots s'(f)$ nicht mehr kommuniziert werden und

$$P_{\text{Test}} = (P\ {}_{\Sigma}||_{\Sigma\cup\{\mathbf{acc}\}}\ Q_0) \setminus \Sigma$$

divergiert nach Definition des Hidings (Regel 2.6).

Lemma 5.4. Sei s ein maximaler Trace von P mit $s \nvDash \phi$, dann kann P_{Test} in einen Deadlock geraten.

Beweis Gelte $s \nvDash \phi$. Da $\mathcal{A}$ deterministisch ist, existiert ein endlicher nicht-akzeptierender Lauf $\rho = \rho(0), \rho(1), \dots \rho(l)$ von $\mathcal{A}$ mit

$$\forall 0 \le i \le l : \rho(i) \in Q_{\mathcal{A}}$$

und

$$\forall 0 \le i \le l : \rho(i) \xrightarrow{s(i)} \rho(i+1)$$

und

$$\rho(l) \notin F,$$

d.h. nach Abarbeitung von s befindet sich $\mathcal{A}$ nicht in einem Endzustand.
Nach Lemma 5.2 existiert dann ein nicht-akzeptierender Ablauf $\rho' = \rho'(0), \rho'(1), \dots \rho'(l)$ von Q_0 mit

$$\rho' = Q_0,$$

$$\forall 0 \le i \le l : \rho'(i) \in \{Q_0, \dots, Q_n\}$$

und

$$\forall 0 \leq i \leq l : \rho'(i) \stackrel{s(i)}{\Rightarrow} \rho'(i+1) \text{ oder } \rho'(i) \stackrel{s(i)\mathbf{acc}}{\Rightarrow} \rho'(i+1)$$

und

$$\rho'(l) \notin Q^F_{\mathbf{acc}}.$$

Der Prozess $\rho'(l) \in \{Q_0, \ldots, Q_n\}$ erlaubt deswegen nach Definition nur Kommunikationen aus Σ. Somit existiert ein Trace s' von Q_0 mit

- $s' \setminus \{\mathbf{acc}\} = s$.
- $(s', \{\mathbf{acc}\}) \in failures(Q_0)$.

Da P nach Voraussetzung nach Abarbeitung von s in einen Deadlock gerät,gilt

$$(s, \Sigma) \in failures(P)$$

und da beide Prozesse in P_{Test} über Σ synchronisiert sind, folgt nach Regel 2.3

$$(s', \Sigma \cup \{\mathbf{acc}\}) \in failures(P \;{}_{\Sigma}\|_{\Sigma \cup \{\mathbf{acc}\}}\; Q_0)$$

und P_{Test} muss nach Regel 2.4 ebenfalls in einen Deadlock geraten.

Beweis Satz 5.1

„$\Longrightarrow$“ Gelte INFACC $\sqsubseteq_{FD} P_{\text{Test}}$ Annahme: $P \not\models \phi$.

1.Fall Es existiert ein unendlicher Trace s mit $s \not\models \phi$. Dann gilt nach Lemma 5.3, dass P_{Test} divergieren kann. Da INFACC nicht divergieren kann, kann P_{Test} kein failures-divergences-refinement von INFACC sein.
$\lightning$ zur Annahme.

2.Fall Es existiert ein endlicher Trace s, so dass P nach s deadlocked mit $s \not\models \phi$. Dann gilt nach Lemma 5.4, dass P_{Test} in einen Deadlock geraten kann. Da INFACC nicht in einen Deadlock geraten kann, kann P_{Test} kein failures-divergences-refinement von INFACC sein.
$\lightning$ zur Annahme.

„$\Longleftarrow$“ Gelte $P \models \phi$ Annahme : INFACC $\not\sqsubseteq_{FD} P_{\text{Test}}$ Dann sind drei Fälle zu unterscheiden:

1. $traces(P_{\text{Test}}) \not\subseteq traces($ INFACC $)$
   Dieser Fall kann nicht auftreten, da P_{Test} wegen des Hiding nur **acc** kommunzieren kann und INFACC immer **acc** kommunizieren kann.

2. $failures(P_{\text{Test}}) \not\subseteq failures(\text{ INFACC }) = \{(\mathbf{acc}^n, \Sigma \setminus \{\mathbf{acc}\}) | n \in \mathbb{N}\}$, d.h. es gibt ein Paar $(s, X) \in failures(P_{\text{Test}})$ mit $s \in \{\mathbf{acc}\}^*$ und $\mathbf{acc} \in X$, also einen Trace s von P_{Test}, der die Kommunikation von **acc** verweigert und nicht divergiert. Da alle Prozesse aus $\{Q_0, \ldots, Q_n\}$ immer alle Kommunikationen aus Σ ausführen können, können sie nicht in einen Deadlock geraten. Nach Definition von P_{Test} muss also P nach Ausführen eines Trace $s = s(0)s(1)\ldots s(l)$ in einen Deadlock geraten (Regel 2.3 und Regel 2.4). Da auch keine Kommunikation von **acc** möglich ist, gibt es einen Ablauf $\rho' = \rho(0)\rho(1)\ldots$ von Q_0 mit

$$\rho(0) = Q_0$$

und

$$\forall 0 \leq i \leq l : \rho'(i) \in \{Q_0, \ldots, Q_n\}$$

und

$$\forall 0 \leq i \leq l : \rho'(i) \stackrel{s(i)}{\Rightarrow} \rho'(i+1) \text{ oder } \rho'(i) \stackrel{s(i)\mathbf{acc}}{\Rightarrow} \rho'(i+1)$$

und

$$\rho'(l) \notin Q^F_{\mathbf{acc}},$$

da alle Prozesse in $Q^F_{\mathbf{acc}}$ **acc** unabhängig von einer Synchronisation kommunizieren können. Nach Lemma 5.2 ($\Longleftarrow$) existiert dann ein Lauf $\rho = \rho(0), \rho(1), \ldots \rho(l)$ von $\mathcal{A}$ mit

$$\forall 0 \leq i \leq l : \rho(i) \in Q_{\mathcal{A}}$$

und

$$\forall 0 \leq i \leq l : \rho(i) \xrightarrow{s(i)} \rho(i+1)$$

und

$$\rho'(l) \notin Q^F_{\mathbf{acc}}.$$

Also akzeptiert $\mathcal{A}$ das Wort s nicht und es gilt $s \not\models \phi$ ↯ zur Annahme.

3. $divergences(P_{\text{Test}}) \not\subseteq divergences(\text{ INFACC }) = \emptyset$ Es gibt also einen Trace von P_{Test}, so dass P_{Test} danach divergiert. Nach Regel 2.5 und Regel 2.6 sind drei Fälle möglich:

1.Fall Falls P divergiert, folgt aus der Definition direkt $P \not\models \phi$.

2.Fall $(P \; {}_{\Sigma}\|_{\Sigma \cup \{\mathbf{acc}\}} \; Q_0)$ führt nur noch Kommunikationen aus Σ aus. In diesem Fall gibt es einen Trace s von P und einen Trace s' von Q_0, so dass $s = s' \setminus \{\mathbf{acc}\}$ und

$$\exists f \in \mathbb{N} : \forall i > f : s'(i) \in \Sigma$$

gilt. Damit existiert ein Ablauf $\rho' = \rho'(0), \rho'(1), \ldots$ von Q_0 mit

$$\forall i \in \mathbb{N} : \rho'(i) \overset{s(i)}{\Rightarrow} \rho'(i+1) \text{ oder } \rho'(i) \overset{s(i)\mathbf{acc}}{\Rightarrow} \rho'(i+1)$$

und

$$\exists f \in \mathbb{N} : \forall i > f : \rho'(i) \notin Q^B_{\mathbf{acc}},$$

da ein Prozess aus $Q^B_{\mathbf{acc}}$ nach einer Kommunikation aus Σ zuerst ein **acc** kommunizieren muss, bevor weitere Kommunikationen aus Σ möglich sind. Wegen Lemma 5.2 gibt es dann einen Lauf $\rho = \rho(0), \rho(1), \ldots$ von $\mathcal{A}$, mit

$$\forall i \in \mathbb{N} : \rho(i) \in Q_{\mathcal{A}}$$

und

$$\forall i \in \mathbb{N} : \rho(i) \xrightarrow{s(i)} \rho(i+1)$$

und

$$\exists f \in \mathbb{N} : \forall i > f : \rho(i) \notin B,$$

d.h. nach Abarbeitung des endlichen Präfix $s(0) \ldots s(f)$ wird kein Büchi-Zustand mehr durchlaufen. $\mathcal{A}$ akzeptiert also s nicht und es gilt $s \nvDash \phi$

3.Fall Q_0 divergiert. Dieser Fall kann nicht auftreten, da Q_0 divergenzfrei ist.

$\lightning$ zur Annahme.

Also gilt

$$\text{INFACC} \sqsubseteq_{\text{FD}} P_{\text{Test}} \quad \text{gdw.} \quad P \vDash \phi$$

$\square$

6 Beispiele

Das vorgestellte Verfahren zur Konstruktion eines CSP-Prozesses soll jetzt für einige Beispielformeln angewendet werden. Dabei wird zuerst zu einer PTL-Formel ein erkennender Automat konstruiert und dieser dann in einen CSP-Prozess umgewandelt. Der Automat wird sowohl als Graph als auch tabellarisch dargestellt. Als besondere Zustände sind

- Büchi-Zustände durch $\bigcirc$,
- Endzustände durch $\circ$ und
- Büchi- und Endzustände durch $\circledcirc$,

markiert.

6.1 Eventually c

Die Formel $\Diamond c$ beschreibt, dass jeder unendliche oder maximale Trace irgendwann ein c enthält. Dazu kann der Automat in Abbildung 1 konstruiert werden. Sobald ein c gelesen wird, wechselt der Automat in den Büchi- und Endzustand ev_1 und bleibt für jede Eingabe in diesem Zustand.

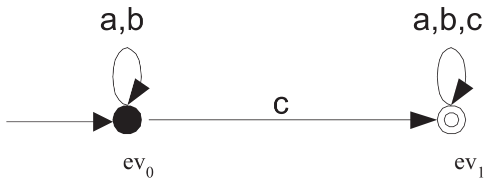


Abbildung 1: Automat $\mathcal{A}_{EV}$ zu $\Diamond c$

Q	Σ	Q
ev_0	a, b	ev_0
ev_0	c	ev_1
ev_1	a, b, c	ev_1

Abbildung 2: Transitionen von $\mathcal{A}_{EV}$

Automat $\rightsquigarrow$ CSP-Prozesse Zu dem Automaten $\mathcal{A}_{EV}$ mit den beiden Zuständen werden CSP-Prozesse EV_0 und EV_1 konstruiert. EV_0 ist durch

$$EV_0 = c \to EV_1 \,\Box\, a \to EV_0 \,\Box\, b \to EV_0$$

definiert, und EV_1 durch

$$EV_1 = a \to \mathbf{acc} \to EV1 \Box b \to \mathbf{acc} \to EV1 \Box c \to \mathbf{acc} \to EV1 \Box \mathbf{acc} \to EV1,$$

mit **acc**-Kommunikationen nach jeder normalen Kommunikation, da ev_1 in $\mathcal{A}_{EV}$ ein Büchi-Zustand ist und mit der sofortigen **acc**-Kommunikation, da ev_1 auch ein Endzustand ist. Für FDR werden die Prozesse in ASCII-Notation dargestellt:

```
channel a,b,c,acc

EV0 = c -> EV1 [] a -> EV0 [] b -> EV0
EV1 = a -> acc -> EV1 [] b -> acc -> EV1
   [] c -> acc -> EV1 [] acc -> EV1
```

Die möglichen Kommunikationen müssen für FDR über das Schlüsselwort `channel` deklariert werden. Für die folgenden Prozesse soll getestet werden, ob sie die Formel $\Diamond c$ erfüllen:

```
I1 = a-> b -> I1
I2 = a-> (b -> I2
          |~| c -> STOP)
I3 = a->b->c->STOP
```

Offensichtlich erfüllt `I3` die Formel und `I1` nicht. Der Prozess `I2` erfüllt $\Diamond c$ ebenfalls nicht, da er sich nichtdeterministisch dafür entscheiden kann unendlich oft a und dann b zu kommunizieren. Um das auch mit FDR testen zu können, werden Testprozesse konstruiert:

```
CHK1 = I1 [{a,b,c}||{a,b,c,acc}] EV0
CHK2 = I2 [{a,b,c}||{a,b,c,acc}] EV0
CHK3 = I3 [{a,b,c}||{a,b,c,acc}] EV0
```

Zusammen mit dem Prozess `INFACC` kann dann die Gültigkeit der Formel überprüft werden:

```
INFACC = acc -> INFACC

assert INFACC [FD= CHK1\{a,b,c} -- FDR liefert NEIN
assert INFACC [FD= CHK2\{a,b,c} -- FDR liefert NEIN
assert INFACC [FD= CHK3\{a,b,c} -- FDR liefert JA
```

6.2 Always a

Die Formel $\Box a$ ist nur dann erfüllt, falls jeder unendliche oder maximale Trace nur aus a besteht, d.h. der zugehörige CSP-Prozess kann immer nur a kommunizieren. Im ersten Schritt ist wieder der Automat $\mathcal{A}_{Always}$ zu konstruieren, der nur Wörter akzeptiert, die nur as enthalten. So ein Automat ist in Abbildung 3 dargestellt. Der Automat $\mathcal{A}_{Always}$ kann in CSP dann

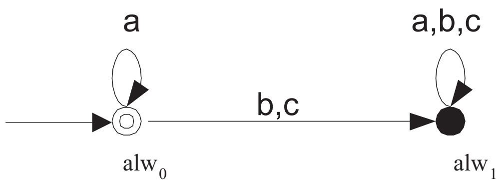


Abbildung 3: Automat $\mathcal{A}_{Always}$ zu $\Box a$

durch folgende Prozesse (in FDR-Notation) simuliert werden:

```
ALW0 = a -> acc -> ALW0 [] b -> acc -> ALW1
    [] c -> acc -> ALW1 [] acc -> ALW0
ALW1 = a -> ALW1 [] b -> ALW1 [] c -> ALW1
```

Als Beispielprozesse können

Q	Σ	Q
alw_0	a	alw_0
alw_0	b, c	alw_1
alw_1	a, b, c	alw_1

Abbildung 4: Transitionen von $\mathcal{A}_{Always}$

```
I1 = a -> b -> I1
I2 = a -> STOP
I3 = a -> I3 [] b -> I3
```

dienen. Als einziger Prozess erfüllt `I2` die Formel, da es keine unendlichen Traces gibt und der einzige maximale Trace nur aus einem a besteht. Der Test erfolgt dann mit FDR:

```
CHKALW1 =I1 [{a,b,c}||{a,b,c,acc}] ALW0
CHKALW2 =I2 [{a,b,c}||{a,b,c,acc}] ALW0
CHKALW3 =I3 [{a,b,c}||{a,b,c,acc}] ALW0

assert INFACC [FD= CHKALW1\{a,b,c} -- FDR liefert NEIN
assert INFACC [FD= CHKALW2\{a,b,c} -- FDR liefert JA
assert INFACC [FD= CHKALW3\{a,b,c} -- FDR liefert NEIN
```

6.3 Schwaches Until

Die etwas komplexere Formel $\phi = (\neg a\ \mathcal{U}\ (a \wedge \bigcirc\Diamond b)) \vee (\Box\neg a)$ wird von allen Traces erfüllt, die nach dem ersten a ein irgendwann b enthalten. Da die Formel auch erfüllt sein soll, falls der Trace kein a enthält, wird ein schwaches Until in der Formel modelliert. Der Automat $\mathcal{A}_\phi$ in Abbildung 5 erkennt genau diese Wörter. Wie bei den anderen Beispielen, wird der Automat durch CSP-Prozesse `F0,F1,F2` simuliert.

```
F0 = a -> F1 [] b -> acc -> F0 [] c -> acc -> F0 [] acc -> F0
F1 = a -> F1 [] b -> F2 [] c -> F1
F2 = a -> acc -> F2 [] b -> acc -> F2
  [] c -> acc -> F2 [] acc -> F2
```

Von den Beispielprozessen

```
I1 = a-> b -> I1
I2 = a-> (b -> I2
         |~| c -> STOP)
I3 = a -> c -> c -> c-> b -> I3
I4 = c-> I4
```

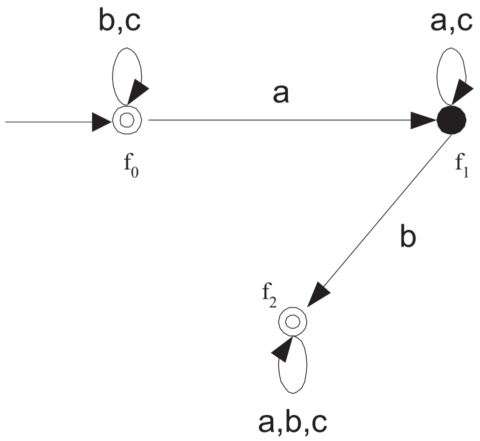


Abbildung 5: Automat $\mathcal{A}_\phi$ zu $\phi = (\neg a\ \mathcal{U}\ (a \wedge \bigcirc\Diamond b)) \vee (\Box\neg a)$

Q	Σ	Q
f_0	b, c	f_0
f_0	a	f_1
f_1	a, c	f_1
f_1	b	f_2
f_2	a, b, c	f_2

Abbildung 6: Transitionen von $\mathcal{A}_\phi$

erfüllen `I1` und `I3` die Formel, weil nach der ersten a-Kommunikation nach null bzw. vier Schritten b kommuniziert wird. Prozess `I4` erfüllt die Formel, da a nie kommuniziert wird. Der Prozess `I2` erfüllt die Formel nicht, da nach dem a ein c kommuniziert und gestoppt werden kann. Der Test in FDR erfolgt ebenfalls durch Konstruktion eines Test-Prozesses und failures-divergences-refinement-Prüfung gegen `INFACC`.

```
CHKF1 =I1 [{a,b,c}||{a,b,c,acc}] F0
CHKF2 =I2 [{a,b,c}||{a,b,c,acc}] F0
CHKF3 =I3 [{a,b,c}||{a,b,c,acc}] F0
CHKF4 =I4 [{a,b,c}||{a,b,c,acc}] F0

assert INFACC [FD= CHKF1\{a,b,c} -- FDR liefert JA
assert INFACC [FD= CHKF2\{a,b,c} -- FDR liefert NEIN
assert INFACC [FD= CHKF3\{a,b,c} -- FDR liefert JA
assert INFACC [FD= CHKF4\{a,b,c} -- FDR liefert JA
```

6.4 Unterschied zwischen Büchi- und Endzuständen

Bei den bisherigen Beispielen, wurde die Unterscheidung zwischen Büchi- und Endzuständen noch nicht benötigt. Erst die Formel $\phi' = \Box(b \rightarrow \Box \bigcirc a)$ erfordert die Unterscheidung. Durch sie wird gefordert, dass nach einem b eine *unendliche* Folge von a steht. Maximale Traces, die ein b enthalten, können diese Formel also nicht erfüllen. Der Automat $\mathcal{A}_{\phi'}$ in Abbildung 7 akzeptiert Traces (maximal oder unendlich), die kein b enthalten mit Zustand b_0. Sobald ein b gelesen wird, wechselt der Automat in Zustand b_1. Dieser Zustand ist kein Endzustand, nur Büchizustand. Wenn auf das b nur noch a und unendlich viele davon folgen, wird der Büchi-Zustand unendlich oft durchlaufen und das Wort akzeptiert. Folgen aber nur endlich viele a bleibt der Automat in dem nicht-Endzustand stecken und das Wort wird nicht akzeptiert. Wird in Zustand b_1 ein b oder c gelesen wechselt $\mathcal{A}_{\phi'}$ in den Fehlerzustand b_2, der nicht mehr verlassen werden kann. Der Unterschied

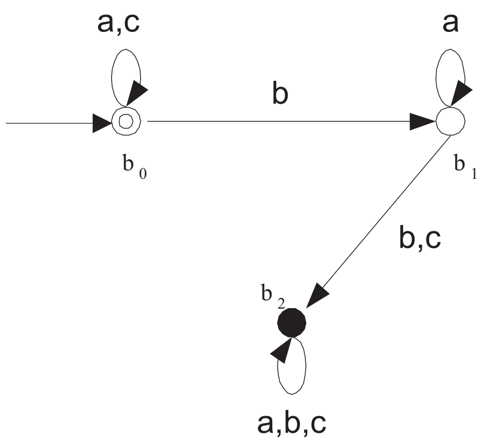


Abbildung 7: Automat $\mathcal{A}_{\phi'}$ zu $\phi' = \Box(b \rightarrow \Box \bigcirc a)$

Q	Σ	Q
b_0	a, c	b_0
b_0	b	b_1
b_1	a	b_1
b_1	b, c	b_2
b_2	a, b, c	b_2

Abbildung 8: Transitionen von $\mathcal{A}_{\phi'}$

zwischen Büchi- und Endzuständen zeigt sich dann auch in den zugehörigen CSP-Prozessen.

```
B0 = a -> acc -> B0 [] b -> acc -> B1
  [] c -> acc -> B0 [] acc -> B0
B1 = a -> acc -> B1 [] b -> acc -> B2 [] c -> acc -> B2
B2 = a -> B2 [] b -> B2 [] c -> F2
```

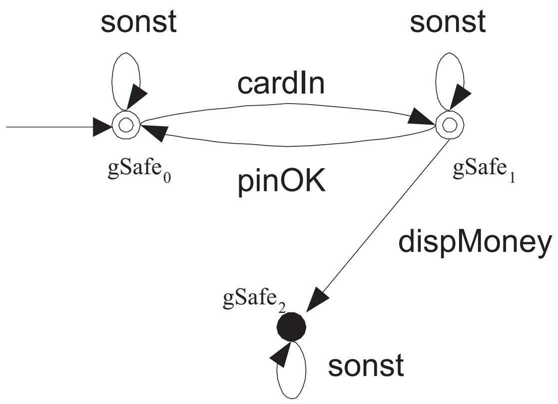
Da b_0 auch ein Endzustand ist, kann `B0` **acc** kommunizieren, ohne vorher a, b, oder c zu kommunizieren. Für `B1` ist das nicht möglich, da b_1 im zugehörigen Automaten $\mathcal{A}_{\phi'}$ nur ein Büchi-Zustand ist. Falls der zu testende Prozess `TEST` in

```
CHKTEST = (TEST [{a,b,c}||{a,b,c,acc}] B0) \{a,b,c}
```

steckenbleibt, während er parallel zu `B1` läuft, bleibt auch `B1` stecken, da beide Seiten über a,b und c synchronisiert sind und `B1` nur diese Kommunikationen erlaubt.

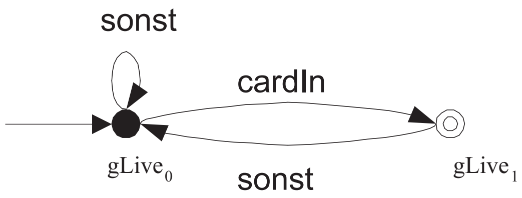
6.5 Geldautomat

Zu dem in Abschnitt 2.3 in CSP definierten Geldautomaten können die in Abschnitt 3.2 durch PTL-Formeln beschrieben Eigenschaften durch die Automaten in den Abbildungen 9 und 10 verifiziert werden. Diese beiden Au-



$$\psi_S = \Box(\mathbf{cardIn} \rightarrow ((\neg\mathbf{dispMoney} \wedge \neg\mathbf{cardIn})\ \mathcal{U}\ (\mathbf{pinOK} \vee \mathbf{cardIn})))$$

Abbildung 9: Automat $\mathcal{A}_{\psi_S}$ zu ψ_S



$$\psi_L = \Box(\Diamond\mathbf{cardIn})$$

Abbildung 10: Automat $\mathcal{A}_{\psi_L}$ zu ψ_L

tomaten können durch die CSP-Prozesse `GSAVE0` ...`GSAVE2` und `GLIVE0`, `GLIVE1` simuliert werden.

Prozesse zu $\mathcal{A}_{\psi_S}$:

```
sonst0 = {pinIn,reqIn,chkPin,pinOK,pinNOK,cardOut,dispMoney}
GSAVE0 = cardIn -> acc -> GSAVE1 []
         acc -> GSAVE0 []
        ([] event : sonst0  @ event -> acc -> GSAVE0)
sonst1 = {cardIn,pinIn,reqIn,chkPin,pinNOK,cardOut}
GSAVE1 = pinOK -> acc -> GSAVE0 []
         dispMoney -> acc -> GSAVE2 []
         acc -> GSAVE1 []
        ([] event : sonst1  @ event -> acc -> GSAVE1)
sonst2 = {cardIn,pinIn,reqIn,chkPin,pinOK,pinNOK,cardOut,dispMoney}
GSAVE2 = ([] event : sonst2  @ event -> GSAVE2)
```

Prozesse zu $\mathcal{A}_{\psi_L}$:

```
sonst0 = {pinIn,reqIn,chkPin,pinOK,pinNOK,cardOut,dispMoney}
GLIVE0 = cardIn -> GLIVE1 []
         ([] event : sonst0  @ event -> GSAVE0)
sonst1 = {cardIn,pinIn,reqIn,chkPin,pinOK,pinNOK,cardOut,dispMoney}
GLIVE1 = acc -> GLIVE1 []
         ([] event : sonst1  @ event -> acc -> GSAVE0)
```

6.6 Grenzen des Verfahrens

Das letzte Beispiel zeigt eine Formel, zu der kein erkennender deterministischer Büchi-Automat konstruiert werden kann.

Beispiel Zu der PTL-Formel

$$\psi = \Diamond\Box a$$

kann kein deterministischer Büchi-Automat konstruiert werden.

Beweis Annahme: Es existiert ein deterministischer Büchi-Automat $\mathcal{D}$, der die Sprache $L(\psi)$ erkennt. Dann existiert eine Zahl $k_0 > 0$, so dass $\mathcal{D}$ mit ba^{k_0} in den Endzustand f_0 gelangt, da $baaa\ldots$ in $L(\psi)$ liegt. Entsprechend existiert $k_1 > 0$, so dass $\mathcal{D}$ mit $ba^{k_0}ba^{k_1}$ in einen Endzustand f_1 gelangt und wegen des Determinismus dabei f_0 durchläuft. Also existiert eine Folge $k_0, k_1, k_2, \ldots$, so dass $\mathcal{D}$ mit $\alpha = ba^{k_0}ba^{k_1}ba^{k_2}\ldots$ die Endzustände $f_0, f_1, f_2, \ldots$ durchläuft. Da die Zustandsmenge endlich ist, muss mindestens ein Endzustand unendlich oft durchlaufen werden und $\mathcal{D}$ akzeptiert α, obwohl $\alpha \notin L(\psi)$. Somit kann ein deterministischer Automat $\mathcal{D}$ nicht existieren.

Somit ist für die Formel ψ das Verfahren nicht anwendbar.

Literatur

[1] C.A.R. Hoare: *Communicating Sequential Processes*, Prentice-Hall International Series in Computer Science, 1985.

[2] A.W. Roscoe: *The Theory and Practice of Concurrency*, Prentice-Hall International Series in Computer Science, 1998.

[3] Z. Manna, A. Pnueli: *The Temporal Logic of Reactive and Concurrent Systems -Specification-*, Springer-Verlag New York, 1992.

[4] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *IEEE Symposium on Logic in Computer Science (LICS)*, volume 1, pages 332–344, 1986.

[5] FDR-Manual, Handbuch zu FDR.