CARL VON OSSIETZKY universität OLDENBURG
FAKULTÄT II
INFORMATIK, WIRTSCHAFTS- UND RECHTSWISSENSCHAFTEN

Individuelles Projekt

Reguläres Model-Checking für parametrisierte Phasen-Event-Automaten

Boris Rosenow

31. Mai 2010

Erstgutachter: Prof. Dr. Ernst-Rüdiger Olderog
Zweitgutachter: Dipl.-Inform. Tim Strazny

Inhaltsverzeichnis

1 Einleitung

Automatische System-Verifikation, das sogenannte *Model-Checking*, existiert für eine Vielzahl unterschiedlicher Anforderungen. In der Abteilung *Correct Systems Design* der *Carl von Ossietzky Universität* wurde zu diesem Zweck das *Phasen-Event-Automaten-Modell* entwickelt, und so die Möglichkeit geschaffen, Systeme unterschiedlicher Ausprägungen zu modellieren und beispielsweise durch den *Abstraction Refinement Model Checker* (*ARMC*) zu verifizieren.

Parametrisierte Systeme bedürfen jedoch besonderer Methoden, da es sich dabei um den Verbund beliebig vieler paralleler Prozesse handelt, die auf bestimmte Weise miteinander kommunizieren und sich synchronisieren. Die einheitliche Verifikation derartiger Systeme für eine beliebige Anzahl von Prozessen ermöglicht das *reguläre Model-Checking*. Eine Grundlage dazu, dies auf das Phasen-Event-Automaten-Modell anwenden zu können, wird in dieser Arbeit behandelt.

In Kapitel 2 werden die dafür benötigten Begrifflichkeiten aufgeführt und erläutert. Es beschäftigt sich mit dem Begriff *Model-Checking* an sich, bevor über eine Vertiefung der *Parametrisierung* zum *regulären Model-Checking* übergegangen wird. Abgeschlossen wird das Kapitel mit der Erklärung des *Phasen-Event-Automaten*-Modells.

Kapitel 3 behandelt daraufhin, auf welche Weise die Parametrisierung von Phasen-Event-Automaten durchgeführt wird. Des weiteren beschäftigt es sich damit, einen formalen Algorithmus aufzuzeigen und zu erläutern, um die zum regulären Model-Checking benötigten *Transducer* aus einem solchen parametrisierten Phase-Event-Automaten zu generieren.

Eine Beschreibung der Implementierung des formalen Algorithmus innerhalb des *PEA-Toolkits* bildet den Inhalt von Kapitel 4. Das PEA-Toolkit ist ein in der Abteilung CSD entwickeltes auf Java basierendes Werkzeugpaket zum Umgang mit Phasen-Event-Automaten.

Die korrekte Funktionsweise des implementierten Algorithmus wird in Kapitel 5 anhand des in [KMM+01] verwendeten parametrisierten Programms MUX gezeigt. Dabei wird dies zunächst in ein entsprechendes Schema der Spezifikationssprache für Phasen-Event-Automaten CSP-OZ-DC übersetzt und daraus ein Phasen-Event-Automat erzeugt,

der darauf als Eingabe des Algorithmus implementiert wurde.

Den Abschluss der Arbeit bildet Kapitel 6. Darin werden die Ergebnisse zusammengefasst und Ansatzmöglichkeiten aufgezeigt, um diese zu erweitern.

2 Grundlagen

2.1 Model-Checking

Das *Model-Checking* ermöglicht die automatisierte Verifikation zustandsendlicher Systeme. Dabei kommt eine Vielzahl unterschiedlicher Methoden zum Einsatz, jedoch ist deren Struktur in jedem Fall die gleiche (nach [CGP99]):

Modellierung Das zu verifizierende System wird in eine für das Model-Checking verständliche Form übersetzt. Dabei kann das komplette System vollständig modelliert werden, oder aber aufgrund zu hoher Komplexität von gewissen Details abstrahiert und somit nur auf die für die Korrektheit des System relevanten Aspekte eingegangen werden.

Spezifikation Die zu überprüfenden Anforderungen an das System müssen ermittelt und spezifiziert werden. In der Regel geschieht dies mittels temporallogischer Formeln.

Dieser Schritt hat eine hohe Bedeutung, da durch das Model-Checking nur aussagekräftige Ergebnisse geliefert werden können, wenn die Anforderungen vollständig spezifiziert wurden.

Verifikation Das modellierte System muss gegen die ermittelten Anforderungen geprüft werden. Dies geschieht im Idealfall vollautomatisch, bedarf jedoch in der Praxis häufig einer gewissen Interaktion. So ist es beispielsweise notwendig, die Ergebnisse der Verifikation zu überprüfen und auf Fehlerfälle zu reagieren. In der Regel wird bei einem Fehlerfall der Ablauf des Systemmodells angegeben, der zu der Verletzung einer der Anforderungen geführt hat. Dies soll dabei helfen, entsprechend das System oder, sofern der Fehler im Unterschied zwischen System und Modell besteht, das Modell, z.B. durch Änderung des Abstraktionsgrads, anzupassen. Anschließend muss das angepasste Modell erneut verifiziert werden.

2.2 Parametrisierung

Ein *parametrisiertes System* besteht aus der parallelen Komposition von Prozessen, die sich lediglich durch ihre Kennung (*ID*) unterscheiden. Innerhalb dieser Prozesse wird jeweils das gleiche Programm ausgeführt, welches Zugriff auf die ID des ausführenden Prozesses hat und dadurch das Verhalten bestimmen kann (vgl. [Xu05, Abschnitt 2.2.1]). In Abbildung 2.1 wird dieses Prinzip anhand einer Grafik anschaulich dargestellt.

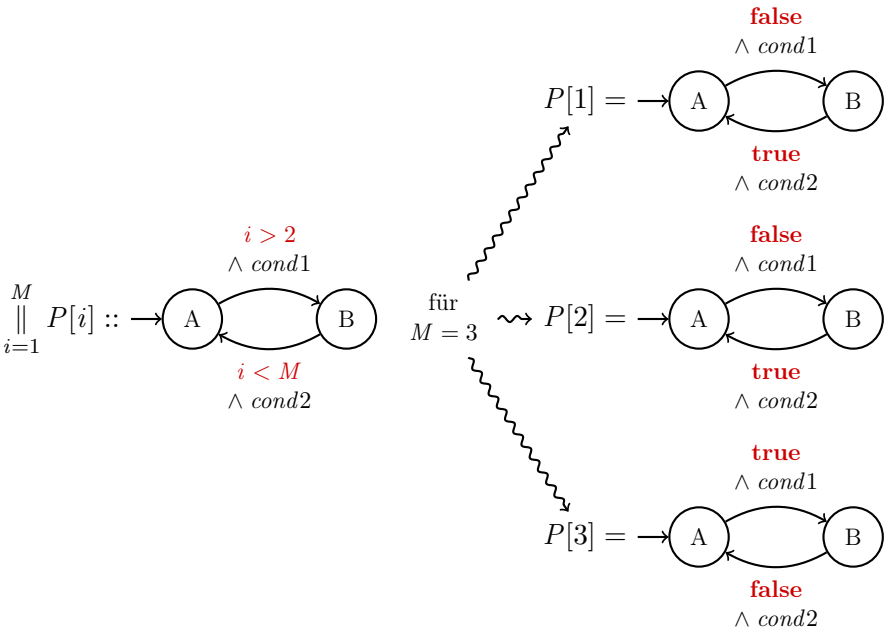


Abbildung 2.1: Beispiel eines parametrisierten Systems

Ein solches parametrisiertes System wird mit einer beliebigen natürlichen Zahl $M \in \mathbb{N}^+$ instantiiert. Dabei entspricht M der Anzahl der parallelen symmetrischen Prozesse. Innerhalb dieser Arbeit werden dabei die eindeutigen IDs der Prozesse als ganzzahliger Wert im Bereich $1 \dots M \in \mathbb{N}$ aufgefasst. Innerhalb des Beispiels aus obiger Abbildung wird deutlich, dass sich die Prozesse $P[i]$ des instantiierten parametrisierten Systems lediglich durch ihre Parameter i unterscheiden und sich dementsprechend verhalten.

Ein derartiges parametrisiertes System zu verifizieren, stellt ein nicht zu unterschätzendes Problem dar. So ist dies zwar durch einfaches Model-Checking für Systeme mit einer festen Größe ohne weiteres möglich, jedoch bedarf eine einheitliche Verifikation für Systeme jeder möglichen Größe einer erweiterten Verifikations-Methode. Eine solche Methode ist das in dieser Arbeit betrachtete *reguläre Model-Checking*, welches im nächsten Abschnitt genauer erläutert wird.

2.3 Reguläres Model-Checking

Reguläres Model-Checking dient der einheitlichen Verifikation parametrisierter Systeme mit beliebiger, aber endlicher Größe. Um dies zu ermöglichen, werden die verschiedenen Zustände des Systems (*Konfigurationen*) als Wörter einer regulären Sprache über dem Alphabet Σ dargestellt. Σ entspricht der Menge der Zustände, in der sich eine einzelne System-Komponente befinden kann. In dieser Arbeit handelt es sich dabei um die Zustandsnamen der Komponente, es können aber auch Repräsentationen der Variablenwerte verwendet werden.

Programmschritte des gesamten Systems werden als Wörter über dem Alphabet $\Sigma \times \Sigma$ dargestellt. Konfigurationspaare, die gültige Folgekonfigurationen des parametrisierten Systems sind, werden durch sogenannte *Transducer* ermittelt.

Definition 1. Ein Transducer $\mathcal{T}$ ist ein endlicher Automat, der die folgende Form hat:

$$\mathcal{T} = (\Sigma_\mathcal{T}, Q_\mathcal{T}, q_0, E_\mathcal{T}, F_\mathcal{T})$$

mit

- $\Sigma_\mathcal{T} \subseteq \Sigma \times \Sigma$: das Alphabet des Transducers. Jedes Symbol $[a_1, a_2]$ des Alphabets entspricht einem gültigen Schritt einer Komponente des parametrisierten Systems.
- $Q_\mathcal{T}$: endliche Menge von Zuständen.
- $q_0 \in Q_\mathcal{T}$: Startzustand.
- $E_\mathcal{T} \subseteq Q_\mathcal{T} \times \Sigma_\mathcal{T} \times Q_\mathcal{T}$: endliche Kantenmenge. Jede Kante $(q, a, q') \in E_\mathcal{T}$ entspricht einem Schritt des Transducers von Zustand q in Zustand q', indem das Symbol a gelesen wurde.
- $F_\mathcal{T} \subseteq Q_\mathcal{T}$: endliche Menge von Endzuständen.

Die Konfigurationen, die gültige Folgekonfigurationen der Menge der Startkonfiguration I darstellen, werden durch $I \circ \mathcal{T}$ ermittelt. Dabei entspricht $I \circ \mathcal{T}$ der Menge:

$$\{(b_1, \ldots, b_M) \in \Sigma_\mathcal{T}^* \mid \exists (a_1, \ldots, a_M) \in I : ([a_1, b_1], \ldots, [a_M, b_M]) \in \mathcal{L}(\mathcal{T})\}$$

Die Menge der erreichbaren Zustände des parametrisierten Systems wird durch $I \circ \mathcal{T}^*$ ermittelt. Dies entspricht dabei der Menge:

$$\bigcup_{i=0}^{\infty} I \circ \mathcal{T}^i$$

Genauer bedeutet dies:

$$I \circ \mathcal{T}^0 := I, I \circ \mathcal{T}^{i+1} = (I \circ \mathcal{T}^i) \circ \mathcal{T}$$

Ziel des regulären Model-Checkings ist es dabei, diese Menge auf Vorkommen von Fehlerzuständen zu überprüfen. Dazu wird eine Fehlerzustandsmenge BAD konstruiert. In dieser Menge enthalten sind Konfigurationswörter, die die entsprechenden Fehlerzustände repräsentieren. Ist der Schnitt dieser Menge mit der Menge der erreichbaren Zustände des Systems nicht leer, sind Fehlerzustände erreichbar und das parametrisierte System fehlerhaft. Die Prüfung auf Korrektheit des Systems ergibt sich demnach dadurch, ob die folgende Gleichung erfüllt wird:

$$I \circ \mathcal{T}^* \cap BAD = \varnothing$$

Problematisch ist dabei die algorithmische Berechnung von $\mathcal{T}^*$. Dies kann nicht iterativ durch die Berechnung von $\mathcal{T}^n$ für jedes $n \in \mathbb{N}$ erfolgen, da dies zu keinem Ende führt. In [AJNS04, Abschnitt 3] wird eine Auswahl von Methoden vorgestellt, mit der (eine Überapproximation von) $\mathcal{T}^*$ berechnet werden kann. Diese Methoden werden im Folgenden kurz erläutert.

Quotienting Beim Quotienting wird zunächst $\mathcal{T}^n$ für eine bestimmte Anzahl von Werten für n berechnet. Daraufhin müssen Relationen zwischen den Zuständen der entstehenden Transducer gefunden und ein Transducer erzeugt werden, der die entsprechenden Zustände zusammenfasst. Um dabei den Sprachraum der Transducer nicht zu vergrößern, wird zwischen Vorwärts- und Rückwärtsbisimulationen unterschieden, die darauf zu einer Äquivalenzrelation kombiniert werden können, wodurch der Transducer $\mathcal{T}^+$ gebildet wird.

Abstraction Bei der Abstraction wird nicht unmittelbar die Berechnung von $\mathcal{T}^*$ behandelt. Vielmehr werden die Automaten berechnet, die durch die mehrfache Anwendung von $\mathcal{T}$ auf die Anfangskonfiguration I entstehen. Die nach jedem Schritt entstehenden Automaten werden auf Eigenschaften untersucht, die für $I \circ \mathcal{T}^* \cap BAD = \varnothing$ nicht von Bedeutung sind. Kann anhand dieser Eigenschaften eine Äquivalenzrelation zwischen den Automaten der einzelnen Schritte gefunden werden, ist es möglich, daraus einen abstrakten Automaten $\mathcal{T}^{lim}$ zu bilden, der eine Überapproximation von $I \circ \mathcal{T}^*$ darstellt. lim stellt dabei eine Obergrenze dar, die sich aus der ermittelten Äquivalenzrelation ergibt.

Gilt $\mathcal{T}^{lim} \cap BAD = \varnothing$, ist sichergestellt, dass ebenfalls $I \circ \mathcal{T}^* \cap BAD = \varnothing$ erfüllt ist,

da die Sprache von $I \circ \mathcal{T}^*$ Teilmenge der Sprache von $\mathcal{T}^{lim}$ ist ($\mathcal{L}(I \circ \mathcal{T}^*) \subseteq \mathcal{L}(T^{lim})$).

Extrapolation Ebenso wie bei der Abstraction wird bei der Extrapolation direkt $I \circ \mathcal{T}^*$ bestimmt. Dazu werden für eine bestimmte Anzahl an Anwendungen von $\mathcal{T}$ auf I die entstehenden Folgekonfigurationen ermittelt. Wird ein Muster entdeckt, kann dadurch darauf geschlossen werden, welchen Effekt die Durchführung einer beliebigen Anzahl an Iterationen auf I haben wird. Dadurch entsteht ebenso wie bei der Abstraction i.A. eine Überapproximation von $I \circ \mathcal{T}^*$, unter bestimmten Bedingungen ist es aber auch möglich, dass das Ergebnis exakt $I \circ \mathcal{T}^*$ entspricht.

2.4 Phasen-Event-Automaten

Bei *Phasen-Event-Automaten* (*PEA*) handelt es sich um spezielle Timed Automata. Sie wurden von Jochen Hoenicke im Rahmen seiner Dissertation [Hoe06] als operationelle Semantik für die von ihm entwickelte Spezifikationssprache *CSP-OZ-DC* eingeführt. Diese vereint die drei Spezifikationssprachen *CSP*, *Object Z* und den *Duration Calculus*.

Der CSP-Teil dient dazu, durch Kommunikationskanäle sogenannte Events zu kommunizieren, die das Verhalten eines Systems und die Interaktion mit seiner Umwelt bestimmen. Dabei wird zwischen lokalen und globalen Kanälen unterschieden. Die lokalen Kanäle dienen in der Regel dazu, das interne Verhalten eines Prozesses zu beschreiben, wobei globale Kanäle der Kommunikation zwischen mehreren Prozessen dienen. Dabei ist es außerdem möglich, einen bestimmten Wert über einen Kanal zu kommunizieren, durch den das resultierende Verhalten weiter spezifiziert wird.

Im Object Z-Teil werden verschiedene Variablen und Datentypen, sowie durch Events entstehende Änderungen der Variablenwerte definiert. Außerdem kann angegeben werden, welche Werte Variablen besitzen müssen, damit das Event eintreten kann.

Der Duration Calculus dient der Spezifikation von Realzeitbedingungen, die an das System gestellt werden. Diese Bedingungen werden in Form von Gegenbeispielen angegeben, welche das System nicht erfüllen darf.

In [Hoe06, Kapitel 5] stellt Hoenicke eine Vorgehensweise bereit, ein CSP-OZ-DC-Schema in einen PEA zu übersetzen. Dabei müssen zunächst der CSP-, der OZ- sowie der DC-Teil in eigene PEA übersetzt werden, welche darauf durch parallele Komposition vereinigt werden.

Nachfolgend wird die formale Definition von PEA angegeben, sowie die Definitionen von Stotterinvarianz und der parallelen Komposition von PEA. Diese finden sich in [Hoe06, Abschnitt 4.2] wieder.

Definition 2. Ein *Phasen-Event-Automat* $\mathcal{A} = (P, V, A, C, E, s, I, E_0)$ besteht aus folgenden Komponenten:

- P: eine endliche Menge von *Zuständen* (*Phasen*).
- $V \subseteq NAME$: eine endliche Menge von *Variablen*.
- $A \subseteq NAME$: eine endliche Menge boolescher *Event-Variablen*.
- C: eine endliche Menge von *Uhren*.
- $E \subseteq P \times \mathcal{L}(V \cup A \cup C \cup V') \times \mathbb{P}(C) \times P$: eine endliche Menge von *Transitionen*. Ein Element (p, g, X, p') dieser Menge entspricht einer Kante vom Zustand p zum Zustand p'. Die Belegung der Variablen und Uhren, sowie die auftretenden Events müssen dem der Kantenbedingung g entsprechen. Alle Uhren aus X werden nach der Transition zurückgesetzt.
- $s : P \to \mathcal{L}(V)$: eine Beschriftungsfunktion, die jedem Zustand eine *Zustandsinvariante* zuweist, die während der Belegung des Zustands erfüllt sein muss.
- $I : P \to \mathcal{L}^c(C)$: eine Funktion, die jedem Zustand eine *Uhreninvariante* zuweist.
- $E_0 \subseteq \mathcal{L}(V) \times P$: eine endliche Menge aus *Startzuständen*. Ein Element (g, p) dieser Menge ermöglicht es dem Automaten in Zustand p zu starten, wenn die Bedingung g erfüllt ist.

Bei einer parallelen Komposition von PEA nehmen alle PEA zur gleichen Zeit eine Transition. Dabei kann eine interne Operation ausgeführt werden, oder aber zwei oder mehr PEA führen einen gemeinsamen Schritt aus, der durch einen globalen Kommunikationskanal bestimmt wird. Um es einem PEA zu ermöglichen, gleichzeitig mit den anderen PEA einen Schritt durchzuführen, ohne dabei die Werte seiner Variablen oder Uhren zu verändern, werden jedem Zustand sogenannte *Stotterkanten* als besondere Transitionen hinzugefügt. Eine solche Transition $(p, g, \varnothing, p)$ fordert in ihrer Kantenbedingung g, dass keinerlei Veränderungen an den Variablenwerten sowie keine Events auftreten. Außerdem müssen auch nach Ausführen der Transition sämtliche Zustands- und Uhreninvarianten weiterhin erfüllt sein. Formal beschreibt dies die folgende Definition.

Definition 3. Ein Automat $\mathcal{A}$ ist *stotterinvariant*, wenn für jeden Zustand $p \in P$ eine *Stotterkante* $(p, g, \varnothing, p) \in E$ existiert, mit

$$(\forall\, x : V \bullet x = x') \wedge (\forall\, e : A \bullet \neg\, e) \wedge s(p) \wedge \mathit{strict}(I(p)) \Rightarrow g.$$

Die Funktion *strict* sorgt dafür, dass es dem PEA ermöglicht werden muss, für eine kurze positive Zeitspanne in dem gewünschten Zustand zu verweilen. Dabei werden sämtliche Vorkommen von $\leq$ innerhalb der Uhreninvarianten $I(p)$ durch $<$ ersetzt.

Definition 4. Die *parallele Komposition zweier Automaten* $\mathcal{A}_1$ *und* $\mathcal{A}_2$, mit $\mathcal{A}_j = (P_j, V_j, A_j, C_j, E_j, s_j, I_j, E_{0,j})$, ist der Produktautomat

$$\mathcal{A} = (P, V, A, C, E, s, I, E_0)$$

definiert wie folgt:

- $P := \{p_1 : P_1;\ p_2 : P_2 \mid s(p_1) \wedge s(p_2) \not\Rightarrow \text{false} \bullet (p_1, p_2)\}$. Die Menge von Zuständen besteht aus dem Kreuzprodukt der Zustandsmengen P_1, P_2, wobei die Zustände mit einer kombinierten Zustandsinvariante äquivalent zu false entfernt werden.
- $V := V_1 \cup V_2$ sowie $A := A_1 \cup A_2$.
- $C := C_1 \dot{\cup} C_2$. Die Menge von Uhren besteht aus der disjunkten Vereinigung der Mengen C_1 und C_2; Uhren, die in beiden Mengen vorkommen, müssen umbenannt werden.
- Die Menge der Transitionen E besteht aus den Tupeln $((p_1, p_2), g_1 \wedge g_2, X_1 \cup X_2, (p_1', p_2'))$ für jedes Transitionspaar $(p_j, g_j, X_j, p_j') \in E_j, j = 1, 2$. Transitionen, für die $g_1 \wedge g_2$ äquivalent zu false ist, können entfernt werden.
- $s(p_1, p_2) = s_1(p_1) \wedge s_2(p_2)$. Die Zustandsinvariante eines der Zustände von $\mathcal{A}$ ergibt sich aus der Konjunktion der Zustandsinvarianten der entsprechenden Zustände aus $\mathcal{A}_1$ und $\mathcal{A}_2$.
- $I(p_1, p_2) = I_1(p_1) \wedge I_2(p_2)$. Die Uhreninvariante eines der Zustände von $\mathcal{A}$ ergibt sich aus der Konjunktion der Uhreninvarianten der entsprechenden Zustände aus $\mathcal{A}_1$ und $\mathcal{A}_2$.
- Die Menge der Startzustände E_0 besteht aus den Tupeln $(g_1 \wedge g_2, (p_1, p_2))$ für jedes Startzustandspaar $(g_j, p_j) \in E_{0,j}$.

3 Reguläres Model-Checking parametrisierter Phasen-Event-Automaten

3.1 Parametrisierung von Phasen-Event-Automaten

Um einen PEA zu parametrisieren, wird zunächst sein Modell um den Parameter i erweitert. Um eine Grundlage zur besseren Einbindung der Parametrisierung in das Werkzeug *Syspect*[1] zu ermöglichen, wurde dies bereits auf CSP-OZ-DC-Basis umgesetzt.

CSP-OZ-DC bietet bereits eine Möglichkeit, Parameter an eine Klasse zu übergeben. Diese bestehen für die korrekte Parametrisierung aus dem Wertebereich des Parameters, sowie dem Parameter selbst. Der Wertebereich muss dabei Zahlen 1 bis M entsprechen. M stellt die Ausprägung des parametrisierten PEA-Systems dar. Im weiteren Verlauf wird für diesen Wertebereich das Symbol $\mathbb{M}$ verwendet. Der Parameter selber wird mit dem Buchstaben i gekennzeichnet. Dieser muss in der Variablendeklaration des Schemas keine besondere Erwähnung erfahren, da er als fester Wert verwendet wird. Der Kopf einer parametrisiertes CSP-OZ-DC-Klasse muss nach diesen Vorgaben der folgenden Form entsprechen:

$$NAME[\mathbb{M}](i : \mathbb{M})$$

Um die lokalen Kanäle eines Schemas für alle i eindeutig zu gestalten, wird ihnen i als Index angehängt.

Das parametrisierte PEA-Modell $\mathcal{A}(i) = (P_i, V_i, A_i, C_i, E_i, s_i, I_i, E_{0,i})$ entspricht in weiten Teilen dem ursprünglichen Modell ohne Parameter.

Es muss lediglich beachtet werden, dass die globalen Events des PEA korrekt mit dem Index i gekennzeichnet sind. Dies gewährleistet bei einer Instantiierung, dass die verschiedenen PEA eines parametrisierten PEA-Systems nur über die korrekten globalen Kanäle kommunizieren. Wie bereits bei den CSP-OZ-DC-Schemata wird der Parameter i innerhalb des PEA als fester Wert verwendet. Wird ein parametrisiertes PEA-System

[1] `http://syspect.informatik.uni-oldenburg.de/`

aus einem solchen PEA instantiiert, muss der Parameter für jeden Automaten korrekt gesetzt werden, so dass jedes i eindeutig ist.

Enthält ein PEA Kommunikationen über globale Kanäle, so können diese nur erfolgen, wenn die dafür benötigten PEA daran teilnehmen. Dies schließt auch den Fall mit ein, dass nicht alle PEA im parametrisierten PEA-System verfügbar sind. Wird also beispielsweise ein solches System mit einer Größe 5 instantiiert und eine Kommunikation innerhalb des Systems benötigt 6 verschiedene PEA, ist die Kommunikation nicht möglich.

Die hier betrachteten CSP-OZ-DC-Schemata besitzen keinen DC-Teil, da in der vorliegenden Arbeit nicht auf die Zeit eingegangen wird. Demnach haben parametrisierte PEA eine leere Uhren-Menge ($C_i = \varnothing$), was zu $\forall\, p \in P_i : I_i(p) = \text{true}$ führt. Außerdem gilt dadurch für jede Transition $(p, g, X, p') : X = \varnothing$. Aufgrund dieser Tatsache, werden Transitionen im weiteren Verlauf zur besseren Lesbarkeit durch (p, g, p') abgekürzt.

Weiterhin wird davon ausgegangen, dass die Zustandsinvarianten immer erfüllt werden. Dabei wird sich im späteren Verlauf noch zeigen, dass bei der Konstruktion eines PEA aus einem CSP-OZ-DC-Schema ohne DC-Teil die Zustandsinvariante für alle Zustände lediglich prüft, ob die genutzten Variablen innerhalb ihres Wertebereichs liegen.

Die in [Hoe06, Kapitel 5] gezeigten Methoden zur Übersetzung von CSP-OZ-DC zu PEA ist vollständig auf die Parametrisierung anwendbar, da keine essentiellen Veränderungen an CSP-OZ-DC durchgeführt wurden.

3.2 Transducer-Generierung

Das Problem der Transducer-Generierung wird darin zerlegt, verschiedene Teil-Transducer zu einem gegebenen parametrisierten PEA zu erzeugen. Der entsprechende Algorithmus soll hier zunächst beispielhaft anhand des in Abbildung 3.1 gezeigten PEA erläutert werden, um ein besseres Verständnis für seine Funktionsweise zu ermöglichen, bevor er dann formal aufgeführt wird. Der PEA selber erfüllt dabei keine nennenswerte Funktion. Die Kantenbedingungen $\mathit{only}(e)$ sagen aus, dass zum Ausführen der Transition lediglich das Event e auftreten darf. Dabei beschreibt $\mathit{only}(\tau)$ den Fall, dass kein Event eintritt.

Das Alphabet der von den Teil-Transducern akzeptierten Konfigurationswörter wird über allen Paaren von Zustandsnamen des behandelten PEA gebildet.

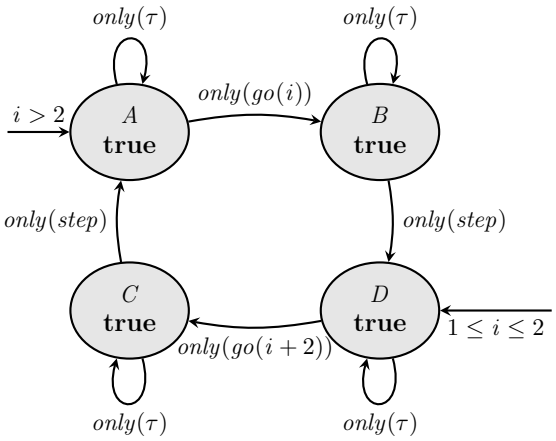
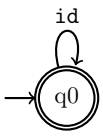
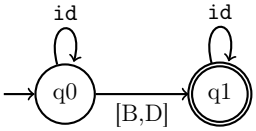


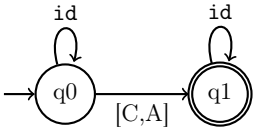
Abbildung 3.1: parametrisierter Phasen-Event-Automat $\mathcal{A}(i)$



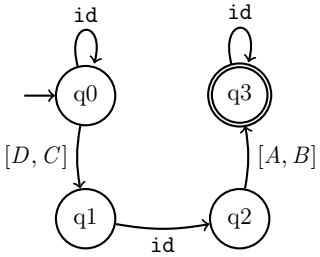
(a) Transducer $\mathcal{T}_1$



(b) Transducer $\mathcal{T}_2$



(c) Transducer $\mathcal{T}_3$



(d) Transducer $\mathcal{T}_4$

Abbildung 3.2: Transducer $\mathcal{T}_1 \ldots \mathcal{T}_4$ für $\mathcal{A}(i)$

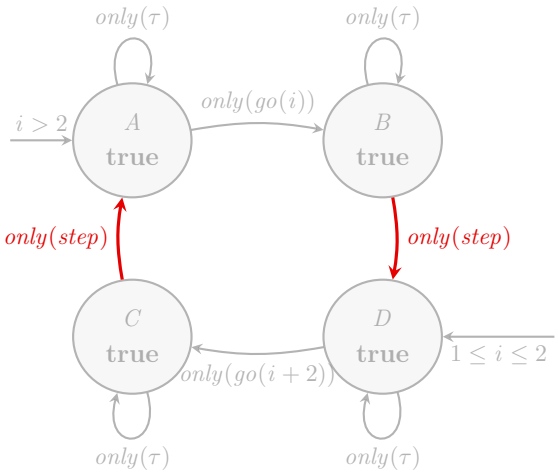
3.2.1 Beispiel

Gegeben sei der in Abbildung 3.1 dargestellte PEA. Ziel ist es nun, auf dieser Basis entsprechend die benötigten Transducer (vgl. Abb. 3.2(a)-(d)) zu erzeugen. Der Transducer aus Abbildung 3.2(a) ist dabei trivial. Er entspricht einem gemeinsamen Stotterschritt aller PEA-Instanzen, ohne dass eine Zustandsänderung eintritt. Da lediglich stotter-invariante PEA betrachtet werden, kann ein derartiger Transducer unmittelbar für jeden parametrisierten PEA erzeugt werden.

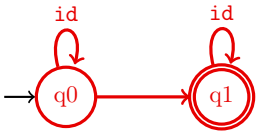
Ähnlich verläuft die Erzeugung der Transducer aus Abbildung 3.2(b) und 3.2(c). Diese

Transducer stellen jeweils einen Zustandsübergang mit einer internen Operation einer PEA-Instanz dar. Demnach müssen für deren Erzeugung zunächst sämtliche Transitionen bestimmt werden, in denen ein Zustandsübergang ohne Kommunikation erfolgt. Dies entspricht den Transitionen:

$$e_1^{step} = (B, only(step), D)$$
sowie
$$e_2^{step} = (C, only(step), A)$$

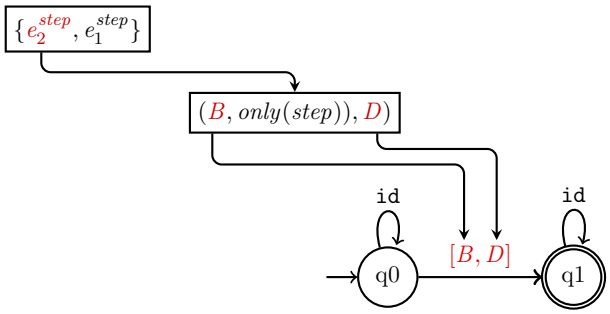


Für jedes Element, der auf diese Weise erzeugten Menge, wird ein Transducer der folgenden Form erzeugt. Dabei wird deutlich, dass der Übergang $q_0 \rightarrow q_1$ zunächst keine Kantenbedingung besitzt.



Diese Kantenbedingung entspricht dabei dem Paar bestehend aus Ausgangszustand p und Zielzustand p' des gewählten Elements.

Folgende Grafik zeigt die Erzeugung der Kantenbedingung für die Transition e_1^{step}.



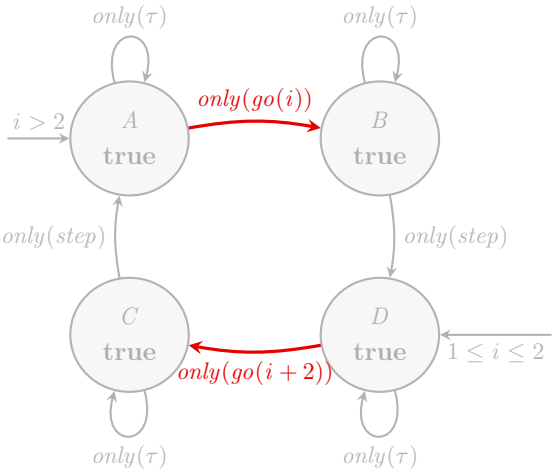
Die Erzeugung des Transducers für die Transition e_2^{step} erfolgt analog.

Die verbleibenden, noch nicht behandelten Transitionen des PEA entsprechen den Transitionen, in denen eine Kommunikation stattfindet. Für diese Transitionen muss ein

komplexerer Transducer erzeugt werden, da, anders als bisher, mehrere PEA gleichzeitig einen Operationsschritt ausführen.

Für unser Beispiel entspricht dies den folgenden Transitionen:

$$e_1^{go} = (A, only(go(i)), B)$$
sowie
$$e_2^{go} = (D, only(go(i+2)), C)$$



An dieser Stelle muss zunächst sichergestellt werden, dass es sich tatsächlich um *kommunizierende Transitionen* handelt. *Kommunizierende Transitionen* bedeutet, dass mehrere *unterschiedliche* PEA-Instanzen anhand der gefundenen Transitionen über den gleichen Kanal miteinander kommunizieren. Um dies zu gewährleisten, muss darauf geachtet werden, dass die Versätze der Parameter jeweils nur unterschiedliche Werte besitzen dürfen, da dies sonst der mehrfachen Teilnahme einer PEA-Instanz an derselben Kommunikation gleichkäme. Transitionen bei denen dies zutrifft, werden zu einer Menge zusammengefasst. Dabei kann es vorkommen, dass mehrere solcher Mengen entstehen. Dies sind beispielsweise verschiedene Mengen für verschiedene Kanäle oder aber verschiedene Mengen, die sich jeweils durch ein Element mit dem gleichen Versatz unterscheiden.

Vergleichen wir demnach aus unserem Beispiel den Versatz von e_1^{go} mit dem von e_2^{go}, stellen wir fest, dass diese ungleich sind ($0 \neq 2$). Demnach handelt es sich um miteinander kommunizierende Transitionen. Da die Kommunikation über den Kanal *go* erfolgt, bilden die Transitionen die Menge E'_{go}.

$$E'_{go} = \{e_1^{go}, e_2^{go}\}$$

Für jede auf diese Weise ermittelte Menge muss nun ein Teil-Transducer generiert werden. Da in unserem Fall lediglich die Menge E'_{go} existiert, ist das Ergebnis ein einziger Transducer.

Um aus einer solchen Menge E' einen Transducer zu konstruieren, müssen ihre Elemente bezüglich ihrer Position während einer Kommunikation sortiert vorliegen. Dies ist

deshalb von Bedeutung, da der Transducer-Algorithmus Kenntnis über die Reihenfolge der an der Kommunikation teilnehmenden PEA besitzen muss. Diese Kenntnis erlaubt es dem resultierenden Transducer, die zu prüfenden Konfigurationen korrekt von „links nach rechts“, also vom niederwertigsten Parameter aufsteigend, einzulesen.

Zur Sortierung werden die Elemente der Menge anhand ihres Versatzes absteigend sortiert. Auf diese Weise entsteht das geordnete Tupel $(e_1, e_2, \ldots, e_{n-1}, e_n)$ mit $e_1, \ldots, e_n \in E'$, welches folgende Bedingung erfüllt:

$$1 \leq i(e_1) < i(e_2) < \cdots < i(e_{n-1}) < i(e_n)$$

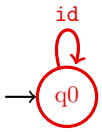
Für unser Beispiel entsteht das Tupel (e_2^{go}, e_1^{go}), da $k_{e_2} = 2 > 0 = k_{e_1}$ gilt.

Wurden die Mengen geordnet, kann für jede Menge ein Transducer konstruiert werden. Dies beginnt mit der Erzeugung eines Startzustandes. Für die weiteren Schritte sind darauf verschiedene Fallunterscheidungen notwendig.

So sind Szenarien denkbar, in denen in den Kantenbedingungen gefordert wird, dass nur ein PEA mit einem bestimmten Parameter (z.B. $i = 5$) oder aber mit einem Parameter innerhalb eines gewissen Bereichs (z.B. $i \leq 5$) diese Transition ausführen darf. Da beide Fälle nicht auf unser Beispiel zutreffen, werden sie an dieser Stelle zunächst nicht weiter betrachtet.

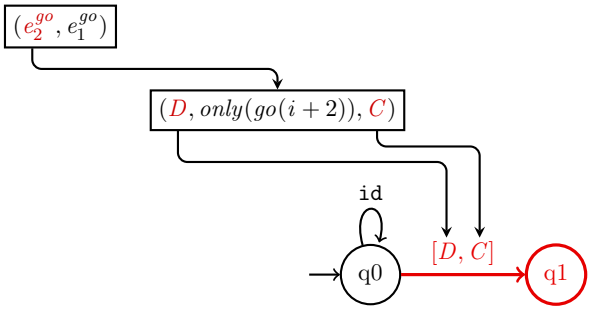
Wird also keinerlei Bedingung an den Parameter gestellt, kann sich der erste kommunizierenden PEA an einem beliebigen Punkt im Konfigurationswort befinden. Demnach muss der Transducer eine beliebige Anzahl an zustandswahrenden PEA-Operationen ermöglichen, bevor die Kommunikation erfolgt. Dies geschieht durch Hinzufügen einer Schleife, an den zu Beginn erzeugten Startzustand, und mit der Kantenbedingung `id`.

Anhand dieser Vorgehensweise entsteht für unser Beispiel folgender Transducer ohne Endzustand.



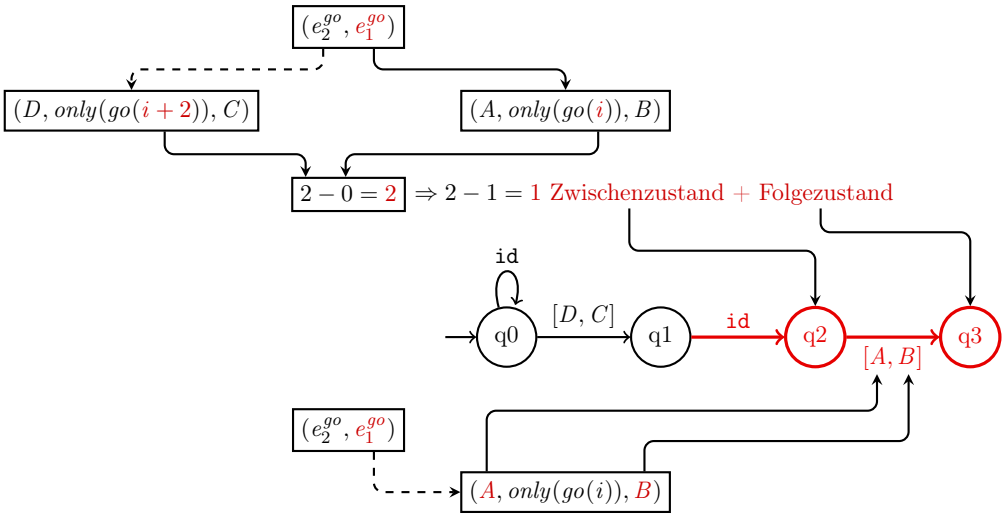
Wurde der Startzustand mit der Schleife erzeugt, wird dem Transducer ein weiterer Zustand, sowie eine Kante vom Startzustand zu diesem Zustand hinzugefügt. Anschließend wird das erste Element des vorher erzeugten Tupels gewählt und anhand dessen Ausgangszustand p und Zielzustandes p' die Kantenbedingung $[p, p']$ erzeugt.

Zur Verdeutlichung führen wir diesen Schritt an unserem Beispiel durch:



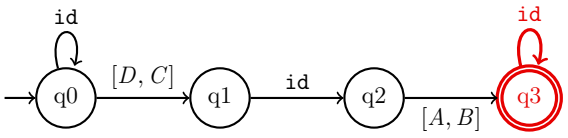
Der nächste Schritt besteht darin, die relative Position des nächsten an der Kommunikation teilnehmenden PEA zu ermitteln. Dazu wird das nächste Element aus dem Tupel gewählt und dessen Versatz von dem des vorigen Elements abgezogen. Sei n die Differenz, so müssen $n-1$ Zwischenzustände erzeugt und jeweils durch entsprechende Kanten mit dem zuletzt erzeugten Zustand verbunden werden. Dabei entsprechen die Kantenbedingungen den zustandswahrenden Operationen des PEA. Abschließend wird analog zum vorigen Konstruktionsschritt eine neue Kante sowie ein neuer Folgezustand generiert.

Die folgende Grafik soll diesen Schritt anhand unseres Beispiels genauer verdeutlichen. Die Verwendung eines vorher bereits genutzten Elements wird durch gestrichelte Linien gekennzeichnet:



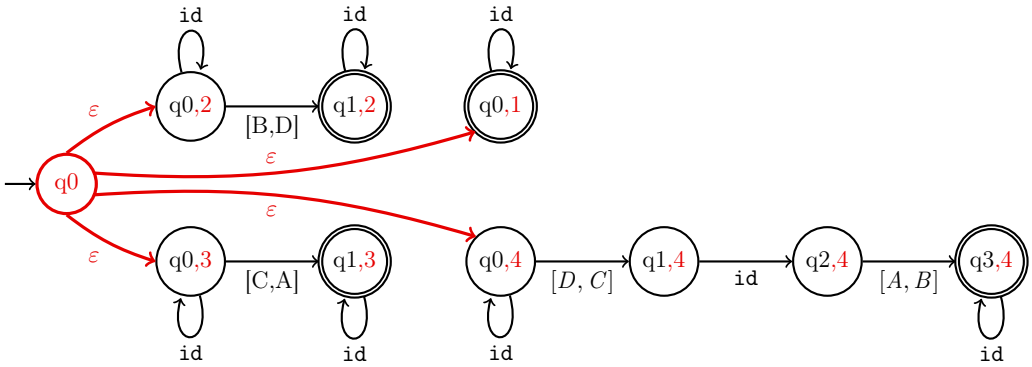
Der letzte auf diese Weise erzeugte Zustand wird Endzustand und erhält analog zum Startzustand eine Schleife mit den zustandswahrenden Operationen des parametrisierten PEA.

Somit erhalten wir für unser Beispiel folgenden Transducer:



Vergleichen wir diesen algorithmisch erzeugten Transducer mit Abbildung 3.2(d), ist auf diese Weise, bis auf die genaue Anordnung der Zustände und Kantenbedingungen, exakt der dort abgebildete Transducer entstanden.

Die so konstruierten Transducer können nun zu einem großen Step-Transducer zusammengefasst werden. Dazu wird ein neuer Startzustand erzeugt, und durch ausgehende ε-Kanten mit den Startzuständen der einzelnen Transducer verbunden. Die Zustände der einzelnen Transducer müssen vorher ggf. umbenannt werden, um doppelte Zustandsnamen zu vermeiden. Besagtes Kombinieren wird anhand folgender Grafik für unser Beispiel exemplarisch dargestellt.



Dies entspricht $\mathcal{L}(\mathcal{T}) = \bigcup_t \mathcal{L}(\mathcal{T}_t)$, der Vereinigung der Sprachen der Teil-Transducer $\mathcal{T}_t$.

3.2.2 Behandlung von Sonderfällen

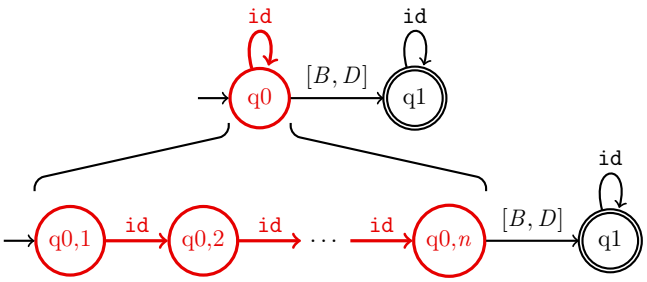
Wie in Abschnitt 3.2.1 bereits angesprochen, kann es vorkommen, dass die Kantenbedingungen eines PEA gewisse Anforderungen bezüglich des Parameters der entsprechenden PEA-Instanz stellen. Diese Sonderfälle spielen zunächst bei der Transducer-Generierung keine Rolle, bedürfen aber einer Anpassung der Ergebnisse. Im Folgenden werden mögliche Sonderfälle aufgeführt und jeweils anhand eines Beispiels erläutert. Dabei werden die im Abschnitt 3.2.1 erzeugten Transducer verwendet. Es ist zu beachten, dass die verwendeten Beispiele darauf nicht mehr den PEA aus Abbildung 3.1 repräsentieren.

Die jeweils behandelten Zustände werden farblich gekennzeichnet, wobei die grünen Zustände unverändert bleiben und die roten je nach Sonderfall auf eine Anzahl von

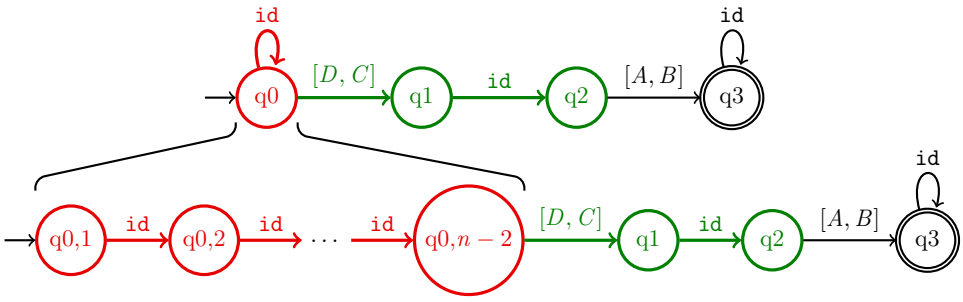
Zuständen verteilt werden. Die Bezeichnung der Fälle erfolgt durch die in der betrachteten Transition ermittelte Forderung an den Parameter.

Fall 1: $i = n \in \mathbb{N}$**:** Die Menge der Zustände des Transducers bis zu der Kante, die die angestrebte Transition des PEA repräsentiert, müssen auf n Zustände erweitert werden. Dazu werden mögliche Schleifen aufgelöst und der Schleifenzustand so oft auf neue Zustände verteilt, dass die angestrebte Anzahl erreicht wird. Um dies zu zeigen, soll die Vorgehensweise anhand von zwei Beispielen genauer dargestellt werden.

Sonderfall $i = n$ in Transition $(B, only(step), D)$:



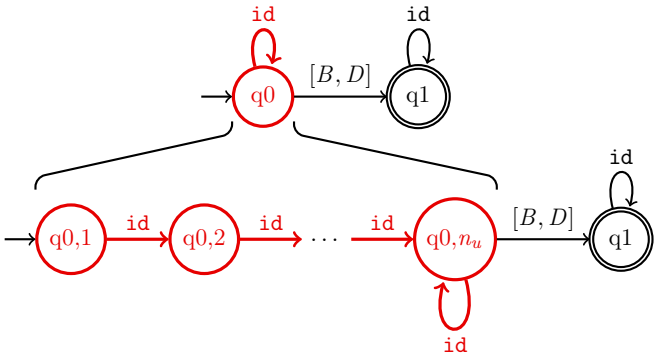
Sonderfall $i = n$ in Transition $(A, only(go(i)), B)$:



Fall 2: $n_u \leq i \leq n_o$ **mit** $n_u, n_o \in \mathbb{N} \wedge n_u \leq n_o$**:** Die Konstruktion des passenden Transducers entspricht der aus Fall 1, mit der Besonderheit, dass in diesem Fall mehrere Transducer erzeugt werden müssen. Dies geschieht analog zu Fall 1 für jede mögliche Ausprägung von i.

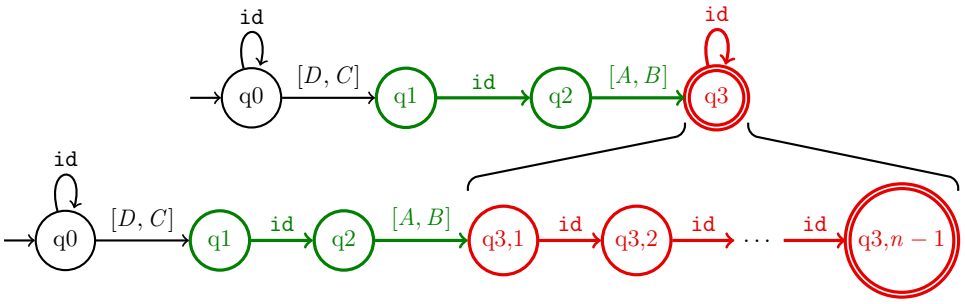
Fall 3: $n_u \leq i$ **mit** $n_u \in \mathbb{N}$**:** Die Anpassung des Transducers erfolgt analog zu Fall 1 für $n_u = i$. Abschließend jedoch wird dem letzten aus einem Schleifenzustand erstellten Zustand die ursprüngliche Schleife hinzugefügt.

Sonderfall $n_u \leq i$ in Transition $(B, only(step), D)$:



Fall 4: $i = M - n$ **mit** $n \in \mathbb{N}$**:** M ist die Anzahl der zur Laufzeit generierten parametrisierten PEA. Demnach entspricht M der Länge des Konfigurationswortes. Um den für diesen Fall benötigten Transducer zu generieren, müssen die Zustände des vorher erzeugten Transducers ab der Kante, die die angestrebte Transition des PEA repräsentiert, auf $n + 1$ Zustände verteilt werden. Dies geschieht dabei analog zu der Zustandsverteilung aus Fall 1. Der letzte auf diese Weise erzeugte Zustand wird Endzustand.

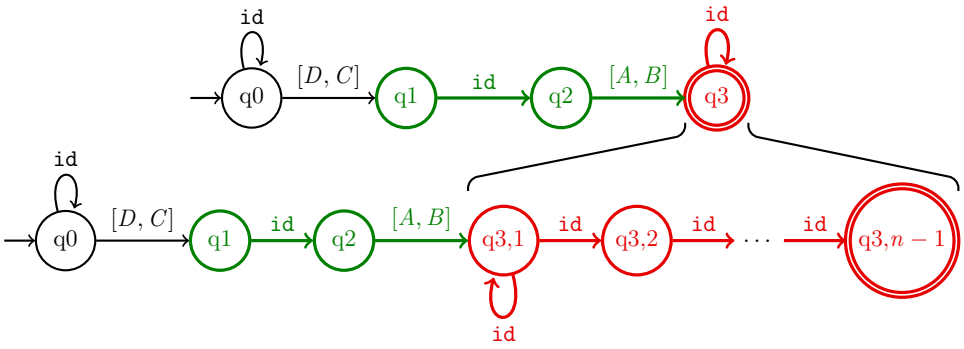
Sonderfall $i = M - n$ in Transition $(D, only(go(i + 2)), C)$:



Fall 5: $M - n_u \leq i \leq M - n_o$ **mit** $n_u, n_o \in \mathbb{N} \wedge n_u \geq n_o$**:** Vergleichbar mit Fall 2, müssen in diesem Fall mehrere Transducer erzeugt werden. Dies geschieht für jede mögliche Ausprägung von i auf die gleiche Weise wie in Fall 4.

Fall 6: $i \leq M - n_o$ **mit** $n_o \in \mathbb{N}$**:** Die Anpassung des Transducers erfolgt analog zu Fall 4 für $n_o = i$. Dem ersten aus einem Schleifenzustand erstellten Zustand muss jedoch die ursprüngliche Schleife hinzugefügt werden.

Sonderfall $i \leq M - n_o$ in Transition $(D, only(go(i + 2)), C)$:



3.2.3 Transducer-Algorithmus

Der folgende formale Algorithmus dient dazu, zu einem gegebenen parametrisierten Phasen-Event-Automaten $\mathcal{A}(i) = (P_i, V_i, A_i, C_i, E_i, s_i, I_i, E_{0,i})$ einen Transducer zu generieren.

Die Vorgehensweise des Algorithmus wurde bereits in Abschnitt 3.2.1 erläutert. Zum besseren Verständnis wurden ihm jedoch zusätzlich erläuternde Textbausteine eingefügt.

Algorithmus:

Erzeuge leere Menge für Transducer:

$Z_{\mathcal{T}} = \varnothing$

Zähler für Transducer:

$z = 1$

Erzeuge `id`*-Transducer:*

$\mathcal{T}_z = (\Sigma_{\mathcal{T}_z}, Q_{\mathcal{T}_z}, E_{\mathcal{T}_z}, q_0, F_{\mathcal{T}_z})$

$\Sigma_{\mathcal{T}_z} = \{\texttt{id}\}$

$Q_{\mathcal{T}_z} = \{q_0\}$

$E_{\mathcal{T}_z} = \{(q_0, \texttt{id}, q0)\}$

$F_{\mathcal{T}_z} = \{q_0\}$

$Z_{\mathcal{T}} = Z_{\mathcal{T}} \cup \{\mathcal{T}_z\}$

$z = z + 1$

Finde Kanten mit Parameter i*:*

$E'_c = \{e \in E \mid g(e) \text{ enthält } c.i(e) + k_e, k_e \in \mathbb{Z}\}$

Bilde Menge aus allen nicht kommunizierenden Transitionen:

$E' = E \setminus E'_c$

Erzeuge Transducer ohne Kommunikation:

`foreach` $e \in E'$ `do`

$\mathcal{T}_z = (\Sigma_{\mathcal{T}_z}, Q_{\mathcal{T}_z}, E_{\mathcal{T}_z}, q_0, F_{\mathcal{T}_z})$
$\Sigma_{\mathcal{T}_z} = \{\texttt{id}, [p(e), p'(e)]\}$
$Q_{\mathcal{T}_z} = \{q_0, q_1\}$
$E_{\mathcal{T}_z} = \{(q_0, \texttt{id}, q_0), (q_0, [p(e), p'(e)], q_1), (q_1, \texttt{id}, q_1)\}$
$F_{\mathcal{T}_z} = \{q_1\}$
$Z_{\mathcal{T}} = Z_{\mathcal{T}} \cup \{\mathcal{T}_z\}$
$z = z + 1$

`od`

Bilde Menge aus größten Teilmengen kommunizierender Transitionen:

$S = \mathcal{P}(E'_c), R = \varnothing$
`foreach` $s \in S$ mit $|s| \geq 2$ `do`
`if` $\forall\, e, e' \in s, e \neq e' : k_e \neq k_{e'}$ `then`
$R = (R \setminus \{r \in R \mid r \subseteq s\}) \cup \{s\}$
`fi`
`od`

Sortiere Transitionen anhand ihres Versatzes:

`foreach` $r \in R$ `do`
$n = |r|$
ordne r anhand von k_e so dass gilt:
$(e_{l_1}, e_{l_2}, \ldots, e_{l_n}) : 1 \leq i(e_{l_1}) < i(e_{l_2}) < \cdots < i(e_{l_n})$

Generiere Transducer anhand geordneten Tupels:

$\mathcal{T}_z = (\Sigma_{\mathcal{T}_z}, Q_{\mathcal{T}_z}, E_{\mathcal{T}_z}, q_0, F_{\mathcal{T}_z})$
$\Sigma_{\mathcal{T}_z} = \{\texttt{id}, [p(e_{l_1}), p'(e_{l_1})]\}$
$Q_{\mathcal{T}_z} = \{q_0, q_1\}$

Erzeuge Kante für Transition e_{l_1}:

$E_{\mathcal{T}_z} = \{(q_0, \texttt{id}, q_0), (q_0, [p(e_{l_1}), p'(e_{l_1})], q_1)\}$
$F_{\mathcal{T}_z} = \varnothing$
$m = 1$

Behandle alle Transitionen ab e_{l_2}:

`for` $j = 2 \ldots n$ `do`

Erzeugen von Zwischenzuständen anhand der Versatzdifferenz:

`for` $p = 1 \ldots (k_e(e_{l_{j-1}}) - k_e(e_{l_j})) - 1$ `do`
$Q_{\mathcal{T}_z} = Q_{\mathcal{T}_z} \cup \{q_{m+1}\}$
$E_{\mathcal{T}_z} = E_{\mathcal{T}_z} \cup \{(q_m, \texttt{id}, q_{m+1})\}$
$m = m + 1$
`od`

$Q_{\mathcal{T}_z} = Q_{\mathcal{T}_z} \cup \{q_{m+1}\}$

Erzeuge Kante für Transition e_{l_j}:

$E_{\mathcal{T}_z} = E_{\mathcal{T}_z} \cup \{(q_m, [p(e_{l_j}), p'(e_{l_j})], q_{m+1})\}$

$\Sigma_{\mathcal{T}_z} = \Sigma_{\mathcal{T}_z} \cup \{[p(e_{l_j}), p'(e_{l_j})]\}$

$m = m + 1$

`od`

$E_{\mathcal{T}_z} = E_{\mathcal{T}_z} \cup \{(q_m, \texttt{id}, q_m)\}$

$F_{\mathcal{T}_z} = \{q_m\}$

$Z_{\mathcal{T}} = Z_{\mathcal{T}} \cup \{\mathcal{T}_z\}$

$z = z + 1$

`od`

Behandlung von Sonderfällen:

Sonderfall (1,...,6) in Transition e:

Wähle den Transducer $\mathcal{T}_z \in Z_{\mathcal{T}}$ mit:

$[p(e), p'(e)] \in \Sigma_{\mathcal{T}_z}$

Bilde Zustandsmenge Q':

Im Folgenden kürzen wir $\exists\, a \in \Sigma_{\mathcal{T}_z} : (q_1, a, q_2) \in E_{\mathcal{T}_z}$ für zwei Zustände $q_1, q_2 \in Q_{\mathcal{T}_z}$ durch $q_1 \to q_2$ ab.

Die reflexive, transitive Hülle von $\to$ ist $\to^*$.

Fall 1-3:

Wähle Zustand q_x, für den gilt:

$(q_x, [p(e), p'(e)], q') \in E_{\mathcal{T}_z}, q_x, q' \in Q_{\mathcal{T}_z}$

$Q' = \{q \in Q_{\mathcal{T}_z} \mid q \to^* q_x\}$

Fall 4-6:

Wähle Zustand q_x, für den gilt:

$(q', [p(e), p'(e)], q_x) \in E_{\mathcal{T}_z}, q', q_x \in Q_{\mathcal{T}_z}$

$Q' = \{q \in Q_{\mathcal{T}_z} \mid q_x \to^* q\}$

Transducer $\mathcal{T}_z$-Anpassung:

Fall 1:

$\mathcal{T}_z = \texttt{adjustTransducer}(Q', \mathcal{T}_z, n, \text{false})$

Fall 2:

`for` $v = n_u \ldots n_o$ `do`

$\mathcal{T}_{z,v} = \texttt{adjustTransducer}(Q', \mathcal{T}_z, v, \text{false})$

$Z_{\mathcal{T}} = Z_{\mathcal{T}} \cup \mathcal{T}_{z,v}$

`od`

$Z_{\mathcal{T}} = Z_{\mathcal{T}} \setminus \mathcal{T}_z$

Fall 3:

$\mathcal{T}_z =$ `adjustTransducer(`$Q', \mathcal{T}_z, n, \text{true}$`)`

Fall 4:

$\mathcal{T}_z =$ `adjustTransducer(`$Q', \mathcal{T}_z, n+1, \text{false}$`)`

Fall 5:

`for` $v = n_o \ldots n_u$ `do`

$\mathcal{T}_{z,v} =$ `adjustTransducer(`$Q', \mathcal{T}_z, v+1, \text{false}$`)`

$Z_{\mathcal{T}} = Z_{\mathcal{T}} \cup \mathcal{T}_{z,v}$

`od`

$Z_{\mathcal{T}} = Z_{\mathcal{T}} \setminus \mathcal{T}_z$

Fall 6:

$\mathcal{T}_z =$ `adjustTransducer(`$Q', \mathcal{T}_z, n+1, \text{true}$`)`

Erzeuge großen Step-Transducer:

$\mathcal{T}$ = mengenmäßige Vereinigung aller Transducer aus $Z_{\mathcal{T}}$

Funktion `adjustTransducer()`:

Zur besseren Lesbarkeit wird die Funktion `adjustTransducer()` gesondert aufgeführt. Bei den übergebenen Parametern handelt es sich um Wertparameter. Demnach führen Änderungen an den übergebenen Elementen nicht zu Änderungen der Elemente, mit denen die Funktion aufgerufen wurde.

`function adjustTransducer(`Q : *Zustandsmenge*, $\mathcal{T}$: *Transducer*, $n : \mathbb{N}_1$, *loop* : $\mathbb{B}$`)`

`returns:` $\mathcal{T}$: *Transducer*

Behandle Schleifenzustand und entferne Schleife:

Wähle Zustand $q_s \in Q$, für den gilt:

$(q_s, \texttt{id}, q_s) \in E_{\mathcal{T}}$

$E_{\mathcal{T}} = E_{\mathcal{T}} \setminus \{(q_s, \texttt{id}, q_s)\}$

Prüfe, ob Zustandsmenge vergrößert werden muss:

`if` $n > |Q|$ `then`

$Q_{\mathcal{T}} = Q_{\mathcal{T}} \cup \{q_{s,1}\}$

$y = 2$

`for` $o = 1 \ldots n - |Q|$ `do`

$Q_{\mathcal{T}} = Q_{\mathcal{T}} \cup \{q_{s,y}\}$

$E_{\mathcal{T}} = E_{\mathcal{T}} \cup \{(q_{s,y-1}, id, q_{s,y})\}$

$y = y + 1$

`od`

Wenn q_s Endzustand, mache $q_{s,y}$ zum Endzustand:

`if` $q_s \in F_\mathcal{T}$ `then`

$F_\mathcal{T} = F_\mathcal{T} \cup \{q_{s,y}\}$

Setzen der Schleife für Fall 6:

`if` *loop* = true `then`

$E_\mathcal{T} = E_\mathcal{T} \cup \{(q_{s,1}, \texttt{id}, q_{s,1})\}$

`fi`

`fi`

Wenn q_s Startzustand, mache $q_{s,y}$ zum Startzustand:

`if` $q_s = q_{0_z}$ `then`

$q_{0_z} = q_{s,1}$

Setzen der Schleife für Fall 3:

`if` *loop* = true `then`

$E_\mathcal{T} = E_\mathcal{T} \cup \{(q_{s,y}, \texttt{id}, q_{s,y})\}$

`fi`

`fi`

Finde Nachfolger q'_s von q_s und füge Ausgangskante von $q_{s,y}$ nach q'_s ein:

Wähle Zustand $q'_s \in Q_\mathcal{T}$, für den gilt:

$\exists\, a \in \Sigma_\mathcal{T} : (q_s, a, q'_s) \in E_\mathcal{T}$

$E_\mathcal{T} = E_\mathcal{T} \cup \{(q_{s,y}, a, q'_s)\}$

$E_\mathcal{T} = E_\mathcal{T} \setminus \{(q_s, a, q'_s)\}$

Finde Vorgänger q''_s von q_s und füge Eingangskante von q''_s nach $q_{s,1}$ ein:

Wähle Zustand $q''_s \in Q_\mathcal{T}$, für den gilt:

$\exists\, a \in \Sigma_\mathcal{T} : (q''_s, a, q_s) \in E_\mathcal{T}$

$E_\mathcal{T} = E_\mathcal{T} \cup \{(q''_s, a, q_{s,1})\}$

$E_\mathcal{T} = E_\mathcal{T} \setminus \{(q''_s, a, q_s)\}$

Entferne q_s:

$Q_\mathcal{T} = Q_\mathcal{T} \setminus \{q_s\}$

`fi`

`return` $\mathcal{T}$

3.2.4 Korrektheit des Algorithmus

Ein formaler Beweis, dass der durch den hier vorgestellten Algorithmus erzeugte Transducer tatsächlich für zwei Konfigurationen entscheidet, ob sie aufeinander folgen können,

bleibt zunächst aus. Um für die korrekte Funktionsweise jedoch verbal zu argumentieren, werden die Teil-Transducer zunächst in verschiedene Klassen eingeteilt. Dabei wird im groben zwischen drei Hauptklassen unterschieden:

1. `id`-Transducer (vgl. Abb. 3.2(a))
2. Transducer ohne Kommunikation (vgl. Abb. 3.2(b) und Abb. 3.2(c))
3. Transducer mit Kommunikation (vgl. Abb. 3.2(d))

Zusätzlich dazu können angepasste Versionen der Transducer der letzten beiden Klassen durch die in Abschnitt 3.2.2 aufgezeigten Sonderfälle auftreten.

Anhand dieser Klassen wird im Folgenden begründet, dass das zu den jeweiligen Transducern gehörende parametrisierte PEA-System die geforderten Schritte auszuführen vermag, sowie, dass jeder Schritt innerhalb dieses parametrisierten PEA-Systems von den Transducern nachvollzogen werden kann.

Wort der Transducer-Sprache entspricht Schritt im parametrisierten PEA-System

Die Sprache der durch den hier beschriebenen Algorithmus generierten Transducer entsteht durch Bearbeitung sämtlicher Transitionen eines parametrisierten PEA. Wird aus diesem PEA ein parametrisiertes PEA-System erzeugt, muss demnach jeder der in den Kantenbedingungen dieser Transducer geforderten Schritte innerhalb eines jeden PEA generell möglich sein. Lediglich die Möglichkeit der parallelen Ausführung der in jeweils einem der Transducer geforderten Schritte muss gewährleistet sein.

Jede der drei genannten Transducer-Klassen enthält `id`-Kanten. Da diese den Transitionen eines parametrisierten PEA entsprechen, durch die der PEA seinen Zustand nicht ändert und auch seine Umwelt nicht beeinflusst, wird durch sie die Parallelität nicht eingeschränkt. Dadurch sind sämtliche `id`-Transducer problemlos anwendbar.

Die zweite Transducer-Klasse enthält, zusätzlich zu den `id`-Kanten, eine Kante ohne Kommunikation. Da es sich dabei ebenfalls um Transitionen des PEA handelt, durch die der PEA seine Umwelt nicht beeinflusst, bleibt durch diese die parallele Ausführung der PEA ebenso unverletzt. Demnach sind Transducer dieser Klasse ohne Probleme anzuwenden.

Transducer der dritten Klasse enthalten Kanten mit Kommunikation. Diese entsprechen verschiedenen Transitionen, die von mehreren PEA parallel genommen werden müssen. Die Position der teilnehmenden PEA innerhalb des parametrisierten PEA-Systems wird durch die eingefügten `id`-Kanten mit Zwischenzuständen bestimmt. Da die Anzahl

dieser `id`-Kanten und Zwischenzustände durch die relativen Parameterangaben der kommunizierenden Transitionen des PEA bestimmt wurden, ist ein gemeinsamer Schritt der an einer Kommunikation teilnehmenden PEA gewährleistet.

Auch die Sonderfälle werden durch die Parallelität nicht eingeschränkt, da die entsprechenden Transducer aus den hier bereits genannten generiert werden. Es muss lediglich sichergestellt sein, dass jeder vom Transducer gewünschte PEA auch in der Lage ist, die entsprechende Transition zu nehmen. Da der Transducer diese Information wiederum bei seiner Generierung aus exakt dieser Transition bezieht, ist auch dies gewährleistet.

Ein Schritt im parametrisierten PEA-System liegt in Transducer-Sprache

Anhand der Argumentation für die Gegenrichtung wurde gezeigt, dass die erzeugten Transducer das Verhalten der PEA innerhalb eines parametrisierten PEA-Systems korrekt darstellen, sofern nur ein Schritt eines PEA erfolgt, bzw. eine Kommunikation mehrerer PEA. Diese Argumentation trifft vollständig auch auf die nun behandelte Richtung zu. Ein Problem scheint zunächst durch Szenarien dargestellt zu werden, in denen zwei oder mehr PEA innerhalb eines parametrisierten PEA-Systems einen Schritt ohne Kommunikation zu tätigen versuchen, oder aber mehrere verschiedene Kommunikationen parallel stattfinden sollen. Die Art der Nutzung der generierten Transducer löst jedoch dieses Problem, da durch die Bildung von $\mathcal{T}^*$ des kombinierten Step-Transducers die Information über Zwischenschritte verloren geht.

So ist es den Transducern zwar nicht möglich, jeden verfügbaren Schritt eines parametrisierten PEA-Systems nachzuvollziehen, jedoch kann jeder Schritt anhand mehrerer Transducer-Durchläufe modelliert werden. Dies ist dadurch gewährleistet, dass ein PEA keine zwei Transitionen innerhalb eines gemeinsamen Operationsschritts nehmen kann und dass Transducer für jeden möglichen einzelnen Schritt verfügbar sind.

Die folgende Grafik in Abbildung 3.3 demonstriert dies anhand eines Beispiels. Dabei wird erneut auf den Beispiel-PEA aus Abbildung 3.1 und die dafür generierten Transducer $\mathcal{T}_1$ bis $\mathcal{T}_4$ zurückgegriffen. Die linke Seite zeigt einen synchronen Schritt eines anhand dieses PEA instantiierten parametrisierten PEA-Systems, während auf der rechten Seite die Einzelschritte durch die Transducer nachvollzogen werden. Die unmarkierten PEA führen Stotterkanten aus.

3.2.5 Begründung der Designentscheidung

In verschiedenen Phasen während der Entstehung des Transducer-Algorithmus wurde zunächst versucht, komplexere Transducer zu erzeugen, um der später notwendigen Ver-

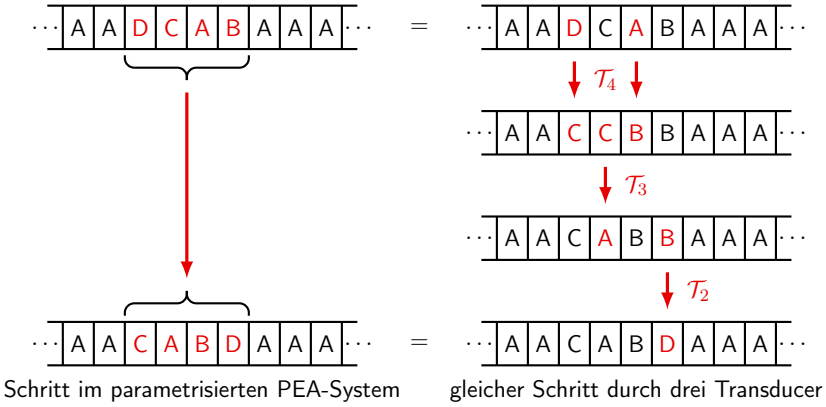


Abbildung 3.3: Demonstration eines synchronen Schritts in einem PEA-System und die dazugehörende Transducer-Repräsentation

einigung der Teil-Transducer und der dadurch möglicherweise benötigten Optimierung und Minimierung vorzubeugen. In den meisten Fällen führte dies dabei lediglich zu weiteren Kanten zwischen den Zuständen.

So wurden beispielsweise jedem Transducer zu den Punkten, an denen beim vorgestellten Algorithmus bloß `id`-Kanten auftreten, zusätzlich jeder Zustandsübergang des PEA ohne Kommunikation hinzugefügt. Obwohl dies bei der Erstellung des Transducers keine Schwierigkeit dargestellt hätte, wurde sich jedoch dagegen entschieden, da dies die Sonderfall-Behandlung unnötig erschwert hätte.

Der zunächst offensichtliche Vorteil der einfacheren Transducer besteht in der besseren Lesbarkeit, da sich jeder Einzelschritt und jede Kommunikation direkt in einem Teil-Transducer wiederfindet.

Es hat sich zudem gezeigt, dass die Behandlung der Sonderfälle aus Abschnitt 3.2.2 durch die komplexeren Transducer unnötig aufwendig gestaltet hätte, da in diesem Fall die Kanten, welche die Transition mit dem entsprechenden Sonderfall repräsentierten, zunächst aus sämtlichen bestehenden Transducern entfernt und ein neuer Transducer erstellt werden musste.

Eine weitere Überlegung bestand darin, sofort einen allumfassenden Step-Transducer zu generieren, der sämtliche möglichen parallelen Schritte eines parametrisierten PEA-System korrekt modelliert. Die Konstruktion eines solchen Transducers erwies sich jedoch, speziell bei der Behandlung von Transitionen mit Kommunikation, als sehr komplex, da viele Nebenbedingungen beachtet werden mussten. Da ein solcher Transducer für das reguläre Model-Checking durch die Bildung von $\mathcal{T}^*$ nicht zwingend benötigt wird, wurde dessen Konstruktion nicht weiter verfolgt.

4 Implementierung

Ausgangsbasis für die Implementierung der Transducer und des Transducer-Algorithmus ist das in der Abteilung *Correct Systems Design* der Carl von Ossietzky Universität Oldenburg entwickelte PEA-Toolkit[1]. Darin werden PEA-Datentypen bereitgestellt, sowie verschiedene Operationen darauf. Die Transducer-Klasse, sowie die damit verbundenen Algorithmen, bilden ein Unterpaket dieses Werkzeugs.

Um einen Transducer aus einem parametrisierten PEA konstruieren zu können, musste das PEA-Toolkit um ein parametrisiertes PEA-Modell erweitert werden.

Im Folgenden werden die dabei entstandenen Klassen aufgeführt und deren Funktionsweise und Methoden genauer erläutert. Dabei wird paketweise vorgegangen. Das Paket `pea.parameterized` beinhaltet dabei das angepasste PEA-Klassenmodell, wohingegen sich das Transducer-Modell in dem Paket `pea.modelchecking.regular` wiederfindet.

4.1 `pea.parameterized`

Das Paket `pea.parameterized` besteht aus den in Abbildung 4.1 dargestellten Klassen `IParameterized`, `Parameter` und `SingleParamPEA`.

4.1.1 `IParameterized`

Das Interface `IParameterized` dient der Parametrisierung. Die von ihm bereitgestellten Methoden ermöglichen das Hinzufügen, Auslesen und einmalige Setzen von Parametern. Zwar werden in dieser Arbeit lediglich einfach parametrisierte PEA behandelt, jedoch sollte eine Grundlage geschaffen werden, um PEA auch durch verschiedene Parameter zu parametrisieren.

Methoden:

`+addParameter (String name) : boolean`

Fügt einen Parameter hinzu mit `name` als Bezeichnung, sofern nicht bereits ein

[1] `http://csd.informatik.uni-oldenburg.de/projects/pea.html`

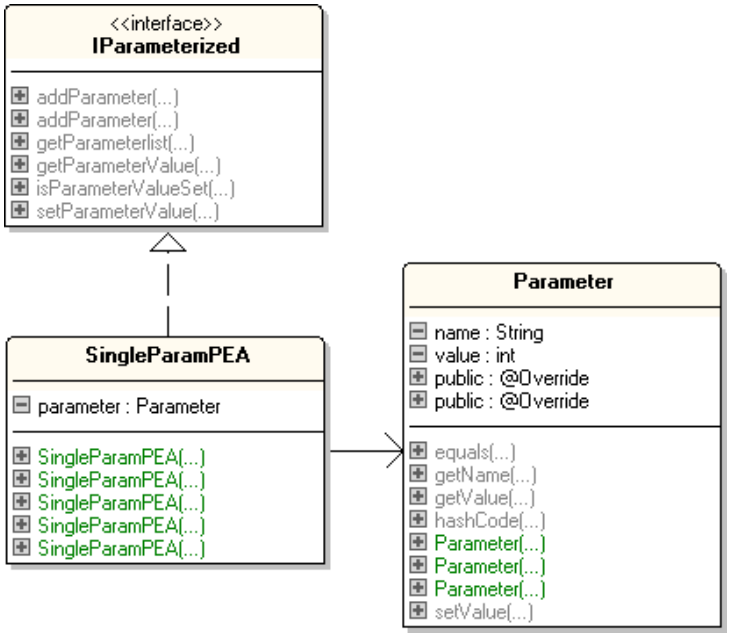


Abbildung 4.1: Klassen des Pakets `pea.parameterized`

solcher Parameter existiert. War das Hinzufügen erfolgreich, wird `true` zurückgeliefert.

`+addParameter (String name, int value) : boolean`
: Gleiche Funktionsweise wie oben, mit der Ausnahme, dass der hinzugefügte Parameter bereits mit `value` initialisiert wird.

`+getParameterlist () : String`
: Gibt ein Array der Bezeichnungen der bestehenden Parameter zurück.

`+getParameterValue (String name) : int`
: Existiert ein Parameter mit der Bezeichnung `name`, wird der Wert dieses Parameters zurückgegeben. Existiert kein derartiger Parameter oder wurde ihm noch kein Wert zugewiesen, wird -1 zurückgeliefert.

`+isParameterValueSet (String name) : boolean`
: Überprüft, ob dem durch `name` gekennzeichneten Parameter bereits ein Wert zugewiesen wurde. Sofern dies nicht der Fall ist, oder aber der gewünschte Parameter nicht existiert, wird `false` zurückgeliefert.

`+setParameterValue (String name, int value) : boolean`
: Setzt den Wert des mit `name` gekennzeichneten Parameters auf `value`, falls ein sol-

cher existiert. Da ein Parameter in der Praxis nur einmal einen Wert zugewiesen bekommen soll, wird `false` zurückgeliefert, wenn versucht wird, einen bereits bestehenden Wert zu verändern. Außerdem liefert die Methode `false` zurück, wenn kein geeigneter Parameter existiert.

4.1.2 SingleParamPEA

`SingleParamPEA` ist eine Unterklasse von `PhaseEventAutomata`, der dem PEA-Toolkit beiliegenden Implementierung des PEA-Modells. Es implementiert die abstrakten Methoden des `IParameterized`-Interfaces.

`SingleParamPEA` besitzt nur einen Parameter und stellt somit das in Kapitel 3 verwendete einfach parametrisierte PEA-Model dar. Die Implementierung der `IParameterized`-Methoden richtet sich nach dieser Einschränkung. Wird beispielsweise versucht, einem `SingleParamPEA`-Objekt einen weiteren Parameter hinzuzufügen, schlägt dies fehl.

Die von `PhaseEventAutomata` geerbten Methoden bleiben unangetastet. Ebenso werden sämtliche Konstruktoren übernommen. Dabei wird in jedem Fall der entsprechende Super-Konstruktor aufgerufen und der Parameter erzeugt.

Konstruktoren:

`+SingleParamPEA (PhaseEventAutomata pea, Parameter param)`

Ein Konstruktor, der das `PhaseEventAutomata`-Objekt `pea` übergeben bekommt und daraus ein `SingleParamPEA`-Objekt erzeugt. Der Parameter wird dabei durch das übergebene `Parameter`-Objekt `param` bestimmt. `Parameter` wird weiter unten gesondert aufgeführt und genauer beschrieben.

`+SingleParamPEA (PhaseEventAutomata pea, String paramName)`

Ein Konstruktor, der analog zum vorigen ein `PhaseEventAutomata`-Objekt übergeben bekommt und daraus ein `SingleParamPEA`-Objekt erzeugt. In diesem Fall wird jedoch lediglich die Bezeichnung des Parameters übergeben. Damit wird ein neues `Parameter`-Objekt generiert mit der Bezeichnung `paramName` generiert.

Methoden:

`+getParameterValue () : int`

Gibt den Wert des Parameters zurück. Sollte dieser noch nicht gesetzt worden sein, wird -1 zurückgeliefert.

`+isParameterValueSet () : boolean`

Überprüft, ob der Wert des Parameters bereits gesetzt wurde. Liefert `true` zurück, wenn dies der Fall ist.

4.1.3 `Parameter`

`Parameter` stellt den verwendeten Datentyp eines Parameters dar. Er hat zwei private Attribute. `String name` charakterisiert die Bezeichnung, `int value` den Wert des Parameters. Die Bezeichnung wird bei Instantiierung bestimmt. Der Wert dagegen, kann jederzeit auf einen Wert größer als 0 gesetzt werden. Wurde dies jedoch einmal getan, kann er nicht mehr geändert werden.

Konstruktoren:

`+Parameter ()`

Standard-Konstruktor, der ein `Parameter`-Objekt mit dem String `"i"` als Bezeichnung erzeugt. Der Wert des Parameters wird auf -1 gesetzt. Dies soll kennzeichnen, dass für den Parameter noch kein Wert festgelegt wurde.

`+Parameter (String name)`

Ein Konstruktor, der ein `Parameter`-Objekt mit der durch den `name` festgelegten Bezeichnung erzeugt. Wie beim Standard-Konstruktor wird der Wert des Parameters auf -1 gesetzt.

`+Parameter (String name, int value)`

Ein Konstruktor mit der eben genannten Funktionsweise. Jedoch wird der Wert des Parameters auf `value` gesetzt, sofern es sich dabei um eine positive, natürliche Zahl handelt.

Methoden:

`+setValue (int value)`

Setter-Methode für den Wert des Parameters. Dabei wird überprüft, ob es sich bei `value` um eine positive, natürliche Zahl handelt und ob der Wert des Parameters nicht bereits gesetzt wurde. Treffen beide Bedingungen zu, wird der Wert auf den von `value` gesetzt und `true` zurückgeliefert. Sobald eine der Bedingungen nicht zutrifft, gibt die Methode `false` zurück.

Bei den verbleibenden Methoden `+getName() : String` und `+getValue() : int` handelt es sich um Standard Getter-Methoden. `equals(Object) : boolean` und `hash-`

`Code() : int` sind geerbte Methoden von `Object` und sind so implementiert, dass zwei `Parameter`-Objekte als gleich gelten, sofern ihre Parameter-Bezeichnung identisch ist.

4.2 pea.modelchecking.regular

Das Paket `pea.modelchecking.regular` besteht aus den in Abbildung 4.2 dargestellten Klassen. Diese stellen zugleich den Datentyp eines Transducers dar.

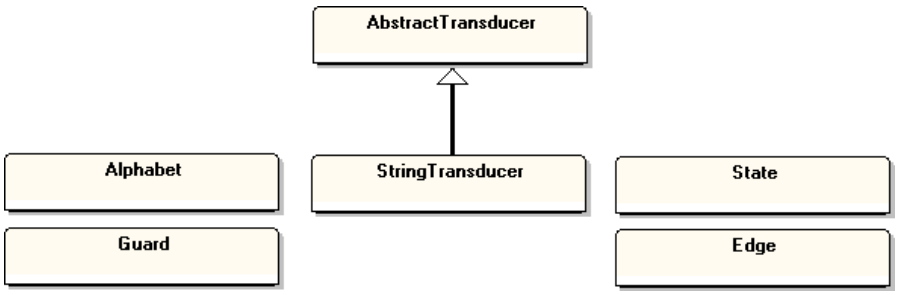


Abbildung 4.2: Klassen des Pakets `pea.modelchecking.regular`

Zum besseren Verständnis der Klassen `AbstractTransducer` und `StringTransducer` werden zunächst die Klassen `Guard`, `Alphabet`, `State` sowie `Edge` aufgeführt und beschrieben.

4.2.1 Guard

`Guard` stellt gleichermaßen Kantenbedingungen und Symbole des Alphabets eines Transducers dar. Da ein Transducer als Eingabealphabet Paare von Symbolen erwartet, besitzt die Guard Klasse die Attribute `wordA` und `wordB` vom generischen Datentyp `V`. Dieser wurde gewählt, um eine größtmögliche Wiederverwendbarkeit des Datenmodells zu gewährleisten. Zwar würde es für unsere Zwecke zunächst genügen, wenn `wordA` und `wordB` `String`-Objekte wären, so ist es jedoch denkbar, dass eine derartige Einschränkung für zukünftige Anpassungen unzureichend wäre.

Außerdem enthält `Guard` die beiden Konstanten `EPSILON` und `IDENTITY`, um eine Verwendung dieser besonderen Symbole bei der Erstellung von Transducern zu gewährleisten. Der Datentyp dieser Konstanten besteht dabei jeweils aus einer anonymen Unterklasse von `Guard`.

Die Methoden von `Guard` bestehen aus Getter-Methoden für die Attribute und Konstanten, sowie einem Konstruktor, der als Eingabe zwei Objekte vom Typ `V` entgegennimmt und damit `wordA` und `wordB` initialisiert. Außerdem wurden die von `Objekt` ge-

erbten Methoden `+equals(Object) : boolean` sowie `+hashCode() : int` so implementiert, dass zwei `Guard`-Objekte als gleich gelten, wenn ihre `wordA`- und `wordB`-Attribute identisch sind. Die weitere von `Object` geerbte Methode `+toString() : String` gibt ein `String`-Objekt der Form ‘‘`[wordA,wordB]`’’ zurück.

4.2.2 `Alphabet`

`Alphabet` dient der Repräsentation des Alphabets ($\Sigma_{\mathcal{T}}$) eines Transducers. Es handelt sich dabei weitestgehend um eine Wrapper-Klasse, die als einziges Attribut ein `HashSet`, mit der Bezeichnung `symbols`, aus `Guard`-Objekten besitzt. Damit die generischen Attribute dieser `Guard`-Objekte korrekt gesetzt werden, wird `Alphabet` bei der Instantiierung ebenfalls ein solcher generischer Datentyp `V` übergeben.

`Alphabet` besitzt die Methoden `+addSymbol(Guard) : boolean` und `+addAlphabet-(Alphabet) : boolean`, durch die dem bestehenden `Alphabet`-Objekt ein oder mehrere Symbole hinzugefügt werden können. Dabei liefern beide Methoden den Wert `true` zurück, sofern sich das Alphabet dadurch vergrößert hat. Außerdem wurde eine Standard-Getter-Methode implementiert, um das `HashSet` mit den `Guard`-Objekten übergeben zu können. Des weiteren ist ein Standard-Konstruktor implementiert, der ein leeres Alphabet erzeugt, sowie ein Konstruktor, der ein Alphabet aus einer übergebenen `Collection` von `Guard`-Objekten erstellt. Die von `Object` geerbte Methode `+toString() : String` erzeugt einen String in Mengennotation[2], der alle Elemente aus dem `HashSet symbols` enthält. Die weiteren von `Object` geerbten Methoden `+equals(Object) : boolean` und `+hashCode() + int` sorgen dafür, dass zwei `Alphabet`-Objekte als identisch gelten, wenn ihre `symbols-HashSet`-Objekte exakt die gleichen Elemente enthalten.

4.2.3 `State`

`State`-Objekte stellen Zustände eines Transducers dar. Somit finden sie sich in der Definition von Transducern als Elemente der Menge $Q_{\mathcal{T}}$ und dadurch ebenso in der Menge $F_{\mathcal{T}}$ (durch $F_{\mathcal{T}} \subseteq Q_{\mathcal{T}}$) sowie im Startzustand q_0 (durch $q_0 \in Q_{\mathcal{T}}$) wieder. `State`-Objekte haben ein `name`-Attribut vom generischen Datentyp `T`, sowie die `HashSet`-Objekte `edges` für ausgehende und `inputEdges` für eingehende Kanten. Wie bereits bei der `Guard`-Klasse, wurde sich hier für einen generischen Datentyp entschieden, um dem Entwickler sämtliche Freiheiten bei der Wahl der Zustandsbezeichnung zu gewährleisten. Ausgangs- und Eingangskanten werden durch `Edge`-Objekte realisiert, welche im Abschnitt 4.2.4 genauer erläutert werden. An dieser Stelle sei jedoch bereits gesagt, dass die `Edge`-Klasse

[2] in geschweiften Klammern durch Kommata getrennt

Guard-Objekte verwendet. Um diese demnach korrekt erzeugen zu können, muss einem State-Objekt bei seiner Instantiierung zu seinem generischen Datentyp T ebenso der generische Datentyp V der Guard-Objekte übergeben werden. Des weiteren besitzt jedes State-Objekt eine schreibgeschützte Sicht auf die Kantenmengen, damit diese nicht ohne weiteres verändert werden können, sowie eine ITransducer-Referenz, um kenntlich zu machen, zu welchem Transducer es gehört. Auf diese Weise kann sichergestellt werden, dass ein State-Objekt nicht zwei Transducern angehört. Die Klasse ITransducer wird im Abschnitt 4.2.5 weiter erläutert.

Konstruktoren:

`+State (T name, ITransducer transducer)`

Erzeugt ein State-Objekt mit der durch name angegebenen Bezeichnung, sowie leeren Eingangs- und Ausgangskantenmengen. Zudem wird der ITransducer-Referenz transducer zugewiesen.

Methoden:

`+addEdge (State destination, Guard guard,`
`Transition peaTransition) : boolean`

Erzeugt ein neues Edge-Objekt, mit guard als Kantenbedingung, destination als Ziel, sofern dies Objekt dem gleichen Transducer angehört, und peaTransition als Transition (s. Abs. 4.2.4) und fügt dies der Kantenmenge hinzu, sofern eine solche Kante nicht bereits besteht und beide State-Objekte zum selben Transducer gehören. Ruft außerdem die private Methode `-addInputEdge(State, Guard, Transition)` von destination auf, um dort entsprechend die Eingangskante zu setzen.

`+addEdge (T destinationName, Guard guard,`
`Transition peaTransition) : boolean`

Verhält sich ähnlich wie die Methode `+addEdge(State, Guard, Transition) : boolean`. Jedoch wird hier lediglich der Name destinationName übergeben. Daraufhin wird zunächst das entsprechende Ziel-State-Objekt aus dem Transducer ausgelesen, bevor die gewünschte Kante erzeugt wird.

`+getEdges () : Collection`

Getter-Methode für das HashSet der Ausgangskanten. Anstatt des normalen HashSet, wird jedoch dessen schreibgeschützte Variante übergeben, damit Randbedingungen beim Entfernen von Kanten behandelt werden können.

`+getEdgesByGuard (Guard guard) : Collection`
: Erzeugt ein `HashSet` bestehend aus `Edge`-Objekten, deren `Guard`-Objekte gleich `guard` sind und gibt diese zurück.

`+getInputEdges () : Collection`
: Verhält sich exakt wie die Methode `+getEdges() : Collection`, gibt jedoch die schreibgeschützte Variante der Eingangskanten zurück.

`-addInputEdge (State source, Guard guard, Transition peaTransition)`
: Erzeugt ein neues `Edge`-Objekt anhand der übergebenen Parameter und fügt dies der Eingangskantenmenge hinzu.

`-removeInputEdge (Edge edge)`
: Entfernt `edge` aus der Eingangskantenmenge.

`#containsEdgeWithTransition (Transition transition) : boolean`
: Überprüft, ob ein `Edge`-Objekt mit dem übergebenen `Transition`-Objekt `transition` (s. Abs. 4.2.4) in der Ausgangskantenmenge enthalten ist. Ist dies der Fall, wird `true` zurückgeliefert.

`#removeAllEdges () : Collection`
: Entfernt sämtliche `Edge`-Objekte aus der Menge der Ausgangskanten. Dazu wird für jedes Objekt die Methode `#removeEdge(Edge) : Transition` aufgerufen, welche nach dem Löschvorgang die im `Edge`-Objekt hinterlegte Transition zurückliefert. Diese `Transition`-Objekte werden in einem `HashSet` gespeichert und dem aufrufendem Programm zurückgeliefert. Außerdem werden die jeweiligen Ziel-Objekte der Kanten aufgerufen, um dort die entsprechenden Objekte aus den Eingangskantenmengen zu entfernen.

`#removeEdge (Edge edge) : Transition`
: Entfernt `edge` aus der Menge der Ausgangskanten und gibt das darin enthaltene `Transition`-Objekt (s. Abs. 4.2.4) zurück.

Verbleibend sind die von `Object` geerbten Methoden `+equals(Object) : boolean`, sowie `+hashCode() : int`. Dabei gelten zwei ein `State`-Objekte als identisch, sofern ihr Name identisch ist. Dies hat sich im Verlauf der Implementierung als vorteilhaft erwiesen, muss jedoch bei der Verwendung der Klasse beachtet werden. Die weitere von `Object` geerbte Methode `+toString() : String` liefert die `String`-Repräsentation des generischen `T`-Objekts `name` zurück.

4.2.4 Edge

`Edge`-Objekte repräsentieren Kanten eines Transducers. Sie spiegeln sich in der Transducer-Definition in den Elementen von $E_{\mathcal{T}}$ wider. Ein `Edge`-Objekt hat vier Attribute. Dabei handelt es sich um die zwei `State`-Objekte `origin` sowie `destination`, welche Start und Ziel der Kante darstellen, das `Guard`-Objekt `guard`, welches die Kantenbedingung repräsentiert und eine Instanz der PEA-Transition, welche von der Kante modelliert wird, in Form des `Transition`-Objekts `peaTransition`. Letzteres wurde hinzugefügt, um sicher zu stellen, dass immer die korrekte Transition behandelt wird. Dies erlangt erst bei der Umsetzung des Transducer-Algorithmus an Relevanz. Es ist jedoch denkbar, dass dies auch für andere Anwendungen von Bedeutung sein könnte.

Da sowohl die `State`- als auch die `Guard`-Objekte einen generischen Parameter verlangen, müssen diese bei der Instantiierung eines `Edge`-Objekts auch an dieses übergeben werden.

Die Klasse `Edge` besitzt nur den Konstruktor `+Edge(State, State, Guard, Transition)`. Dieser erzeugt ein neues `Edge`-Objekt, mit den beiden `State`-Objekten als Start- und als Zielzustand und dem übergebenen `Guard`-Objekt als Kantenbedingung. Außerdem wird `peaTransition` das übergebene `Transition`-Objekt zugewiesen.

Die Methoden der `Edge`-Klasse belaufen sich lediglich auf Standard-Getter-Methoden für die vier Attribute, sowie den überschriebenen Methoden `+equals(Object) : true`, `+hashCode() : int` und `+toString() : String`. Dabei sind zwei `Edge`-Objekte identisch, wenn all ihre Attribute identisch sind. Die Methode `+toString() : String` erzeugt einen String der Form ‘‘`(origin, guard, destination)`’’.

4.2.5 ITransducer

Das Interface `ITransducer` bildet die Basis der Transducer-Implementierung. Durch die von ihm bereitgestellten Methoden wird es ermöglicht, Transducer durch Hinzufügen oder Entfernen von Zuständen zu verändern und bietet verschiedene Optimierungsmöglichkeiten, wie beispielsweise das Entfernen von Epsilon-Kanten. Außerdem besitzt auch `ITransducer` die generischen Datentypen `T` und `V`, um korrekt mit `Guard`- und `State`-Objekten umgehen zu können.

Methoden:

`+add (ITransducer transToAdd)`

Fügt dem Transducer `transToAdd` hinzu.

`+addAll (Collection transducers)`
: Fügt dem Transducer alle Elemente aus `transducers` hinzu.

`+addState (T name, boolean isInit, boolean isFinal)`
: Fügt dem Transducer einen neuen, durch `name` gekennzeichneten Zustand hinzu. Durch `isInit` und `isFinal` wird festgelegt, ob der neue Zustand Start- oder Endzustand wird.

`+containsName (T name) : boolean`
: Überprüft, ob `name` bereits für einen Zustand dieses Transducers verwendet wird, und liefert `true` zurück, wenn dem so ist.

`+getAlph () : Alphabet`
: Liefert das Alphabet des Transducers in Form eines `Alphabet`-Objekts.

`+getDeterministic () : ITransducer`
: Erzeugt aus diesem Transducer ein deterministisches Gegenstück und liefert dies zurück.

`+getFinals () : Collection`
: Liefert die Menge der Endzustände des Transducers.

`+getInit () : State`
: Liefert den Startzustand des Transducers.

`+getPEATransitions () : Collection`
: Liefert die PEA-Transitionen, die von dem Transducer repräsentiert werden.

`+getState (T name) : State`
: Liefert den durch `name` identifizierbaren Zustand zurück.

`+isFinal (State state) : boolean`
: Überprüft, ob Zustand `state` Endzustand dieses Transducers ist und gibt `true` zurück, wenn dem so ist.

`+isInit (State state) : boolean`
: Überprüft, ob Zustand `state` der Startzustand `init` dieses Transducers ist und gibt `true` zurück, wenn dem so ist.

`+minimize () : ITransducer`
: Erzeugt aus diesem Transducer den minimalen Transducer. Dazu muss sichergestellt sein, dass es sich um einen deterministischen Transducer handelt.

`+removeEpsilonEdges ()`
: Macht den Transducer frei von Epsilon-Kanten.

`+removeState (State state)`
: Entfernt Zustand `state` aus dem Transducer. Dabei muss sichergestellt sein, dass hinterher keine Kante mehr zu `state` führt oder davon abgeht.

`+removeUnreachables ()`
: Entfernt sämtliche Zustände, die nicht mehr vom Startzustand aus erreichbar sind.

`+toDotString () : String`
: Erzeugt eine `String`-Repräsentation des Transducers inklusive eines möglichen Fehlerzustands, welche als Eingabe für *Graphviz*[3] genutzt werden kann.

`+toDotString (boolean withDeadEnds) : String`
: Erfüllt die gleiche Aufgabe wie `+toDotString()`, mit der Ausnahme, dass hier durch Angabe von `withDeadEnds` angegeben werden kann, ob der Fehlerzustand mit ausgegeben wird.

`+updateState (State state, boolean isInit, boolean isFinal)`
: Aktualisiert einen bereits bestehenden Zustand `state` des Transducers anhand der übergebenen `boolean`-Werte `isInit` und `isFinal`.

4.2.6 AbstractTransducer

Die `AbstractTransducer`-Klasse ist eine Referenz-Implementierung des `Itransducer`-Interfaces. Dabei wird an darin weiterhin mit den generischen Datentypen `T` und `V` gearbeitet. Erbende Klassen müssen eine Anzahl von abstrakten Methoden implementieren, die eine genaue Kenntnis der Struktur dieser Datentypen voraussetzen.

Die Attribute der `AbstractTransducer`-Klasse entsprechen weitestgehend den Elementen der Transducer-Definition in Abschnitt 2.3:

- `Alphabet alph` entspricht $\Sigma_{\mathcal{T}}$.
- `Map states` entspricht der Menge $Q_{\mathcal{T}}$. Dabei wurde hier aus Performanzgründen eine `Map` gewählt, mit der Zustandsnamen auf den jeweiligen Zustand abgebildet werden, da auf diese Weise einzelne Zustände anhand ihres Namens schneller bestimmt werden können.

[3] Graphviz ist eine Software zur automatischen Graphen-Erzeugung aus einer Textdatei - `http://www.graphviz.org/`

- `State init` entspricht dem Startzustand q_0.
- `Collection finals` entspricht $F_{\mathcal{T}}$. Hier wurde auf eine Map verzichtet. Dies ist dadurch begründet, dass innerhalb der Klasse bedeutend weniger auf die `finals`-Menge zugegriffen werden muss.

$E_{\mathcal{T}}$ findet keine direkte Repräsentation innerhalb der `AbstractTransducer`-Klasse, da die Kanten innerhalb der `State`-Objekte realisiert sind.

Des weiteren wird das `Collection`-Objekt `peaTransitions` vorgehalten, welches die PEA-Transitionen enthält, die vom Transducer nachvollzogen werden. Außerdem besitzt die `AbstractTransducer`-Klasse verschiedene konstante `Comparator`-Objekten, um an verschiedenen Bereichen die Möglichkeit bereitzustellen, bestimmte Objektmengen zu sortieren.

`AbstractTransducer` liegt zudem die interne Klasse `Pair` bei. Diese wird an verschiedenen Stellen innerhalb der Klasse dazu verwendet, zwei beliebige Objekte miteinander zu verknüpfen.

`AbstractTransducer` implementiert sämtliche Methoden von `ITransducer` nach dessen Vorgaben. Daher werden im folgenden lediglich die Konstruktoren, sowie die weiteren Methoden aufgelistet und deren Funktionsweise beschrieben.

Konstruktoren:

`+AbstractTransducer ()`

Standard-Konstruktor, der sämtliche Mengen-Attribute initialisiert, sowie ein leeres Alphabet erzeugt.

`+AbstractTransducer (PhaseEventAutomata pea)`

Erzeugt anhand von `pea` einen Transducer. Dazu wird zunächst der Standard-Konstruktor aufgerufen und darauf die private `-generate(PhaseEventAutomata)`-Methode des entstehenden Objekts.

Methoden:

```
-adjustTransducer (Collection states, int n, boolean loop,
        boolean containsInit) : AbstractTransducer
```

Entspricht der in Abschnitt 3.2.3 aufgeführten Funktion `adjustTransducer()`. Somit werden an dieser Stelle die in Abschnitt 3.2.2 genannten Sonderfälle bei der Transducer-Generierung behandelt.

`-copy () : AbstractTransducer`

Erzeugt eine exakte Kopie dieses Transducers und gibt diese zurück.

`-createOrderedPair (State p, State q) : Pair`

Liefert ein `Pair`-Objekt zurück, in dem zwei anhand ihrer `name`-Attribute geordnete `State`-Objekte enthalten sind.

`-fixState (State state)`

Behandelt die Ausgangskanten des übergebenen Zustands `state`. Sofern es sich dabei um eine Epsilon-Kante handelt, wird diese entfernt und dafür sämtliche Kanten vom Folgezustand zu dessen Folgezustand mit der jeweiligen Kantenbedingung zum Zustand `state` hinzugefügt.

`-generate (PhaseEventAutomata pea)`

Macht diesen Transducer zur Repräsentation von `pea`. Dies erfolgt weitestgehend nach der Vorgehensweise des in Abschnitt 3.2.3 aufgeführten Algorithmus, sofern es sich bei `pea` um eine Instanz von `singleParamPEA` und somit um einen parametrisierten PEA handelt. Ist dies nicht der Fall, wird dennoch ein Transducer erzeugt. Dabei handelt es sich dann jedoch lediglich um einen Transducer, der nur einen Schritt eines einzelnen PEA nachvollziehen kann.

`-generateComTransducer (List tupel) : AbstractTransducer`

Erzeugt einen Kommunikations-Transducer anhand einer geordneten Liste bestehend aus PEA-Transitionen und liefert diesen zurück.

`-generateSimpleTransducer (Transition transition, boolean isParam,`
`boolean isID) : AbstractTransducer`

Erzeugt einen einfachen Transducer für `transition` und liefert diesen zurück. Unter einem einfachen Transducer sind die in Kapitel 3 genannten `id`-Transducer sowie die Transducer ohne Kommunikation zu verstehen. Zusätzlich kann aber auch ein dritter Typ von Transducern entstehen, der einem Transducer ohne Kommunikation entspricht, jedoch ohne die Schleifenkanten. Dieser entsteht, wenn die Eingabe von `-generate(PhaseEventAutomata)` einem nicht parametrisierten PEA entspricht.

`-getHigherStateSet (AbstractTransducer transToTreat,`
`Transition transition) : Set`

Liefert eine Menge von Zuständen aus `transToTreat` zurück, die nach dem Ausführen der Kante erreicht werden können, die die PEA-Transition `transition` repräsentiert.

`-getLowerStateSet (AbstractTransducer transToTreat,`
`Transition transition) : Set`

Liefert eine Menge von Zuständen aus `transToTreat` zurück, die ausgehend vom Startzustand Zwischenzustände bis zu dem Zustand darstellen, der die Ausgangskante besitzt, die die PEA-Transition `transition` repräsentiert, inklusive der beiden Grenzzustände.

`-getRealComSubsets (List comTrans) : Collection`

Erzeugt aus der übergebenen Liste `comTrans` aus `PEA-Transitionen` eine Menge von Listen, die jeweils Elemente von `comTrans` enthalten, denen es möglich ist, gleichzeitig miteinander zu kommunizieren. Dabei kann eine PEA-Transition in mehreren Listen enthalten sein.

`-getStateWithLoopAndKillLoop (Collection states,`
`AbstractTransducer transducer) : State`

Ermittelt innerhalb von `states` den Zustand, der eine Schleife enthält. Diese Schleife wird entfernt und der gefundene Zustand zurückgeliefert.

`-getValueAndOp (String termString, String param) : Pair`

Liefert für einen gegebenen `termString` ein `Pair`-Objekt, bestehend aus dem Vergleichsoperator (=, >, <, >=, =<), sowie dem Wert, mit dem darin der übergebene durch `param` definierte Parameter in Relation gesetzt wird.

`-removeAllEdgesFromState (State state) : boolean`

Ruft die Methode `#removeAllEdges()` von `state` auf und entfernt die davon zurückgelieferten PEA-Transitionen aus `peaTransitions`, wenn keine andere Kante innerhalb dieses Transducers existiert, die die gleiche Transition repräsentiert. Liefert `true` zurück, wenn sich dadurch `peaTransitions` verändert hat.

`-removeEdge (State state, Edge edge) : boolean`

Verhält sich ähnlich wie die eben genannte Methode, mit der Ausnahme, dass hier nur eine Kante entfernt wird.

`-reverseOperator (String operator)`

Liefert `operator` gespiegelt zurück. Dabei wird beispielsweise ‘‘>=’’ zu ‘‘=<’’.

`-specialTreatment (List paramValueOpList,`
`AbstractTransducer transToTreat,`
`Transition transition) : AbstractTransducer`

Ermittelt anhand von `paramValueOpList` um welchen der Sonderfälle aus Abschnitt 3.2.2 es sich handelt und ruft entsprechend mit den sich daraus ergebenden Parametern die Methode `-adjustTransducer()` auf.

`-ensureDeterministic ()`
Stellt sicher, dass es sich bei diesem Transducer um einen deterministischen Transducer handelt.

Abstrakte Methoden:

`+newInstance () : AbstractTransducer`
Gibt eine Objekt-Instanz der implementierenden Klasse zurück.

`#getNameFor (Collection) : T`
Erzeugt einen Namen für eine Menge von Zuständen, die zu einem neuen Zustand zusammengefasst werden sollen.

`#getNewStateName () : T`
Gibt einen Zustandsnamen zurück, der noch nicht in dem Transducer vorgekommen ist.

`#getGuardComparator () : Comparator`
Liefert ein `Comparator`-Objekt zurück, mit dem sich `Guard`-Objekte vergleichen lassen.

`#getStateComparator () : Comparator`
Liefert ein `Comparator`-Objekt zurück, mit dem sich `State`-Objekte vergleichen lassen.

`#newGuard (Transition transition)`
Erzeugt ein neues `Guard`-Objekt aus der PEA-Transition `transition` und liefert dies zurück.

In der Auflistung nicht genannte Methoden sind die Standard-Getter-Methoden der zu Beginn genannten Attribute, sowie die von `Object` geerbte Methode `+toString() : String`, welche eine `String`-Repräsentation des Transducers erzeugt. Diese gleicht weitestgehend der formalen Schreibweise von Transducern.

4.2.7 `StringTransducer`

Die Klasse `StringTransducer` ist eine Unterklasse von `AbstractTransducer`. Die generischen Datentypen `T` und `V` bei einem `StringTransducer`-Objekt jeweils dem Datentyp `String`. Die von `AbstractTransducer` geerbten abstrakten Methoden sind demnach vollständig so implementiert, dass Objekte diesen Datentyps erzeugt werden. In den Fällen, in denen Zustandsnamen erzeugt werden müssen, besitzen diese den Präfix ‘‘`p`’’ gefolgt von einer natürlichen Zahl. Diese wird durch das Attribut `int stateCount` von `StringTransducer` bestimmt, welches immer dann inkrementiert wird, wenn ein neuer Zustandsname generiert wurde.

Die Form der `State`- und `Guard`-Objekte eines `StringTransducer`-Objekts entsprechen den in bisherigen Kapiteln verwendeten Formen für Zustände und Kantenbedingungen.

5 Beispiel

Für eine beispielhafte Anwendung der Ergebnisse, wurde das in Abbildung 5.1 gezeigte parametrisierte Programm MUX aus [KMM+01] gewählt, welches den wechselseitigen Ausschluss durch synchrone Kommunikation implementiert. Ein wechselseitiger Ausschluss ist so zu verstehen, dass ein kritischer Abschnitt besteht, der in einem parallelen Prozessverbund immer nur von einem Prozess zur Zeit betreten werden kann.

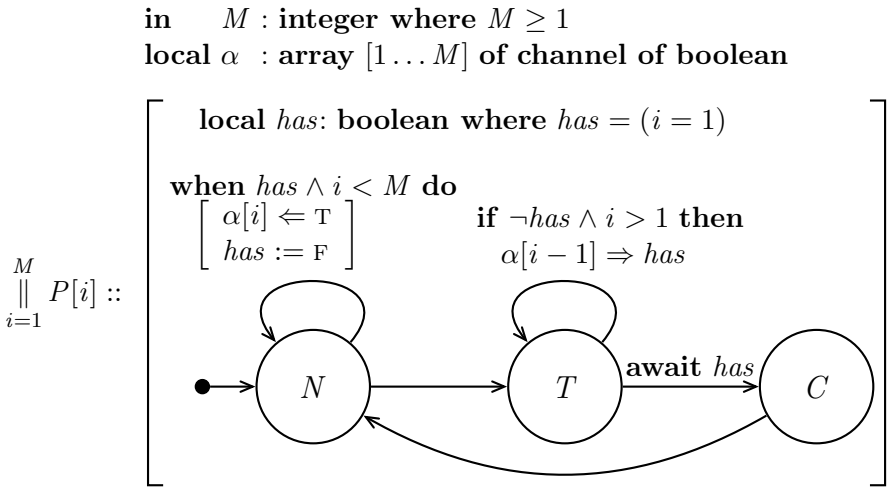


Abbildung 5.1: Parametrisiertes Programm MUX

Jeder Prozess $P[i]$ mit $i \leq M$, der das Programm MUX ausführt, besitzt die zwei Zustandsvariablen *has* und π. *has* ist dabei eine boolesche Variable und π gibt an, ob sich der Prozess derzeit in dem unkritischen Abschnitt (N) oder dem kritischen Abschnitt (C) befindet, bzw. ob er versucht, den kritischen Abschnitt zu betreten (T). Die Synchronisation findet über das globale Array α statt. Dies wird von jedem Prozess $P[i]$ dazu verwendet, einen booleschen Wert durch Zugriff auf $\alpha[i]$ an seinen rechten Nachbarn zu senden, oder durch $\alpha[i-1]$ von seinem linken Nachbarn zu empfangen.

Entspricht der Wert der *has*-Variable von Prozess $P[i]$ dem booleschen Wert T, und seine Variable π hat den Wert N, gibt er den Wert von *has* an seinen rechten Nachbarn $P[i+1]$ ab, sofern dieser dazu bereit ist, ihn zu empfangen und $i < M$ gilt. Dazu muss die

Variable π von Prozess $P[i+1]$, den Wert T und die Variable *has* den Wert F besitzen. Ist dies der Fall, wird der Wert T von Prozess $P[i+1]$ empfangen und seine Variable *has* auf diesen Wert gesetzt, sowie die Variable von Prozess $P[i]$ auf den Wert F. Auf diese Weise ist sichergestellt, dass ein Prozess $P[i]$ den kritischen Abschnitt C nur dann betreten kann, wenn $P[i].has = \text{T}$ gilt.

Dieses Programm wird im Folgenden zunächst in ein passendes CSP-OZ-DC-Schema modelliert, bevor es darauf anhand der in [Hoe06, Abschnitt 4.2] genannten Methoden zu einem entsprechenden PEA übersetzt wird. Daraufhin wird dieser PEA im PEA-Toolkit implementiert und durch die `StringTransducer`-Klasse der entsprechende Transducer erzeugt und ausgegeben.

5.1 CSP-OZ-DC-Schema

Abbildung 5.2 auf der gegenüberliegenden Seite zeigt das beschriebene parametrisierte Programm MUX als CSP-OZ-DC-Schema. Da der in dieser Arbeit entwickelte Transducer-Algorithmus die Variablen eines PEA, die sich bei der Modellierung aus denen eines CSP-OZ-DC-Schemas ergeben, noch nicht berücksichtigt, wurden die Prozess-Namen des CSP-Teils, außer dem von `main`, so gewählt, dass sich die Ausprägungen der Variablen π und *has* direkt darin ablesen lassen. Die Schreibweise entspricht dabei der in [KMM+01] verwendeten Notation. Durch $\langle N, \text{T} \rangle$ wird beispielsweise der Zustand $P[i].\pi = N, P[i].has = \text{T}$ dargestellt.

Am CSP-Teil des Schemas lässt sich demnach bereits das Verhalten bei der Ausführung des Programms MUX ablesen.

Der ausführende Prozess beginnt als `main`. Dabei werden bereits die Werte der Variablen durch das `Init`-Schema auf die geforderten Anfangswerte gesetzt. π entspricht dabei zu Beginn immer dem Wert N. Die *has*-Variable jedoch wird nach Abhängigkeit vom Parameter i gesetzt. *has* entspricht nur in dem Fall dem Wert true wenn der ausführende Prozess den Parameter $i = 1$ hat. Die *has*-Variable aller anderen Prozesse bekommt den Wert false. Darauf wird durch den $\Box$-Operator festgelegt, dass das weitere Verhalten des Prozesses dadurch bestimmt wird, welche der beiden Events $equalsOne_i$ sowie $greaterOne_i$ ausführbar sind. Dies wird durch die Operationen `com_`$equalsOne_i$ und `com_`$greaterOne_i$ bestimmt. Diese prüfen, ob der Wert von i gleich 1, respektive größer 1 ist. Demnach geht der Prozess `main` in $\langle N, \text{T} \rangle$ über, wenn sein Parameter den Wert 1 hat. In jedem anderen Fall findet ein Übergang in Prozess $\langle N, \text{F} \rangle$ statt.

Prozess $\langle N, \text{T} \rangle$ stößt das globale Event $go.i+1$ an, was dem Versuch einer synchronen Kommunikation entspricht. Befindet sich der Prozess mit dem Parameter $i+1$ zu dem

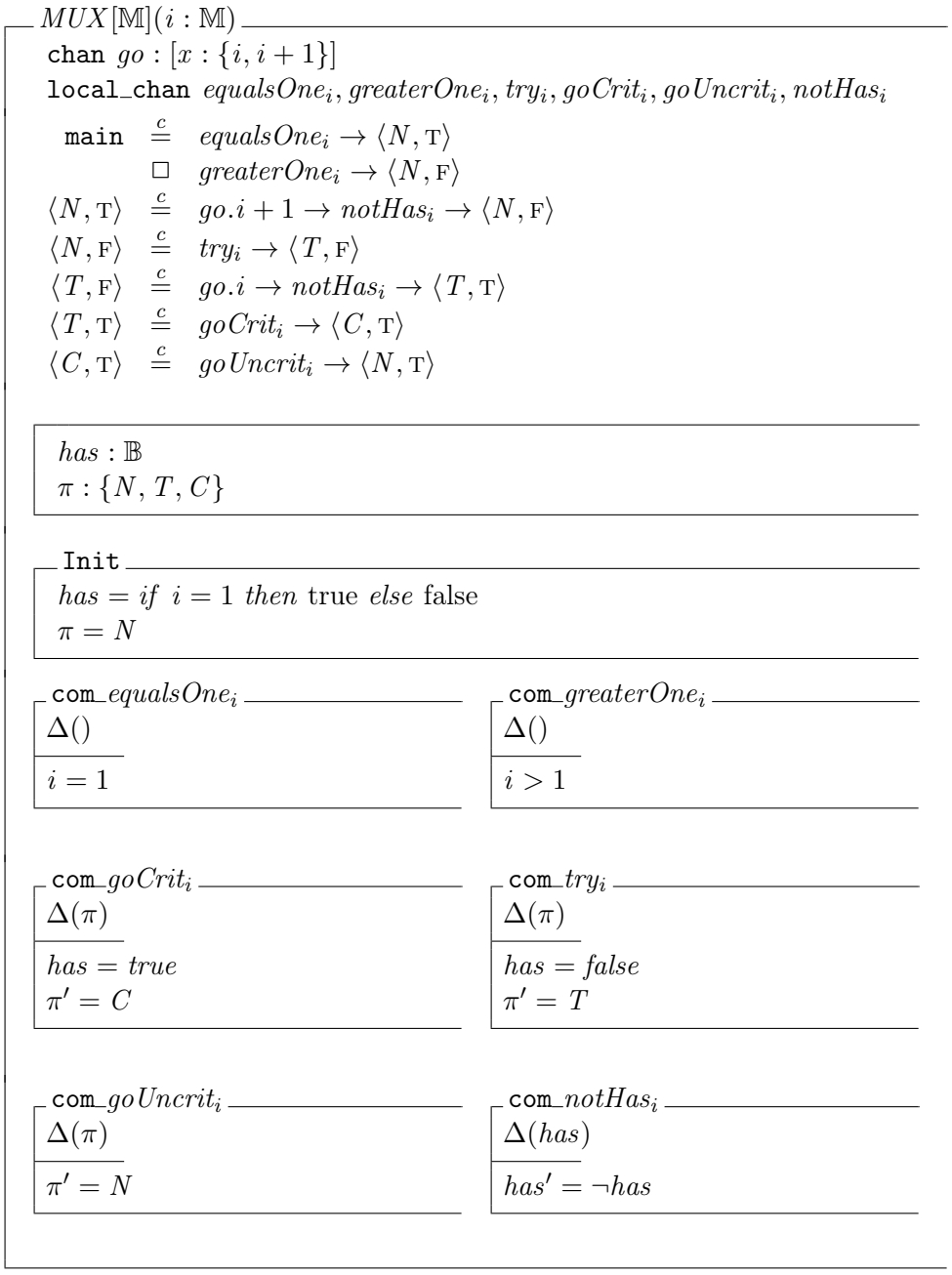


Abbildung 5.2: CSP-OZ-DC-Schema zum parametrisierten Programm MUX

Zeitpunkt in $\langle T, \text{F} \rangle$, kann die Kommunikation stattfinden, da an dieser Stelle das Event $go.i$ angestoßen wird. Sollte sich der gewünschte Prozess nicht in $\langle T, \text{F} \rangle$ befinden, kann keine Kommunikation stattfinden und das Event $go.i + 1$ wird nicht ausgeführt. Ist die Kommunikation jedoch erfolgreich, können die teilnehmenden Prozesse jeweils das Event *notHas* anstoßen. Dadurch wird die Operation `com_`*notHas* ausgeführt, die dafür sorgt, dass die Prozesse die Werte ihrer jeweiligen *has*-Variable negieren. So geht der Prozess $\langle N, \text{T} \rangle$ in $\langle N, \text{F} \rangle$ über, sowie der Prozess $\langle T, \text{F} \rangle$ in $\langle T, \text{T} \rangle$. $\langle T, \text{T} \rangle$ kann darauf das Event *goCrit* anstoßen und in den Prozess $\langle C, \text{T} \rangle$ übergehen. Die Operation `com_`*goCrit* stellt dabei sicher, dass der Wert π auf C gesetzt wird. Damit hat der Prozess den kritischen Zustand betreten.

Anhand dieser Vorgehensweise ist erkennbar, dass ein Prozess $P[i]$ nur dann in den kritischen Zustand eintreten darf, wenn es ihm von seinem „linken Nachbarn“ $P[i-1]$ erlaubt wurde, was dem gewünschten Verhalten entspricht.

5.2 Phasen-Event-Automat

In diesem Abschnitt wird das CSP-OZ-DC-Schema aus Abbildung 5.2 in einen PEA übersetzt. Dazu wird schrittweise nach dem Verfahren aus [Hoe06, Kapitel 5] vorgegangen und dadurch gezeigt, dass sich dieses auch für parametrisierte CSP-OZ-DC-Schemata ohne DC-Teil eignet. Wie bereits in Abschnitt 3.1 erwähnt, werden Transitionen (p, g, X, p') durch (p, g, p') abgekürzt, da keine Uhren existieren werden, die zurückgesetzt werden müssen.

Der durch die Vorgehensweise aus [Hoe06, Abschnitt 5.1] entstehende PEA $\mathcal{A}_{MUX_{CSP}}$ für den CSP-Teil wird in Abbildung 5.3 gezeigt.

Die Kantenbedingungen haben jeweils die Form $only(e)$ mit $e \in A_i \cup \{\tau\}$. Damit wird angegeben, dass die entsprechende Transition nur dann ausgeführt werden kann, wenn einzig das angegebene Event auftritt. τ ist dabei als leeres Event zu verstehen und tritt lediglich an den Stotterkanten auf. Stotterkanten dürfen demnach nur ausgeführt werden, wenn kein Event aus der Eventmenge A_i von $\mathcal{A}_{MUX_{CSP}}$ auftritt.

Der Parameter i findet sich korrekt als Index der lokalen Events, sowie innerhalb des Parameters des globalen Events *go* wieder.

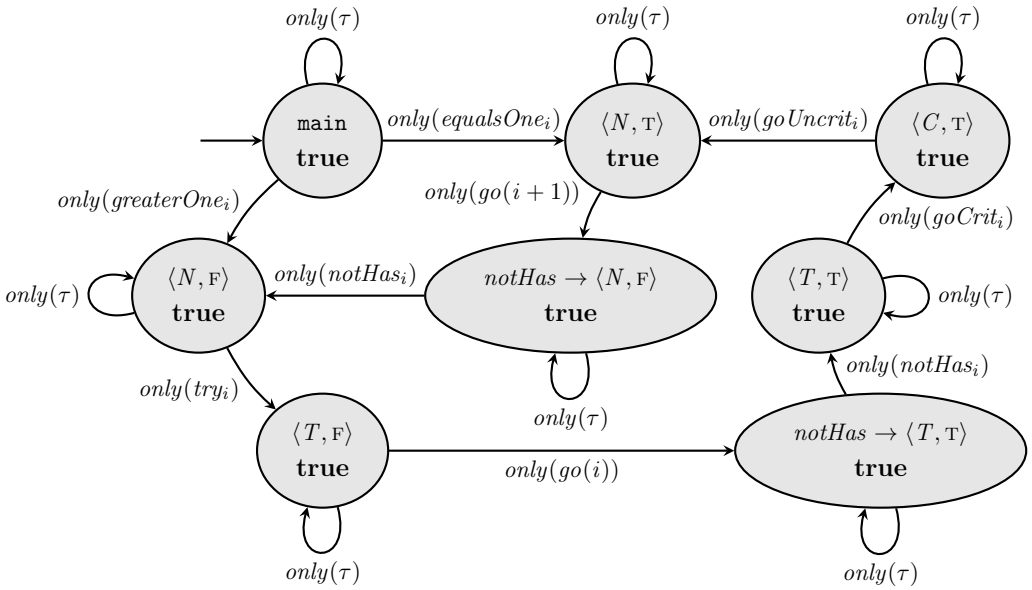


Abbildung 5.3: Phasen-Event-Automat $\mathcal{A}_{MUX_{CSP}}$ für den CSP-Teil von MUX

Das Auftreten des Werts **true** innerhalb jedes Zustands stellt die jederzeit geltende positive Uhren-Invariante dar.

Der in Abbildung 5.4 abgebildete PEA $\mathcal{A}_{MUX_{OZ}}$ stellt die Übersetzung des OZ-Teils anhand der Vorgehensweise aus [Hoe06, Abschnitt 5.2] dar. Das Vorkommen des Werts **true** im Zustand p_0, sowie die Kantenbedingungen $only(e)$ haben die gleiche Bedeutung, wie bei der Übersetzung des CSP-Teils. Auffällig ist hierbei das Fehlen von go. Dies ist dadurch begründet, dass die Kantenbedingungen von $\mathcal{A}_{MUX_{OZ}}$ nur die Events aufführen, die eine Operation im OZ-Teils des CSP-OZ-DC-Schemas. Demnach dient $\mathcal{A}_{MUX_{OZ}}$ dazu, sicher zu stellen, dass sämtliche Bedingungen dafür erfüllt sind, wohingegen $\mathcal{A}_{MUX_{CSP}}$ die gültige Reihenfolge der Events aufzeigt.

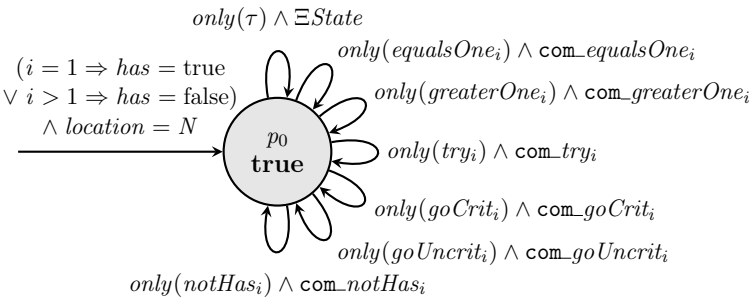


Abbildung 5.4: Phasen-Event-Automat $\mathcal{A}_{MUX_{OZ}}$ für den OZ-Teil von MUX

Die beiden PEA lassen sich durch die Definition der parallelen Komposition zweier PEA zu dem in Abbildung 5.5 auf der nächsten Seite gezeigten Produktautomaten $\mathcal{A}_{MUX}$ zusammenfassen.

Dadurch wurde gezeigt, dass es ohne weiteres möglich ist, einen parametrisierten PEA durch bestehende Methoden aus einem parametrisierten CSP-OZ-DC-Schema zu erzeugen. Dabei wurde der Parameter i jeweils als fester Wert behandelt und so bei den Übersetzungen direkt übernommen. Wird nun aus dem entstandenen PEA ein parametrisiertes PEA-System initialisiert, können die Vorkommen von i mit tatsächlichen Werten versehen werden.

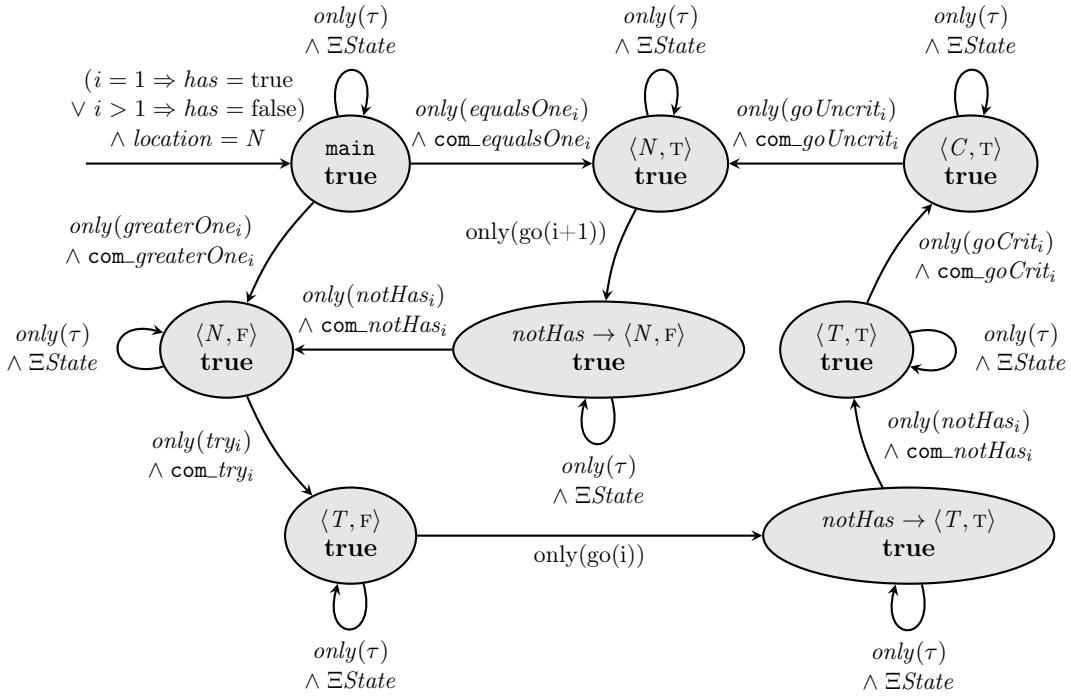


Abbildung 5.5: Produkt-Phasen-Event-Automat $\mathcal{A}_{MUX}$ für MUX

5.3 Anwendung des implementierten Algorithmus

Der im vorigen Kapitel erstellte PEA $\mathcal{A}_{MUX}$ (s. Abb. 5.5) kann nun als Eingabe für den implementierten Transducer-Algorithmus dienen. Dazu muss ein entsprechendes `SingleParamPEA`-Objekt erzeugt werden. Da der Algorithmus alle Kantenbedingungen, die nicht direkt den Parameter i behandeln, ignoriert, sofern es sich nicht um auftretende Events handelt, müssen diese ebenfalls in der Objekt-Repräsentation nicht berücksichtigt

werden. Dies stellt kein Problem dar, da das CSP-OZ-DC-Schema, aus der der PEA hervorgegangen ist, unter Berücksichtigung dieser Tatsache entworfen wurde.

Ausgangspunkt der Implementierung bildet die eigens dafür erzeugte Klasse `Test` innerhalb des Pakets `pea.modelchecking.regular`. Innerhalb dieser Klasse wurde die private Methode `-createMux() : PhaseEventAutomata` realisiert. Diese erzeugt den gewünschten PEA in Form eines `PhaseEventAutomata`-Objekts und liefert dies zurück.

Innerhalb der `+main()`-Methode der Klasse `Test` wird die Methode `-createMux()` aufgerufen und das zurückgegebene Objekt dazu verwendet, ein entsprechendes `SingleParamPEA`-Objekt mit dem Parameter ‘‘`i`’’ zu erzeugen.

Durch die Übergabe dieses `SingleParamPEA`-Objekt als Parameter des entsprechenden `StringTransducer`-Konstruktors wird das `StringTransducer`-Objekt erzeugt und mit der `muxTransducer`-Variable referenziert.

An dieser Stelle können jetzt sämtliche der öffentlichen Methoden von `StringTransducer` angewendet werden. Als erstes soll jedoch die Ausgabe der `+toDotString()`-Methode die korrekte Funktion der Transducer-Algorithmus-Implementierung aufzeigen. Diese soll hier zugunsten der Lesbarkeit nicht weiter aufgeführt werden. Die Eingabe des zurückgelieferten `String`-Objekts in das Programm *Graphviz*[1] erzeugt jedoch die in Abbildung 5.6 gezeigte grafische Repräsentation.

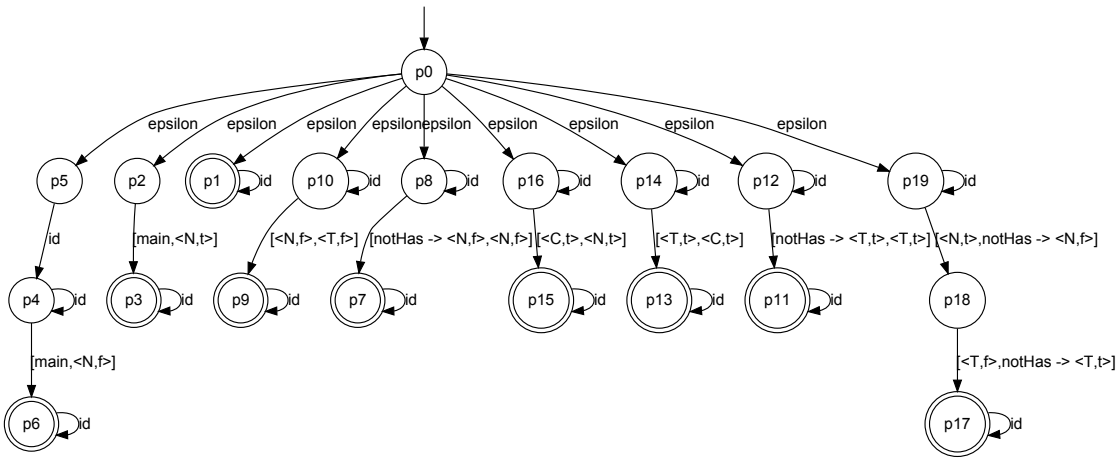


Abbildung 5.6: Struktur des `muxTransducer`-Objekts unmittelbar nach Instantiierung

Bei dem Ergebnis handelt es sich um einen annährend optimal angeordneten Transducer. Daran lässt sich die Struktur der einzelnen vom Algorithmus erzeugten Teiltransducer erkennen, die mit dem Startzustand p0 durch Epsilon-Kanten verknüpft wurden. So

[1] `http://www.graphviz.org/`

entspricht der Zustand p1 dem zu Beginn des Algorithmus erzeugten `id`-Transducer. Das Fehlen der Schleife an den Zuständen p2 und p5 deutet auf die Behandlung von Sonderfällen hin. Bei genauerer Betrachtung wird deutlich, dass in der Tat die von der Transducer-Kante $(p2, [main, \langle N, \mathrm{T}\rangle], p3)$ repräsentierte PEA-Transition $(main, only(equalsOne_i) \wedge \texttt{com\_}equalsOne_i, \langle N, \mathrm{T}\rangle)$ von $\mathcal{A}_{MUX}$ fordert, dass $i = 1$ gelten muss. In den Zuständen p19, p18 und p17 lässt sich außerdem der in [KMM+01] für den Kommunikationsschritt erstellte Transducer wiedererkennen, wenn von der exakten Beschriftung der Kanten abgesehen wird.

Durch die Anwendung der Methoden `+removeEpsilonEdges()`, `+getDeterministic()` und `+minimize()` entsteht darauf der deterministische Minimal-Transducer zu dem aus Abbildung 5.6. Wird dessen durch `+getDotString()` zurückgelieferter String in Graphviz eingegeben, entsteht die in Abbildung 5.7 gezeigte Grafik. Zur besseren Lesbarkeit wurde die Methode `+getDotString()` mit dem Parameter `false` aufgerufen, um den Fehlerzustand auszublenden.

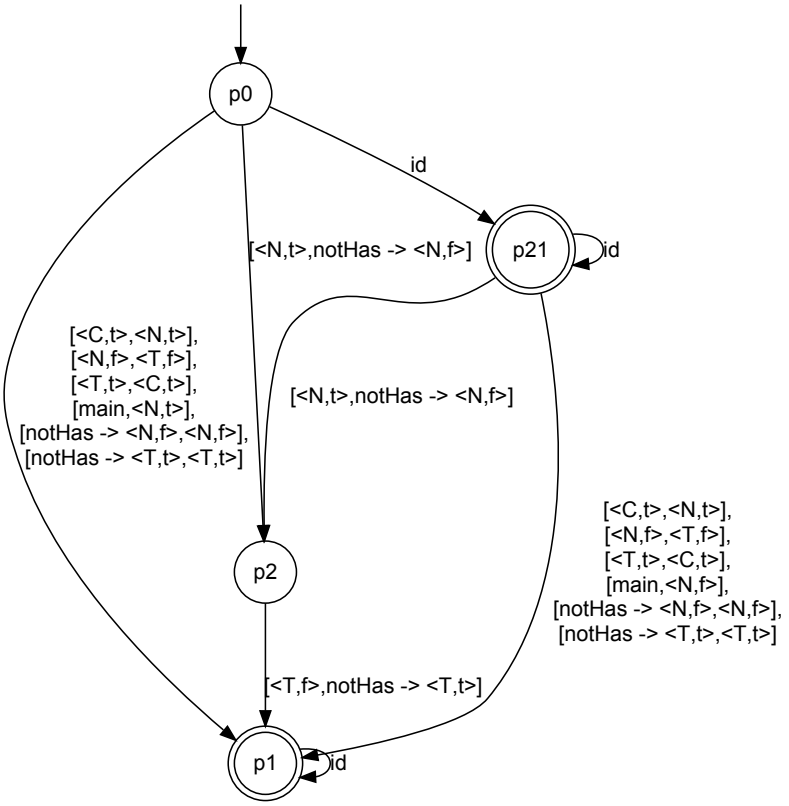


Abbildung 5.7: Minimal-Transducer zu $\mathcal{A}_{MUX}$

Anhand dieser Grafik lässt sich erkennen, dass es lediglich dem PEA, dessen Parameter

dem Wert 1 entspricht, möglich ist, vom Zustand `main` in den Zustand $\langle N, \mathrm{T} \rangle$ überzugehen. Außerdem wird deutlich, dass die PEA-Transition $(\langle T, \mathrm{F} \rangle, \mathit{only}(\mathit{go}(i)), \mathit{notHas} \rightarrow \langle T, \mathrm{T} \rangle)$, die den Übergang in den kritischen Zustand vorbereitet, nur dann von einem PEA i ausgeführt werden kann, wenn der PEA $i - 1$ während des gleichen Schritts die Transition $(\langle N, \mathrm{T} \rangle, \mathit{only}(\mathit{go}(i + 1)), \mathit{notHas} \rightarrow \langle N, \mathrm{F} \rangle)$ realisiert.

Auf diese Weise wurde gezeigt, dass ein aus einem parametrisierten CSP-OZ-DC-Schema entstandener PEA als Eingabe des implementieren Transducer-Algorithmus dienen kann, um daraus einen Transducer zu erzeugen. Für diesen Transducer kann durch die in Abschnitt 2.3 vorgestellten Methoden darauf die reflexive transitive Hülle $\mathcal{T}^*$ berechnet werden.

6 Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, eine Grundlage für das reguläre Model-Checking parametrisierter PEA zu schaffen. Dazu wurde in Abschnitt 3.1 zunächst gezeigt, dass das formale PEA-Modell für die Parametrisierung keinerlei Anpassung bedurfte. Lediglich der korrekte Umgang mit dem Parameter i als Konstante muss eingehalten werden.

Abschnitt 3.2 stellt zudem einen formalen Algorithmus vor, der dazu in der Lage ist, den zum regulären Model-Checking benötigten Transducer aus einem parametrisierten PEA zu generieren. Die daraus entstandene Implementierung ist so gehalten, dass Anpassungen an den Algorithmus minimale Erweiterungen der Transducer-Klassenstruktur nach sich ziehen. Wird beispielsweise eine Variante des Transducer-Algorithmus erstellt, die auch die zeitliche Komponente eines parametrisierten PEA betrachtet, bei der spezielle Kantenbedingungen zum Einsatz kommen, so kann dieser ohne weiteres von der Transducer-Klasse realisiert werden. Gleiches gilt für die bessere Einbindung der Variablenwerte. Dabei wäre ein Algorithmus denkbar, der die Zustandsübergänge eines parametrisierten PEA nicht nur anhand der Zustandsnamen realisiert, sondern die Variablen eines PEA ausliest und daraus selbstständig die in Kapitel 5 genutzte Form $\langle \mathit{Variablenwert}1, \mathit{Variablenwert2}, \ldots \rangle$ generiert. Wird dazu ein neuer Datentyp für die Kantenbedingungen des Transducer entwickelt, muss dafür lediglich eine neue Unterklasse zu `AbstractTransducer` implementiert werden, die mit diesem Datentyp umzugehen versteht. Die Methoden zum Entfernen von Epsilon-Übergängen, sowie zur Erzeugung des deterministischen Automaten und zu dessen Minimierung, lassen sich darauf bei korrekter Implementierung der geerbten abstrakten Methoden ohne weiteres auf die neu entstandene Transducer-Klasse anwenden.

Durch die Entscheidung, die implementierte Klassenstruktur direkt in das PEA-Toolkit einzubinden, wurde außerdem das Fundament dazu geschaffen, die Ergebnisse dieser Arbeit in das Werkzeug Syspect zu integrieren.

Abbildungsverzeichnis

Literaturverzeichnis

[AJNS04] P.A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. A survey of regular model checking. In *Lecture Notes in Computer Science*, pages 35–48. Springer, 2004.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999.

[Hoe06] J. Hoenicke. *Combination of Processes, Data, and Time*. PhD thesis, Carl von Ossietzky Universität Oldenburg, 2006.

[KMM+01] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *Theoretical Computer Science*, volume 256, pages 93–112. Elsevier, 2001.

[Xu05] J.J. Xu. *Automatic Verification of Parameterized Systems*. PhD thesis, New York University, 2005.