UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

# Minimierung symbolischer nichtdeterministischer Büchi-Automaten

*Minimization of symbolic nondeterministic Büchi automata*

**Masterarbeit**
im Rahmen des Studiengangs
**Informatik**
der Universität zu Lübeck

vorgelegt von
**Johannes Friedrich Thorn**

ausgegeben und betreut von
**Prof. Dr. Martin Leucker**

mit Unterstützung von
Torben Scheffel

Lübeck, den 15. Mai 2015

# Kurzfassung

In der Softwareverifikation stellen nichtdeterministische Büchi-Automaten (NBA) ein anerkanntes Hilfsmittel für eine große Anzahl von Anwendungen dar. Zur Verringerung des Rechenaufwands besteht ein Bedarf an praktisch anwendbaren Minimierungstechniken für solche Automaten.

Diese erhalten wir durch die Verwendung von symbolisch beschriebenen Automaten, die genauer und effizienter als explizite NBAs beschrieben werden können. Während die Minimierung expliziter NBAs bereits ausführlich untersucht wurde, ist die Minimierung der symbolischen NBAs bisher unerforscht geblieben.

Daher werden wir in dieser Arbeit verschiedene Minimierungstechniken für symbolische NBAs entwickeln. Wir präsentieren zwei Automatenmodelle als Abwandlung der expliziten nichtdeterministischen Büchi-Automaten. Diese Modelle nutzen die besondere Struktur der Eingabealphabete.

Außerdem wurden die Minimierungsverfahren für explizite NBAs aus [CM13] auf die beiden neuen Automatenmodelle übertragen. Schließlich ist das Minimierungsframework auf der Grundlage binärer Entscheidungsdiagramme in der Programmiersprache Scala implementiert worden.

# Abstract

In software verification nondeterministic Büchi automata (NBA) are a powerful tool for a remarkable variety of applications. In order to reduce the computational effort there is a need for practical minimization techniques.

We achieve this by using automata with symbolic representations, which provide a more accurate and efficient formulation than explicit NBAs. While minimization of explicit NBAs has been extensively studied, minimization of symbolic NBAs has remained unstudied.

Hence in this work we will develop different minimization techniques for symbolic NBAs. We present two automata models modifying the explicit nondeterministic Büchi automata. These models make use of a particular structure of the input alphabets.

The minimization procedure for explicit NBAs from [CM13] was transferred to the two new automata models. Finally we implemented the minimization framework on the basis of binary decision diagrams in the programming language Scala.

# Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

_____

Lübeck, 15. Mai 2015

# Inhaltsverzeichnis

# 1 Einleitung

In der heutigen Zeit nehmen Computer und Algorithmen immer mehr Aufgaben wahr, die sicherheitskritische Aspekte oder anderweitige erhöhte Ansprüche an die Korrektheit der Implementierung aufweisen. Um diesen Ansprüchen gerecht zu werden, wurden verschiedene Ansätze zur Überprüfung der Korrektheit von Software entwickelt, die alle überprüfen, ob ein gegebenes Programm seine Spezifikation erfüllt. Dies wird auch als Softwareverifikation bezeichnet und wird in drei große Bereiche unterteilt: Testen, Laufzeitverifikation und Model-Checking.

Manuelles oder automatisiertes *Testen* überprüft die Korrektheit, indem die Ausgabe des Programms für eine gegebene Eingabe mit der erwarteten Ausgabe verglichen wird. Dieses Vorgehen ist sehr einfach durchzuführen, da nur eine Eingabe bestimmt werden muss, für die die Ausgabe des Programms überprüft wird. Gleichzeitig können immer nur die Programmausführungen getestet werden, für die auch eine Eingabe existiert, die genau diesen Ablauf der Verarbeitung bedingt. Testen kann als Verifikationsmethode also nicht die Korrektheit des Programms beweisen, sondern nur beim Auffinden von Fehlern helfen.

*Laufzeitverifikation* überprüft während der Ausführung des Programms fortlaufend, ob die formal beschriebene Spezifikation eingehalten wird. Dabei wird die Erfüllung der Spezifikation auf einer höheren Ebene, als es durch die Überprüfung von Ein- und Ausgaben des Programms möglich ist, überwacht. Zudem können zeitliche Abfolgen innerhalb des Programms überwacht werden. Im Gegensatz zum Testen kann die Laufzeitverifikation also jeden möglichen Ablauf des Programms auf Einhaltung der Spezifikation überprüfen. Das gilt allerdings nur, wenn der Ablauf des Programms auch tatsächlich während der Beobachtung auftritt. Außerdem bietet Laufzeitverifikation die Möglichkeit, die zu

überprüfende Eigenschaft des Systems auf einer höheren Abstraktionsebene als Ein- und Ausgabesequenzen zu spezifizieren.

Beim *Model-Checking* wiederum wird zuerst ein formales Modell des Programms bestimmt, das dann mit der formalen Spezifikation automatisiert verglichen wird. Dieses Vorgehen trifft Aussagen über das gesamte Programm und nicht nur über eine Ausführung des Programms, da alle möglichen Ausführungen des Programms betrachtet werden. Dies zeigt gleichzeitig auch einen der Nachteile des Model-Checkings auf. Da die heutigen Programme immer komplexer werden und damit auch die Anzahl der internen Zustände und Abhängigkeiten der Programme zunehmen, können sie häufig nicht mehr komplett durch Methoden des Model-Checkings untersucht werden. Dies führt dazu, dass nicht mehr das gesamte System, sondern nur noch Teilsysteme überwacht werden, wodurch die Vollständigkeit dieser Verifikationsmethode verloren geht.

Wie oben beschrieben, verwenden die Methoden der Laufzeitverifikation und des Model-Checkings formale Beschreibungen der zu überprüfenden Eigenschaften des Systems. Die Spezifikationen werden dabei häufig durch Spezifikationssprachen beschrieben, die auf verschiedenen formalen Logiken wie Linearzeit-Temporallogik oder Computation Tree Logic basieren. Die Auswertung der Logiken kann durch Büchi-Automaten erfolgen. Im Falle des Model-Checkings kann auch das Systemmodell durch Büchi-Automaten beschrieben werden.

Büchi-Automaten sind endliche Automaten, die unendliche Worte über einem endlichen Eingabealphabet akzeptieren. Dadurch bieten sie eine einfache Möglichkeit zur Beschreibung $\omega$-regulärer Sprachen aus unendlichen Wörtern und eignen sich daher besonders für die Software-Verifikation reaktiver Systeme, die fortlaufend mit ihrer Umwelt interagieren und für die weder ein Berechnungsbeginn noch ein Berechnungsende existiert. In diesem Kontext dienen Büchi-Automaten einerseits der Auswertung von Formeln der Linearzeit-Temporallogik [Var96; EH00], andererseits werden Büchi-Automaten verwendet, um die Korrektheit von Programm-Abstraktionen gegen das ursprüngliche Programm zu testen. Dabei werden sowohl das ursprüngliche Programm

als auch die berechnete Abstraktion in Automaten überführt, deren akzeptierte Sprachen dann verglichen werden.

Allgemein gilt jedoch für alle Anwendungen von Automaten, dass möglichst präzise und kompakte Beschreibungen der Automaten und dadurch auch ihrer akzeptierten Sprachen zu bevorzugen sind. Ein weiterer Grund für die Betrachtung minimierter Automaten ist der Effizienzgewinn in den auf ihnen aufbauenden Verfahren. Daher wurde in verschiedenen Arbeiten die Minimierung von Büchi-Automaten betrachtet [EH00; EWS01; EF10; Cle11; CM13]. Da jedoch bereits die Entscheidung, ob ein gegebener Büchi-Automat minimal ist, ein PSPACE-vollständiges Problem ist [JR91], werden meist nur partielle Minimierungstechniken betrachtet, die häufig auf Simulationsrelationen basieren.

Die in der Verifikation überprüften Eigenschaften von Systemen lassen sich häufig soweit zerlegen, bis sie nur noch von atomaren Aussagen über den aktuellen Zustand des Systems abhängig sind. Beispielsweise kann eine Brauprozesssteuerung, die den Brauprozess in einer Brauerei überwacht, verifiziert werden, indem verschiedene Kennwerte des Prozesses durch boolesche Ausdrücke überwacht werden, wie zum Beispiel:

- `mashTemp <= 78`
- `step == mashing`

Dies macht man sich bei der Spezifikation der zu überwachenden Eigenschaften zunutze und betrachtet die booleschen Ausdrücke als atomare Aussagen in der entsprechen formalen Logik. Das bedeutet, dass zum Beispiel Linearzeit-Temporallogik-Formeln über einer Menge $P$ der atomaren Propositionen gebildet werden. Bei der Umwandlung von Linearzeit-Temporallogik in Büchi-Automaten wird daher das Eingabealphabet der Automaten als $\Sigma = 2^P$ gewählt, da dies ermöglicht, jeden Zustand des überwachten Systems als eine Belegung der atomaren Propositionen zu interpretieren. Diese Struktur des Eingabealphabets machen wir uns zunutze und werden die symbolischen nichtdeterministischen Büchi-Automaten als Abwandlung der expliziten NBAs definieren, indem

die Transitionsbedingungen durch Formeln über den atomaren Propositionen statt explizit beschrieben werden. Die verwendeten Formeln beschreiben dabei die charakteristische Funktion der Menge der Eingabesymbole, die zu einem Zustandsübergang führen.

## 1.1 Aufbau der Arbeit

In dieser Arbeit werden wir zuerst die in [CM13] beschriebenen Verfahren zur Minimierung expliziter Büchi-Automaten in Kapitel 2 vorstellen und anschließend durch Konzepte aus dem symbolischen Model-Checking erweitern.

Da die in dieser Arbeit betrachteten Minimierungstechniken überwiegend von der Qualität der gewählten Simulationsrelationen abhängig sind, werden in den Abschnitten 2.2, 2.3 und 2.4 verschiedene Möglichkeiten der Bildung von Quasiordnungen auf den Zuständen des Automaten betrachtet, beispielsweise die $k$-Lookahead-Simulationen.

Unter anderem werden wir in Unterabschnitt 2.5.2 die folgenden Verfahren und Methoden auf expliziten NBAs betrachten. *Transitionsstutzung* entfernt Transitionen aus dem Automaten, wenn diese durch andere Transitionen beschrieben bzw. überdeckt werden, wobei jedoch die durch den NBA akzeptierte Sprache erhalten bleibt. Bei der *Quotientenbildung* wiederum werden Zustände des Ausgangsautomaten verschmolzen, wenn diese nicht voneinander unterscheidbar sind.

Nachdem wir die Minimierungstechniken auf expliziten NBAs vorgestellt haben, werden wir in Abschnitt 3.1 die bekannten expliziten NBAs durch symbolische NBAs erweitern. Abschnitt 3.2 beschreibt, wie die $k$-Lookahead-Simulationen auf die symbolischen NBAs übertragen werden. Darauf aufbauend werden die oben erwähnten Minimierungstechniken in Abschnitt 3.3 auf die symbolischen Simulationsrelationen angepasst.

In Abschnitt 3.4 weiten wir die symbolische Beschreibung des Eingabealphabets auch auf die Zustandsmenge der Automaten aus und definieren die vollsymbolischen NBAs, um die Simulationsrelationen und

Minimierungsverfahren effizient berechnen zu können. Auch auf dieses Automatenmodell werden in Abschnitt 3.5 die $k$-Lookahead-Simulationen übertragen, indem die Operatoren zur Berechnung der Relationen durch quantifizierte Aussagenlogik ausgedrückt werden. Ebenso werden in Abschnitt 3.6 die Minimierungstechniken durch quantifizierte Aussagenlogik ausgedrückt.

Die Beschreibung der Realisierung des entwickelten vollsymbolischen Minimierungsframeworks erfolgt in Kapitel 4. In Abschnitt 4.1 wird zuerst die Automaten- und Logikbibliothek RltlConv beschrieben, in deren Kontext das vollsymbolische Minimierungsframework realisiert wurde. Verschiedene alternative Ansätze für die Realisierung werden in Abschnitt 4.2 betrachtet. Die gewählte Realisierungsform wird dann in Abschnitt 4.3 aufgegriffen und die zur Realisierung gewählte Bibliothek beschrieben. Abschließend beschreiben wir in Abschnitt 4.4 die einzelnen Aspekte der Implementierung.

Am Ende wird in Kapitel 5 eine Zusammenfassung der im Rahmen dieser Arbeit entworfenen Verfahren zur Minimierung von NBAs und ihrer Realisierung geliefert. Zudem wird ein Ausblick auf mögliche weitere Untersuchungen gegeben.

# 2 Grundlagen der expliziten Automatenminimierung

In diesem Kapitel führen wir das von Clemente und Mayr in [CM13] beschriebene Framework zur Minimierung von nichtdeterministischen Büchi-Automaten ein und beschreiben kurz, wie dieses explizite nichtdeterministische Büchi-Automaten minimiert. Das Framework verwendet Quasiordnungen auf den Zuständen eines Automaten, um diesen zu minimieren. Eine zweistellige Relation $\sqsubseteq$ auf einer Menge $M$ heißt Quasiordnung, wenn sie reflexiv und transitiv ist. Die zur Minimierug verwendeten Ordnungen werden mithilfe eines spieltheoretischen Ansatzes bestimmt.

In Abschnitt 2.1 wiederholen wir zuerst die Definition eines nichtdeterministischen Büchi-Automaten. Danach stellen wir in den Abschnitten 2.2, 2.3 und 2.4 verschiedene Vorgehen vor, um Quasiordnungen auf den Zuständen eines Automaten zu berechnen. Darauf aufbauend werden dann in Abschnitt 2.5 verschiedene Techniken zur Minimierung von Automaten vorgestellt, die die berechneten Quasiordnungen verwenden. Darüber hinaus werden zwei Verfahren beschrieben, die diese Techniken kombiniert zur Anwendung bringen.

## 2.1 Nichtdeterministische Büchi-Automaten (NBAs)

Wir beginnen dieses Kapitel, indem wir die Definition eines nichtdeterministischen Büchi-Automaten wiederholen, dieser zählt zu den endlichen Automaten und akzeptiert unendliche Worte über einem endlichen Eingabealphabet.

**Definition 2.1** Nichtdeterministischer Büchi-Automat (NBA)
Ein *nichtdeterministischer Büchi-Automat (NBA)* ist definiert als ein 5-Tupel $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$, wobei

- $\Sigma$ ein endliches Alphabet,
- $Q$ eine endliche Menge von Zuständen,
- $I \subseteq Q$ eine Menge von *initialen* Zuständen,
- $\Delta \subseteq Q \times \Sigma \times Q$ die Transitionsrelation und
- $F \subseteq Q$ eine Menge von *akzeptierenden* Zuständen ist.

Ein NBA $\mathcal{A}$ kann auch als Graph repräsentiert werden. Abbildung 2.1 zeigt ein Beispiel eines einfachen NBAs $\mathcal{A}$ sowohl durch mathematische Strukturen beschrieben als auch seine grafische Repräsentation.

Eine Transition zwischen zwei Zuständen $q_i$ und $q_j$ des Automaten existiert, gdw. $(q_i, \sigma, q_j) \in \Delta$ gilt, wir schreiben dann auch $q_i \xrightarrow{\sigma} q_j \in \Delta$. Wir bezeichnen einen Zustand $q \in Q$ als *tot*, falls kein Pfad von einem initialen Zustand zu $q$ existiert oder von $q$ aus kein Pfad zu einem akzeptierenden Kreis in $\mathcal{A}$ existiert. In beiden Fällen ist der Zustand für die Akzeptanz eines gelesenen Wortes unerheblich und kann entfernt werden, ohne die akzeptierte Sprache zu ändern.

Aus Gründen der einfacheren Darstellung nehmen wir an, dass die betrachteten Automaten *vorwärts und rückwärts vollständig* sind: Es existieren für jeden Zustand $q_i \in Q$ und jedes Symbol $\sigma \in \Sigma$ zwei Zustände $q_h, q_j \in Q$, sodass $q_h \xrightarrow{\sigma} q_i \xrightarrow{\sigma} q_j$ gilt. Jeder Automat kann vervollständigt werden, indem zwei Zustände und für jeden Knoten maximal $|\Sigma|$ viele Transitionen hinzugefügt werden.

Ein *unendlicher Lauf* von $\mathcal{A}$ über einem unendlichen Wort $w = \sigma_0\sigma_1 \ldots \in \Sigma^\omega$ beginnt in einem beliebigen Zustand $q_0 \in Q$ und ist eine unendliche Folge von Transitionen $\rho = q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \cdots$. Mit $\rho[0..i] = q_0 \xrightarrow{\sigma_0} \cdots \xrightarrow{\sigma_{i-1}} q_i$ bezeichnen wir ein endliches Präfix von $\rho$ und $\rho[i..] = q_i \xrightarrow{\sigma_i} q_{i+1} \xrightarrow{\sigma_{i+1}} \cdots$ bezeichnet ein unendliches Suffix von $\rho$. *Endliche Läufe*, die in $q_0$ beginnen und in $q_m \in Q$ enden, sind entsprechend definiert. Weiterhin bezeichne $\inf(\rho)$ für einen Lauf $\rho$ die Menge

$$\Sigma = \{a, b, c, d\}$$
$$Q = \{q_0, q_1\}$$
$$I = \{q_0\}$$
$$\Delta = \begin{array}{l}\{(q_0, b, q_0), (q_0, d, q_0),\\ (q_0, c, q_1), (q_0, d, q_1),\\ (q_1, a, q_1), (q_1, b, q_1),\\ (q_1, c, q_1), (q_1, d, q_1)\}\end{array}$$
$$F = \{q_1\}$$

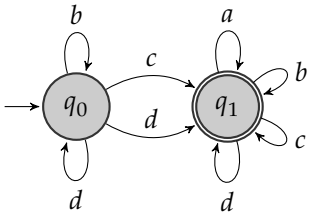



Abbildung 2.1: Beispiel für einen einfachen NBA $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ in mathematischer und grafischer Darstellung. Der Automat akzeptiert die Sprache $\{uvw \in \Sigma^\omega \mid u \in \{b, d\}^*, v \in \{c, d\}$ und $w \in \Sigma^\omega\}$.

der Zustände, die in $\rho$ unendlich oft besucht werden. Ein endlicher oder unendlicher Lauf ist *initial*, wenn er in einem Zustand $q_0 \in I$ beginnt. Ein unendlicher Lauf ist *fair*, gdw. $\inf(\rho) \cap F \neq \emptyset$ gilt, der Automat besucht unendlich oft einen akzeptierenden Zustand. Ein Lauf heißt *akzeptierend*, gdw. er unendlich, initial und fair ist. Der Automat $\mathcal{A}$ akzeptiert die Sprache

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^\omega \mid \mathcal{A} \text{ hat mind. einen akzeptierenden Lauf auf } w\}.$$

Seien $\mathcal{A} = (\Sigma, Q^\mathcal{A}, I^\mathcal{A}, \Delta^\mathcal{A}, F^\mathcal{A})$ und $\mathcal{B} = (\Sigma, Q^\mathcal{B}, I^\mathcal{B}, \Delta^\mathcal{B}, F^\mathcal{B})$ zwei beliebige NBAs, dann schreiben wir $\mathcal{A} \subseteq \mathcal{B}$ gdw. $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ und $\mathcal{A} \approx \mathcal{B}$ gdw. $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$.

## 2.2 Simulationen auf NBAs

In diesem Abschnitt beschreiben wir, wie Simulationsrelationen auf den Zuständen eines Automaten $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ als Ergebnis eines

Simulationsspiels zwischen zwei Spielern bestimmt werden können. Die Simulationsrelationen bilden dabei Quasiordnungen auf $Q \times Q$. Für die Berechnung der Simulationsrelationen findet der in [HKR02] vorgestellte spieltheoretische Ansatz Verwendung.

Die Simulationsbeziehung zweier Zustände $q_0$ und $\hat{q}_0$ kann formal als ein Spiel zwischen zwei Spielern beschrieben werden [EWS01]. In jeder Runde des Spiels platzieren die beiden Spieler zwei Pebble *Rot* und *Blau* auf zwei Zuständen, die sich nicht unterscheiden müssen. Am Anfang des Spiels, werden in Runde 0 die Pebble Rot auf Zustand $q_0$ und Blau auf $\hat{q}_0$ platziert.

**Definition 2.2** Grundspiel [EWS01]
Sei $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ ein beliebiger NBA und $q_0, \hat{q}_0$ zwei beliebige Zustände von $\mathcal{A}$. Das *Grundspiel* $G_{\mathcal{A}}(q_0, \hat{q}_0)$ auf $\mathcal{A}$ wird von zwei Spielern *Spoiler* und *Duplikator* rundenweise wie folgt gespielt:

Zu Beginn von Runde $i$ befinden sich die Pebble Rot auf $q_i$ und Blau auf $\hat{q}_i$.

1. Spoiler wählt eine Transition $q_i \xrightarrow{\sigma} q_{i+1} \in \Delta$ und bewegt Rot von $q_i$ nach $q_{i+1}$.
2. Duplikator muss eine passende Transition $\hat{q}_i \xrightarrow{\sigma} \hat{q}_{i+1} \in \Delta$ auswählen und bewegt Blau von $\hat{q}_i$ nach $\hat{q}_{i+1}$. Sollte keine passende Transition existieren, endet das Spiel und Spoiler gewinnt.
3. Die Folgekonfiguration ist dann $(q_{i+1}, \hat{q}_{i+1})$.

Eine passende Transition ist dadurch bestimmt, dass sie dieselbe Beschriftung wie die vorgegebene Transition trägt. Spoiler und Duplikator müssen also gleichzeitg dasselbe Eingabesymbol lesen.

Entweder endet das Grundspiel vorzeitig und Spoiler gewinnt das Spiel oder es läuft unendlich lange. Dabei werden zwei unendliche Läufe $\rho = q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \cdots$ und $\hat{\rho} = \hat{q}_0 \xrightarrow{\sigma_0} \hat{q}_1 \xrightarrow{\sigma_1} \cdots$ produziert, die sich

aus den gewählten Transitionen ergeben. Das Paar $(\rho, \hat{\rho})$ bezeichnen wir auch als *Resultat* des Spiels.

Die Siegbedingung für Duplikator ist von der gewünschten Simulation abhängig. Wir betrachten im Folgenden Bedingungen *direct* [DHW92], *delayed* [EWS01] und *fair* [HKR02]. Um die verschiedenen Ausprägungen der Simulationsbeziehung zu beschreiben, wird das Grundspiel auf drei verschiedene Arten erweitert.

**Definition 2.3** Simulationsspiele [EWS01]
Sei $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ ein beliebiger NBA, $(q_0, \hat{q}_0) \in Q \times Q$ und $(\rho, \hat{\rho})$ ein Resultat von $G_{\mathcal{A}}(q_0, \hat{q}_0)$ mit $\rho = q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \cdots$ und $\hat{\rho} = \hat{q}_0 \xrightarrow{\sigma_0} \hat{q}_1 \xrightarrow{\sigma_1} \cdots$, dann definieren wir folgende Simulationsspiele, die das Grundspiel $G_{\mathcal{A}}(q_0, \hat{q}_0)$ erweitern:

**Das *direkte Simulationsspiel*** (engl. *direct*) $G^{\mathrm{di}}_{\mathcal{A}}(q_0, \hat{q}_0)$ gewinnt Duplikator mit dem Resultat $(\rho, \hat{\rho})$, gdw. gilt:
Immer wenn Spoiler einen akzeptierenden Zustand besucht, muss auch Duplikator einen akzeptierenden Zustand besuchen.

**Das *verzögerte Simulationsspiel*** (engl. *delayed*) $G^{\mathrm{de}}_{\mathcal{A}}(q_0, \hat{q}_0)$ gewinnt Duplikator mit dem Resultat $(\rho, \hat{\rho})$, gdw. gilt:
Immer wenn Spoiler einen akzeptierenden Zustand besucht, muss auch Duplikator mit maximal endlicher Verzögerung einen akzeptierenden Zustand besuchen.

**Das *faire Simulationsspiel*** (engl. *fair*) $G^{\mathrm{f}}_{\mathcal{A}}(q_0, \hat{q}_0)$ gewinnt Duplikator mit dem Resultat $(\rho, \hat{\rho})$, gdw. gilt:
Es existieren unendlich viele $j \in \mathbb{N}$, für die $\hat{q}_j \in F$ gilt, oder es existieren nur endlich viele $i \in \mathbb{N}$, für die $q_i \in F$ gilt. In allen anderen Fällen gewinnt Spoiler.

Die bisher betrachteten Simulationsrelationen beschreiben alle eine Simulationbeziehung, die von den zukünftig noch zu lesenden Eingabesymbolen abhängt. Wir wollen außerdem auch eine Simulationsbeziehung betrachten, die von den bisher bereits gelesenen Eingabesymbolen

abhängt. Für die rückgerichtete Simulation wird das Grundspiel so angepasst, dass Spoiler und Duplikator die Transitionen in Rückrichtung wählen.

**Definition 2.4** Rückgerichtetes Grundspiel [CM13]
Sei $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ ein beliebiger NBA und $q_0, \hat{q}_0$ zwei beliebige Zustände von $\mathcal{A}$. Das *rückgerichtete Grundspiel* $G_{\mathcal{A}}(q_0, \hat{q}_0)$ auf $\mathcal{A}$ wird von zwei Spielern *Spoiler* und *Duplikator* rundenweise wie folgt gespielt:

1. Aus der Konfiguration $(q_i, \hat{q}_i)$ wählt Spoiler eine rückgerichtete Transition $q_{i+1} \xrightarrow{\sigma_{i+1}} q_i$.
2. Als Antwort darauf wählt Duplikator eine passende ebenfalls rückgerichtete Transition $\hat{q}_{i+1} \xrightarrow{\sigma_{i+1}} \hat{q}_i$.
3. Die Folgekonfiguration ist dann $(q_{i+1}, \hat{q}_{i+1})$.

Wie zuvor beschreibt $(\rho, \hat{\rho})$ das Resultat des Grundspiels mit $\rho = q_0 \xleftarrow{\sigma_0} q_1 \xleftarrow{\sigma_1} \cdots$ und $\hat{\rho} = \hat{q}_0 \xleftarrow{\sigma_0} \hat{q}_1 \xleftarrow{\sigma_1} \cdots$.

Auch das rückgerichtete Grundspiel wird zu einem rückgerichteten Simulationsspiel erweitert, dieses ähnelt dem direkten Simulationsspiel betrachtet jedoch neben den akzeptierenden auch die initialen Zustände des Automaten.

**Definition 2.5** Rückgerichtetes Simulationsspiel [EWS01]
Sei $\mathcal{A}$ ein beliebiger Büchi-Automat, $(q_0, \hat{q}_0) \in Q \times Q$ und $(\rho, \hat{\rho})$ ein Resultat von $G_{\mathcal{A}}(q_0, \hat{q}_0)$ mit $\rho = q_0 \xleftarrow{\sigma_0} q_1 \xleftarrow{\sigma_1} \cdots$ und $\hat{\rho} = \hat{q}_0 \xleftarrow{\sigma_0} \hat{q}_1 \xleftarrow{\sigma_1} \cdots$, dann erweitert das *rückgerichtete Simulationsspiel* (engl. *backwards*) $G^{\mathrm{bw}}_{\mathcal{A}}(q_0, \hat{q}_0)$ das rückgerichtete Grundspiel, sodass das Resultat $(\rho, \hat{\rho})$ einen Gewinn für Duplikator darstellt, gdw. gilt:
Immer wenn Spoiler einen akzeptierenden Zustand besucht, muss auch Duplikator einen akzeptierenden Zustand besuchen und immer wenn Spoiler einen initialen Zustand besucht, muss auch Duplikator einen initialen Zustand besuchen.

Das rückgerichtete Simulationsspiel gleicht dem direkten Simulationsspiel, betrachtet die Transitionen allerdings in umgekehrter Richtung. Die Betrachtung von rückgerichteten verzögerten oder fairen Simulationsspielen entfällt, da vor dem ersten besuchten Zustand keine weiteren Zustände existieren. Duplikator kann daher nicht verzögert einen initialen Zustand besuchen.

Jetzt können wir ausgehend von den Simulationsspielen aus Definition 2.3 und Definition 2.5 Siegbedingungen für Duplikator definieren, die nur noch von den Zuständen des Automaten abhängen, die in den Läufen $\rho$ bzw. $\hat{\rho}$ des Simulationsspiels besucht werden.

**Definition 2.6** $x$-Siegbedingungen [CM13]
Sei $x \in \{\mathrm{di}, \mathrm{bw}, \mathrm{de}, \mathrm{f}\}$, dann definieren wir die folgenden *Siegbedingungen* $C^x(\rho, \hat{\rho})$ für Duplikator hinsichtlich der vorwärtsgerichteten Simulationsspiele mit

$$\begin{aligned} C^{\mathrm{di}}(\rho, \hat{\rho}) &\iff \forall i \geq 0 : q_i \in F \implies \hat{q}_i \in F \\ C^{\mathrm{de}}(\rho, \hat{\rho}) &\iff \forall i \geq 0 : q_i \in F \implies \exists j \geq i : \hat{q}_j \in F \\ C^{\mathrm{f}}(\rho, \hat{\rho}) &\iff \text{wenn } \rho \text{ fair ist, dann ist auch } \hat{\rho} \text{ fair} \end{aligned}$$

Die Siegbedingung für das rückgerichtete Simulationsspiel hängt von den akzeptierenden und den initialen Zuständen ab.

$$C^{\mathrm{bw}}(\rho, \hat{\rho}) \iff \forall i \geq 0 : \begin{cases} q_i \in F & \implies \hat{q}_i \in F \text{ und} \\ q_i \in I & \implies \hat{q}_i \in I \end{cases}$$

Offensichtlich gilt

$$C^{\mathrm{di}}(\rho, \hat{\rho}) \implies C^{\mathrm{de}}(\rho, \hat{\rho}) \text{ und } C^{\mathrm{de}}(\rho, \hat{\rho}) \implies C^{\mathrm{f}}(\rho, \hat{\rho}).$$

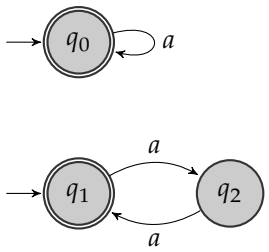
Wir sehen, dass die Simulationsspiele aus Definition 2.3 und Definition 2.5 und die Siegbedingungen in Zusammenhang stehen, sodass Duplikator das Simulationsspiel $G^x_{\mathcal{A}}(q_0, \hat{q}_0)$ gewinnt, gdw. $(\rho, \hat{\rho})$ das Resultat des Grundspiels ist und $C^x(\rho, \hat{\rho})$ erfüllt ist.

Auf Basis der Siegbedingungen können wir nun die verschiedenen Simulationsrelationen definieren.

**Definition 2.7** $x$-Simulationsrelationen [CM13]
Sei $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ ein beliebiger NBA und $(\rho, \hat{\rho})$ das Resultat des Grundspiels $G_{\mathcal{A}}(q_0, \hat{q}_0)$ auf $\mathcal{A}$. Eine *$x$-Simulationsrelation* $\sqsubseteq^x \subseteq Q \times Q$ ist eine Quasiordnung auf den Zuständen des Automaten $\mathcal{A}$. Der Zustand $\hat{q}$ simuliert $q$, gdw. $C^x(\rho, \hat{\rho})$ gilt, bzw. wenn Duplikator das $x$-Simulationsspiel $G^x_{\mathcal{A}}(q_0, \hat{q}_0)$ gewinnt. Wir schreiben dann auch $q \sqsubseteq^x \hat{q}$.

Aus Definition 2.6 folgt direkt $\sqsubseteq^{\mathrm{di}} \subseteq \sqsubseteq^{\mathrm{de}} \subseteq \sqsubseteq^{\mathrm{f}}$. Der Automat in Abbildung 2.2 akzeptiert die Sprache $\{a\}^\omega$ und veranschaulicht für $\Sigma = \{a\}$ den Zusammenhang zwischen direkter und verzögerter Simulation. Der Zustand $q_0$ simuliert die Zustände $q_1$ und $q_2$, da in dem direkten Simulationsspiel $G^{\mathrm{di}}_{\mathcal{A}}(q_1, q_0)$ Spoiler nur Transitionen mit $a$ von $q_1$ nach $q_2$ oder umgekehrt wählen kann. In beiden Fällen kann Duplikator die Transition $q_0 \xrightarrow{a} q_0$ wählen und sieht damit trivialerweise immer einen akzeptierenden Zustand, wenn Spoiler einen akzeptierenden Zustand sieht. Gleiches gilt für $G^{\mathrm{di}}_{\mathcal{A}}(q_2, q_0)$. Damit sieht Duplikator auch immer mit einer maximalen Verzögerung von 0 einen akzeptierenden Zustand, daraus folgt, dass $q_0$ die Zustände $q_1$ und $q_2$ auch verzögert simuliert. Umgekehrt simulieren $q_1$ und $q_2$ den Zustand $q_0$ nicht direkt aber verzögert. Sobald im direkten Simulationsspiel Spoiler $q_0$ und Duplikator $q_2$ betritt, wird die Siegbedingung für das direkte Simulationsspiel widerlegt. Im verzögerten Simulationsspiel hingegen reicht es aus, dass Duplikator von $q_2$ nach $q_1$ wechselt und damit mit einer maximalen Verzögerung von 1 einen akzeptierenden Zustand sieht.

Auch der Automat in Abbildung 2.3 akzeptiert die Sprache $\{a\}^\omega$, veranschaulicht dabei jedoch für $\Sigma = \{a, b\}$ den Zusammenhang zwischen verzögerter und fairer Simulation. Da $q_1$ der einzige akzeptierende Zustand ist, kann Spoiler nur in $q_1$ einen akzeptierenden Zustand sehen. Somit kann Duplikator beginnend in $q_1$ jeden Zug von Spoiler nachvollziehen, daraus folgt, dass $q_1$ alle Zustände verzögert simuliert. Da von $q_2$ aus kein akzeptierender Zustand erreichbar ist, simuliert $q_2$ nur




direkte Simulation

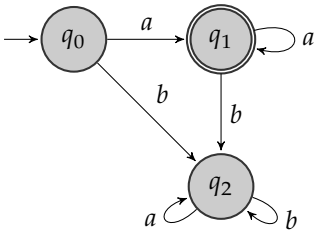
| $x \sqsubseteq^{\mathrm{di}} y$ | $q_0$ | $q_1$ | $q_2$ |
|---|---|---|---|
| $q_0$ | ✓ | × | × |
| $q_1$ | ✓ | ✓ | × |
| $q_2$ | ✓ | × | ✓ |

verzögerte Simulation

| $x \sqsubseteq^{\mathrm{de}} y$ | $q_0$ | $q_1$ | $q_2$ |
|---|---|---|---|
| $q_0$ | ✓ | ✓ | ✓ |
| $q_1$ | ✓ | ✓ | ✓ |
| $q_2$ | ✓ | ✓ | ✓ |

Abbildung 2.2: Der Automat akzeptiert die Sprache $\{a\}^\omega$. Für $\Sigma = \{a\}$ simuliert der Zustand $q_0$ die Zustände $q_1$ und $q_2$ direkt und verzögert. Die Zustände $q_1$ und $q_2$ simulieren $q_0$ jedoch nur verzögert und nicht direkt. Zudem simulieren sie sich gegenseitig verzögert.

sich selbst verzögert. Der Zustand $q_0$ simuliert den Zustand $q_1$ nicht verzögert, da Spoiler in dem verzögerten Simulationsspiel $G^{\mathrm{de}}_{\mathcal{A}}(q_1, q_0)$ nur die Transition $q_1 \xrightarrow{b} q_2$ wählen muss, damit Duplikator darauf nur mit der Wahl von $q_0 \xrightarrow{b} q_2$ reagieren kann. Dann hat Spoiler in $q_1$ allerdings einen akzeptierenden Zustand gesehen und Duplikator kann keinen akzeptierenden Zustand mehr sehen, wodurch die Siegbedingung der verzögerten Simulation widerlegt wird. Der Zustand $q_0$ simuliert $q_1$ jedoch fair, da der Teilpfad $q_1 \xrightarrow{b} q_2$ nicht Teil eines akzeptierenden Laufs sein kann. Spoiler liefert durch Wahl der Transition $q_1 \xrightarrow{b} q_2$ also keinen fairen Lauf mehr, wodurch auch Duplikator keinen fairen Lauf mehr liefern muss.




verzögerte Simulation

| $x \sqsubseteq^{\text{de}} y$ | $q_0$ | $q_1$ | $q_2$ |
|---|---|---|---|
| $q_0$ | ✓ | ✓ | × |
| $q_1$ | × | ✓ | × |
| $q_2$ | ✓ | ✓ | ✓ |

faire Simulation

| $x \sqsubseteq^{\text{f}} y$ | $q_0$ | $q_1$ | $q_2$ |
|---|---|---|---|
| $q_0$ | ✓ | ✓ | × |
| $q_1$ | ✓ | ✓ | × |
| $q_2$ | ✓ | ✓ | ✓ |

Abbildung 2.3: Der Automat akzeptiert die Sprache $\{a\}^\omega$. Für $\Sigma = \{a, b\}$ simuliert der Zustand $q_1$ die Zustände $q_0$ und $q_2$ verzögert und fair. Der Zustand $q_0$ simuliert $q_1$ jedoch nur fair und nicht verzögert. Der Zustand $q_2$ simuliert nur sich selbst.

## 2.3 Laufinklusionen auf NBAs

Simulationen sind effizient zu berechnen, ihr Optimierungspotential ist jedoch verglichen mit anderen Quasiordnungen beschränkt. Ein Beispiel für eine bessere Quasiordnung sind *Laufinklusionen*, die wir in diesem Abschnitt betrachten werden. Dafür werden wir die Definitionen der Simulationsspiele entsprechend anpassen.

Wir beschreiben, wie Laufinklusionen auf den Zuständen eines Automaten $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ als Ergebnis eines Simulationsspiels zwischen Spoiler und Duplikator bestimmt werden können. Während des Basisspiels der Simulationsrelationen produzieren die beiden Spieler abwechselnd die beiden Läufe $\rho$ und $\hat{\rho}$, indem sie jeweils nur auf einen einzelnen Zug des anderen Spielers reagieren. Um Laufinklusionen zu erhalten, erweitern wir das Grundspiel der Simulationsrelationen, sodass Spoiler zu Beginn des Spiels seinen gesamten Lauf $\rho$ festlegen muss, auf den Duplikator mit seinem Lauf $\hat{\rho}$ antwortet. Analog zu $x$-Simulationen bilden

$x$-Laufinklusionen eine Quasiordnung $\subseteq^x$ auf $Q \times Q$.

**Definition 2.8** $x$-Laufinklusionen [CM13]
Sei $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ ein beliebiger NBA. Für $x \in \{\mathrm{di}, \mathrm{de}, \mathrm{f}\}$ definieren wir, dass *$x$-Laufinklusion* zwischen $q$ und $\hat{q}$ besteht, gdw. für alle Wörter $w = \sigma_0\sigma_1 \ldots \in \Sigma^\omega$ und für alle unendlichen Läufe $\rho = q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \cdots$ des Automaten auf $w$ mit $q_0 = q$ ein unendlicher Lauf $\hat{\rho} = \hat{q}_0 \xrightarrow{\sigma_0} \hat{q}_1 \xrightarrow{\sigma_1} \cdots$ auf $w$ mit $\hat{q}_0 = \hat{q}$ existiert, sodass $C^x(\rho, \hat{\rho})$ gilt.

Genauso können wir die rückgerichtete Simulationsrelation auf den Zuständen des Automaten zur rückgerichteten Laufinklusion erweitern.

**Definition 2.9** Rückgerichtete Laufinklusion [CM13]
Sei $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ ein beliebiger NBA. Die *rückgerichtete Laufinklusion* $\subseteq^{\mathrm{bw}}$ besteht zwischen $q$ und $\hat{q}$ ($q \subseteq^{\mathrm{bw}} \hat{q}$), gdw. für alle endlichen Wörter $w = \sigma_0\sigma_1 \ldots \sigma_{m-1} \in \Sigma^*$ und jeden endlichen initialen Lauf $\rho = q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_{m-1}} q_m$, der in $q_m = q$ endet, ein endlicher, initialer Lauf $\hat{\rho} = \hat{q}_0 \xrightarrow{\sigma_0} \hat{q}_1 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_{m-1}} \hat{q}_m$ mit $\hat{q}_m = \hat{q}$ existiert, sodass $\forall i \geq 0 : q_i \in F \implies \hat{q}_i \in F$.

## 2.4 Lookahead-Simulationen auf NBAs

Mit den $x$-Laufinklusionen haben wir nun Quasiordnungen auf den Zuständen eines Automaten, die eine bessere Minimierung des Automaten erlauben. Allerdings können wir die Laufinklusionen nicht effizient berechnen, da sie über unendlichen Läufen der Automaten definiert sind und bereits das Entscheidungsproblem der Laufinklusionen für Quasiordnungen PSPACE-vollständig ist. Daher werden in [CM13] verschiedene alternative Simulationsrelationen vorgestellt.

Die *$k$-Pebble-Simulation* erweitert die Standard-Simulation um mehrere Pebble für Duplikator. Spoiler kontrolliert weiterhin genau einen Pebble, während Duplikator bis zu $k$ Pebble kontrollieren kann. Allerdings sind

auch $k$-Pebble-Simulationen nicht für eine effiziente Minimierung geeignet, da die Berechnung einer solchen auf einem NBA mit $n$ Zuständen dem Lösen eines Pebble-Spiels der Größe $n \cdot n^k$ entspricht.

Die *$k$-Step-Simulation* erweitert die Standard-Simulation, sodass beide Spieler Transitionsfolgen der Länge $k > 0$ wählen statt einzelner Transitionen. Duplikator erhält dadurch mehr Informationen über die folgenden Schritte von Spoiler, dies führt zu einer größeren Simulationsrelation. Jedoch variiert der Lookahead, den Duplikator über Spoilers Schritte hat, zwischen $k$, bei der Wahl der ersten Transition, und 1, bei der Wahl der letzten Transition. Duplikator steht also nicht für jede Entscheidung die gleiche Informationsmenge zur Verfügung.

Die *$k$-Continuous-Simulation* erweitert die Standard-Simulation, sodass Duplikator jederzeit über die nächsten $k$ Schritte von Spoiler informiert ist. Duplikator verfügt also jederzeit über einen Lookahead von $k$. Die Berechnung der Simulation auf einem NBA mit $n$ Zuständen und einem maximalen Ausgrad $d$ verlangt jedoch das Speichern von $n^2 \cdot d^{k-1}$ Konfigurationen.

### 2.4.1 $k$-Lookahead-Simulationen

In [CM13] wird die *$k$-Lookahead-Simulation* als optimale Alternative zwischen $k$-Step- und $k$-Continous-Simulation eingeführt. Sie erweitert die Standard-Simulation, sodass Duplikator in jeder Runde den Lookahead, den er über Spoilers Züge hat, beliebig zwischen 1 und $k$ wählen kann.

Das $k$-Lookahead-Grundspiel ändert das Grundspiel aus Definition 2.2 so ab, dass Spoiler in jeder Runde eine Transitionsfolge der Länge $k$ anstatt einzelner Transitionen wählt, während Duplikator in jeder Runde selbst über die Länge der gewählten Transitionsfolge entscheiden kann.

**Definition 2.10** $k$-Lookahead-Grundspiel [CM13]
Sei $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ ein beliebiger NBA und $q_0$ und $\hat{q}_0$ zwei beliebige Zustände von $\mathcal{A}$. Das *$k$-Lookahead-Grundspiel* $G^k_{\mathcal{A}}(q_0, \hat{q}_0)$ wird wie folgt von zwei Spielern gespielt:

Zu Beginn von Runde $i$ befindet sich das Spiel in der Konfiguration $(q_i, \hat{q}_i)$ und Spoiler beginnt:

1. Spoiler wählt eine Folge von $k$ zusammenhängenden Transitionen $q_i \xrightarrow{\sigma_i} q_{i+1} \xrightarrow{\sigma_{i+1}} \cdots \xrightarrow{\sigma_{i+k-1}} q_{i+k}$.
2. Duplikator wählt eine Zahl $1 \leq m \leq k$ aus und antwortet mit einer passenden Folge von $m$ Transitionen $\hat{q}_i \xrightarrow{\sigma_i} \hat{q}_{i+1} \xrightarrow{\sigma_{i+1}} \cdots \xrightarrow{\sigma_{i+m-1}} \hat{q}_{i+m}$.
3. Die verbleibenden Transitionen von Spoiler werden ignoriert und die neue Konfiguration ist $(q_{i+m}, \hat{q}_{i+m})$.

Auf diese Weise wird ein Resultat $(\rho, \hat{\rho})$ produziert.

Das rückgerichtete $k$-Lookahead-Grundspiel wird entsprechend mit rückwärtsgerichteten Transitionsfolgen definiert und ändert das Grundspiel aus Definition 2.4 so ab, dass Spoiler und Duplikator rückwärtsgerichtete Transitionsfolgen statt einzelner Transitionen wählen.

**Definition 2.11** Rückgerichtetes $k$-Lookahead-Grundspiel [CM13]
Sei $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ ein beliebiger NBA und $q_0$ und $\hat{q}_0$ zwei beliebige Zustände von $\mathcal{A}$. Das *rückgerichtete $k$-Lookahead-Grundspiel* wird wie folgt von zwei Spielern gespielt:

Zu Beginn von Runde $i$ befindet sich das Spiel in der Konfiguration $(q_i, \hat{q}_i)$ und Spoiler beginnt:

1. Spoiler wählt eine Folge von $k$ in Rückrichtung zusammenhängenden Transitionen $q_i \xleftarrow{\sigma_i} q_{i+1} \xleftarrow{\sigma_{i+1}} \cdots \xleftarrow{\sigma_{i+k-1}} q_{i+k}$.
2. Duplikator wählt eine Zahl $1 \leq m \leq k$ aus und antwortet mit einer passenden Folge von $m$ rückwärtsgerichteten Transitionen $\hat{q}_i \xleftarrow{\sigma_i} \hat{q}_{i+1} \xleftarrow{\sigma_{i+1}} \cdots \xleftarrow{\sigma_{i+m-1}} \hat{q}_{i+m}$.

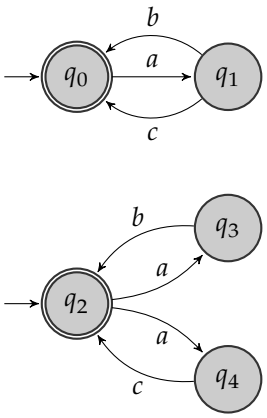
3. Die verbleibenden Transitionen von Spoiler werden ignoriert und die neue Konfiguration ist $(q_{i+m}, \hat{q}_{i+m})$.

Da wir in Abschnitt 2.2 festgestellt haben, dass die Definitionen 2.3, 2.5 und 2.6 jeweils nur von den besuchten Zuständen des Resultats der Grundspiele abhängen, können diese Definitionen unverändert auch auf die $k$-Lookahead-Simulationen übertragen werden.

**Definition 2.12** $x$-$k$-Lookahead-Simulationen [CM13]
Sei $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ ein beliebiger NBA und $(\rho, \hat{\rho})$ das Resultat eines $k$-Lookahead Grundspiels auf $\mathcal{A}$. Zwei Zustände $q_0$ und $\hat{q}_0$ stehen in *$x$-$k$-Lookahead-Simulation*, gdw. $C^x(\rho, \hat{\rho})$ gilt, bzw. wenn Duplikator eine $x$-Gewinnstrategie in dem $k$-Lookahead-Grundspiel hat. Wir schreiben dann $q_0 \sqsubseteq^{k\text{-}x} \hat{q}_0$.

Die $k$-Lookahead-Simulationen $\sqsubseteq^{k\text{-}x}$ sind nicht transitiv, daher verwenden wir die transitive Hülle $\preceq^{k\text{-}x}$ der $k$-Lookahead-Simulationen, da diese Quasiordnungen auf $Q \times Q$ bilden.

Der Automat in Abbildung 2.4 akzeptiert die Sprache $\{w_0 w_1 w_2 \ldots \in \Sigma^\omega | \forall i \geq 0 : w_i = ab \vee w_i = ac\}$. Außerdem lässt sich an ihm der Unterschied zwischen direkter Simulation und direkter 2-Lookahead-Simulation veranschaulichen. Für $\Sigma = \{a, b, c\}$ simuliert der Zustand $q_0$ den Zustand $q_2$ direkt, da Spoiler aus $q_2$ keinen Lauf angeben kann, den Duplikator aus $q_0$ nicht simulieren könnte. Die Umkehrung $q_0 \sqsubseteq^{\text{di}} q_2$ gilt nicht, da Spoiler sobald er sich in $q_1$ befindet abhängig davon, ob sich Duplikator in $q_3$ oder $q_4$ befindet, die Transition wählt, deren Transitionsbedingung Duplikator gerade nicht wählen kann. Der Zustand $q_1$ simuliert die Zustände $q_3$ und $q_4$ direkt, auch hier gilt die Umkehrung nicht. Betrachtet man jedoch die direkte 2-Lookahead-Simulation gilt zusätzlich auch $q_0 \sqsubseteq^{2\text{-di}} q_2$, da Spoiler in jeder Runde des direkten 2-Lookahead-Simulationsspiels eine Transitionsfolge der Länge 2 liefern muss. Duplikator kann also von $q_2$ aus die Transition nach $q_3$ oder $q_4$ wählen, für die eine passende Folgetransition für Spoilers zweite Transition existiert. Es




direkte Simulation

| $x \sqsubseteq^{\text{di}} y$ | $q_0$ | $q_1$ | $q_2$ | $q_3$ | $q_4$ |
|---|---|---|---|---|---|
| $q_0$ | ✓ | × | × | × | × |
| $q_1$ | × | ✓ | × | × | × |
| $q_2$ | ✓ | × | ✓ | × | × |
| $q_3$ | × | ✓ | × | ✓ | × |
| $q_4$ | × | ✓ | × | × | ✓ |

direkte 2-Lookahead-Simulation

| $x \sqsubseteq^{\text{2-di}} y$ | $q_0$ | $q_1$ | $q_2$ | $q_3$ | $q_4$ |
|---|---|---|---|---|---|
| $q_0$ | ✓ | × | ✓ | × | × |
| $q_1$ | × | ✓ | × | × | × |
| $q_2$ | ✓ | × | ✓ | × | × |
| $q_3$ | × | ✓ | × | ✓ | × |
| $q_4$ | × | ✓ | × | × | ✓ |

Abbildung 2.4: Der Automat akzeptiert die Sprache $\{w_0w_1w_2 \ldots \in \Sigma^\omega \mid \forall i \geq 0 : w_i = ab \vee w_i = ac\}$. Für $\Sigma = \{a, b, c\}$ simuliert der Zustand $q_0$ den Zustand $q_2$ direkt, umgekehrt gilt $q_0 \sqsubseteq^{\text{di}} q_2$ nicht. Der Zustand $q_1$ simuliert die Zustände $q_3$ und $q_4$ direkt, auch hier gilt die Umkehrung nicht. Betrachtet man jedoch die direkte 2-Lookahead-Simulation, gilt zusätzlich auch $q_0 \sqsubseteq^{\text{2-di}} q_2$.

gilt jedoch weiterhin nicht $q_1 \sqsubseteq^{\text{2-di}} q_3$ oder $q_1 \sqsubseteq^{\text{2-di}} q_4$, da Spoiler wie bereits zuvor die gerade nicht passende Transition als erste Transition wählen kann.

### 2.4.2 Berechnung der $k$-Lookahead-Simulationen

In [EWS01] werden die Simulationsrelationen durch Lösen eines Paritätsspiels bestimmt, das auf einem Graphen gespielt wird, der sich aus den Zuständen des Automaten und einer maximalen Parität von zwei

ergibt. Für das Lösen der $k$-Lookahead-Simulationen würde das einem Paritätsspielgraphen mit folgenden Knoten und Kanten entsprechen:

1. Für jedes Paar aus $Q \times Q$ und jede der drei Paritäten $p \in \{0, 1, 2\}$ einen Knoten $(q, \hat{q}, p)$.
2. Für jeden Pfad der Länge $k$ von $q_0 \in Q$ aus, jedes $\hat{q} \in Q$ und jede Parität $p \in \{0, 1, 2\}$ einen Knoten $(q_0 \rightarrow^k q_k, \hat{q}, p)$.
3. Zwischen diesen Knoten müssen dann passende Kanten konstruiert werden, sodass eine korrekte Aussage über die Simulationsbeziehung getroffen wird.

Der entstehende Paritätsspielgraph wäre auch für kleine $k$ und kleine Automaten bereits sehr groß. In [CM13] wird alternativ dazu auch ein fixpunktbasierter Ansatz vorgestellt, den wir im Weiteren verfolgen werden, da er für große Automaten und das weitere Vorgehen besser geeignet erscheint.

Sei $x \in \{\mathrm{di}, \mathrm{bw}, \mathrm{de}, \mathrm{f}\}$ und $k > 0$. Dann gilt $q \sqsubseteq^{k\text{-}x} \hat{q}$, gdw. Duplikator eine Gewinnstrategie in dem entsprechenden Simulationsspiel hat. Damit Duplikator ein Simulationsspiel gewinnt, muss er die Siegbedingung des Simulationsspiels erfüllen oder das Spiel in eine Konfiguration überführen, für die bekannt ist, dass Duplikator das entsprechende Spiel aus dieser Konfiguration heraus gewinnt. Daher kann $k$-Lookahead-Simulation als die größte Menge $X \subseteq Q \times Q$ beschrieben werden, die abgeschlossen bezüglich eines monotonen Vorgänger-Operators ist. Um die Darstellung für die verzögerte Simulation zu vereinfachen, wird nicht $\sqsubseteq^{\mathrm{k\text{-}x}}$ selbst berechnet, sondern die komplementäre Relation $W^x = (Q \times Q) \setminus \sqsubseteq^{\mathrm{k\text{-}x}}$, die die Konfigurationen des Simulationsspiels angibt, aus denen Spoiler gewinnt.

### Direkte und rückgerichtete $k$-Lookahead-Simulation

Für die direkte und die rückgerichtete $k$-Lookahead-Simulation lassen sich einfach Vorgänger-Operatoren bestimmen, die nur von einer Menge

von Konfigurationen des zugrunde liegenden Simulationsspiels abhängig sind.

Wir betrachten zuerst den Vorgänger-Operator $\mathrm{CPre}^{\mathrm{di}}(X)$ für eine beliebige Menge $X \subseteq Q \times Q$ von Konfigurationen:

**Definition 2.13** Direkter Vorgänger-Operator [CM13]
Sei $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ ein beliebiger NBA und $X$ eine beliebige Menge von Konfigurationen im direkten Simulationsspiel auf $\mathcal{A}$, dann ist der *direkte Vorgänger-Operator* $\mathrm{CPre}^{\mathrm{di}} : 2^{(Q\times Q)} \to 2^{(Q\times Q)}$ definiert durch:

$$\begin{aligned}\mathrm{CPre}^{\mathrm{di}}(X) = \Big\{ (q_0, \hat{q}_0) \;\Big|\; \exists \left( q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_{k-1}} q_k \right) : \\ \forall 0 < m \leq k : \\ \forall \left( \hat{q}_0 \xrightarrow{\sigma_0} \hat{q}_1 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_{m-1}} \hat{q}_m \right) : \\ \left( \exists 0 \leq j \leq m : q_j \in F \wedge \hat{q}_j \notin F \right) \\ \vee (q_m, \hat{q}_m) \in X \Big\}\end{aligned}$$

Es gilt $(q, \hat{q}) \in \mathrm{CPre}^{\mathrm{di}}(X)$, gdw. Spoiler ausgehend von der Konfiguration $(q, \hat{q})$ innerhalb einer Runde des direkten $k$-Lookahead-Simulationsspiels entweder die Siegbedingung für das direkte Simulationsspiel verletzt oder das Spiel in eine Konfiguration aus $X$ zwingen kann.

Für die rückgerichtete $k$-Lookahead-Simulation wird der Vorgänger-Operator $\mathrm{CPre}^{\mathrm{bw}}(X)$ analog dazu definiert, mit den Abwandlungen, dass die Transitionen umgekehrt werden und zusätzlich initiale Zustände betrachtet werden.

**Definition 2.14** Rückgerichteter Vorgänger-Operator [CM13]
Sei $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ ein beliebiger NBA und $X$ eine beliebige Menge von Konfigurationen des rückgerichteten Simulationsspiels auf $\mathcal{A}$, dann ist der *rückgerichtete Vorgänger-Operator* $\mathrm{CPre}^{\mathrm{bw}} : 2^{(Q\times Q)} \to 2^{(Q\times Q)}$

definiert durch:

$$\begin{aligned}\mathrm{CPre}^{\mathrm{bw}}(X) = \Big\{ (q_0, \hat{q}_0) \;\Big|\; \exists \Big( q_0 \xleftarrow{\sigma_0} q_1 \xleftarrow{\sigma_1} \cdots \xleftarrow{\sigma_{k-1}} q_k \Big) : \\ \forall 0 < m \leq k : \\ \forall \Big( \hat{q}_0 \xleftarrow{\sigma_0} \hat{q}_1 \xleftarrow{\sigma_1} \cdots \xleftarrow{\sigma_{m-1}} \hat{q}_m \Big) : \\ \Big( \exists 0 \leq j \leq m : q_j \in F \wedge \hat{q}_j \notin F \Big) \\ \vee \Big( \exists 0 \leq j \leq m : q_j \in I \wedge \hat{q}_j \notin I \Big) \\ \vee (q_m, \hat{q}_m) \in X \Big\}\end{aligned}$$

Es gilt $(q, \hat{q}) \in \mathrm{CPre}^{\mathrm{bw}}(X)$, gdw. Spoiler ausgehend von der Konfiguration $(q, \hat{q})$ entweder die Siegbedingung für das rückgerichtete $k$-Lookahead-Simulationsspiel verletzt oder das Spiel in eine Konfiguration aus $X$ zwingen kann.

**Bemerkung 1**
Die Definition von $\mathrm{CPre}^{x}(X)$ setzt voraus, dass der Automat vorwärts und rückwärts vollständig ist, andernfalls würde Spoiler fälschlicherweise das Simulationsspiel verlieren, wenn er nur $k' < k$ Transitionen wählen kann.

Für $X = \emptyset$ liefert $\mathrm{CPre}^{x}(X)$ die Menge der Konfigurationen, aus denen Spoiler das zugrunde liegende Simulationsspiel $G^{x}_{\mathcal{A}}$ in maximal einer Runde gewinnt. Daraus folgt, dass Spoiler gewinnt, sollte er das Spiel in eine dieser Konfigurationen zwingen können.

Damit ergibt sich für $x \in \{\mathrm{di}, \mathrm{bw}\}$ die Fixpunktgleichung

$$W^{x} = \mu(W \mapsto \mathrm{CPre}^{x}(W)).$$

Beide Operatoren sind so aufgebaut, dass Spoiler einen Pfad der Länge $k$ finden muss, sodass für alle Pfade der Länge $0 < m \leq k$, die Duplikator finden kann, die Siegbedingung des Simulationsspiels verletzt wird oder das Simulationsspiel in eine der Konfigurationen aus $X$ überführt wird.

### Verzögerte und faire $k$-Lookahead-Simulation

Für die verzögerte und faire $k$-Lookahead-Simulation wird der grundlegende Aufbau von $\mathrm{CPre}^{\mathrm{di}}$ und $\mathrm{CPre}^{\mathrm{bw}}$ übernommen. Der Vorgänger-Operator muss jedoch um zwei Argumente zu $\mathrm{CPre}(X, Y, Z)$ erweitert werden.

Intuitiv ist eine Konfiguration $(q, \hat{q}) \in \mathrm{CPre}(X, Y, Z)$, gdw. Spoiler eine Transitionsfolge im zugehörigen Simulationsspiel wählen kann, sodass für jede Antwort Duplikators mindestens eine der folgenden Bedingungen gilt:

1. Spoiler besucht mindestens einen akzeptierenden Zustand, während Duplikator keinen akzeptierenden Zustand besucht und das Spiel befindet sich danach in einer Konfiguration aus $X$.

2. Duplikator besucht keinen akzeptierenden Zustand und das Spiel befindet sich danach in $Y$.

3. Das Spiel befindet sich nach der Runde in $Z$.

**Definition 2.15** Vorgänger-Operator [CM13]
Sei $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ ein beliebiger NBA und $X$, $Y$ und $Z$ beliebige Mengen von Konfigurationen eines verzögerten oder fairen Simulationsspiels auf $\mathcal{A}$, dann ist der *Vorgänger-Operator* $\mathrm{CPre} : 2^{Q^2} \times 2^{Q^2} \times 2^{Q^2} \to 2^{Q^2}$

definiert durch:

$$\begin{aligned}\mathrm{CPre}(X,Y,Z) = \Bigg\{(q_0,\hat{q}_0) \;\Bigg|\; \exists \left(q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_{k-1}} q_k\right) : \\ \forall 0 < m \leq k : \\ \forall \left(\hat{q}_0 \xrightarrow{\sigma_0} \hat{q}_1 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_{m-1}} \hat{q}_m\right) : \\ \left(\begin{array}{l} \exists 0 \leq j \leq m : q_j \in F \\ \wedge \forall 0 \leq j \leq m : \hat{q}_j \notin F \\ \wedge (q_m,\hat{q}_m) \in X \end{array}\right) \\ \vee \left(\begin{array}{l} \forall 0 \leq j \leq m : \hat{q}_j \notin F \\ \wedge (q_m,\hat{q}_m) \in Y \end{array}\right) \\ \vee \quad (q_m,\hat{q}_m) \in Z\Bigg\}\end{aligned}$$

Für das faire Simulationsspiel gilt: Spoiler gewinnt, gdw. er bis auf endlich viele Runden unendlich viele akzeptierende Zustände besucht, während Duplikator keine akzeptierenden Zustände besucht. Wie zuvor gewinnt Spoiler, wenn er das Simulationsspiel in eine Konfiguration überführen kann, für die bekannt ist, dass Spoiler gewinnt. Daraus ergibt sich die Bedingung $(q_m,\hat{q}_m) \in Z$ und die Formulierung als kleinster Fixpunkt einer Funktion. Die zweite Bedingung mit $(q_m,\hat{q}_m) \in Y$ ergibt sich, da Duplikator das faire Simulationsspiel verliert, sollte er nur endlich viele akzeptierende Zustände sehen. Solange Duplikator also keine akzeptierenden Zustände sieht, wird das Simulationsspiel in der aktuellen Runde weder für Spoiler noch für Duplikator entschieden. Die Entscheidung wird also erst in einer späteren Runde fallen. Die erste Bedingung mit $(q_m,\hat{q}_m) \in X$ ergibt sich, da Spoiler unendlich viele akzeptierende Zustände besuchen muss, um das Simulationsspiel zu gewinnen, während Duplikator keine akzeptierenden Zustände sieht.

Daher ergibt sich die Fixpunktgleichung

$$W^{\mathrm{f}} = \mu(Z \mapsto \nu(X \mapsto \mu(Y \mapsto \mathrm{CPre}(X,Y,Z)))).$$

Spoiler gewinnt das verzögerte Simulationsspiel, wenn nach endlich vielen Runden die folgenden beiden Bedingungen erfüllt sind:

1. Spoiler besucht einen akzeptierenden Zustand und
2. Spoiler verhindert, dass Duplikator in der Zukunft einen akzeptierenden Zustand besucht.

Daraus lassen sich die beiden folgenden Operatoren ableiten.

$$\mathrm{CPre}^1(X,Y) := \mathrm{CPre}(X,Y,Y) \text{ und } \mathrm{CPre}^2(X,Y) := \mathrm{CPre}(X,X,Y)$$

Eine Konfiguration $(q_i, \hat{q}_i)$ ist in $\mathrm{CPre}^2(X,Y)$, wenn die Folgekonfiguration nach einer Runde des verzögerten Simulationsspiels in $Y$ ist oder wenn Spoiler das Spiel in eine Konfiguration aus $X$ überführen kann, ohne dass Duplikator einen akzeptierenden Zustand besucht. Der Operator $\mathrm{CPre}^2$ berechnet also die Menge der Konfigurationen, aus denen Duplikator keine akzeptierenden Zustände sehen kann.

Der Operator $\mathrm{CPre}^1(X,Y)$ wiederum berechnet die Menge der Konfigurationen, für die gilt, dass Spoiler das Simulationsspiel in eine Konfiguration aus $Y$ überführen kann und damit gewinnt oder zumindest einen akzeptierenden Zustand besucht, während Duplikator keinen akzeptierenden Zustand besucht und sich das Spiel danach in einer Konfiguration aus $X$ befindet.

Kombiniert man beide Operatoren, ergibt sich die Fixpunktgleichung

$$W^{\mathrm{de}} = \mu(W \mapsto \mathrm{CPre}^1(\nu(X \mapsto \mathrm{CPre}^2(X,W)),W)).$$

## 2.5 Minimierung von NBAs

In den vorhergehenden Abschnitten wurden verschiedene Quasiordnungen und die Möglichkeiten ihrer Berechnung beschrieben. In diesem Abschnitt soll nun dargestellt werden, wie diese zur Minimierung von Automaten verwendet werden können.

Bevor wir jedoch beschreiben, wie Automaten minimiert werden können, beschreiben wir noch einmal, was Minimierung bedeutet. Minimierung eines Automaten bedeutet eine Reduktion der Größe des Automaten bei gleichzeitiger Erhaltung der durch den Automaten akzeptierten Sprache. Die Reduktion kann dabei durch Manipulation zweier Kennwerte des Automaten erreicht werden:

- Entfernen von Zuständen des Automaten und
- Entfernen von Transitionen zwischen Zuständen des Automaten.

Dabei muss jedoch immer der Erhalt der akzeptierten Sprache im Vordergrund stehen.

Wie bereits erwähnt, beschreiben wir durch $\preceq^{k\text{-}x}$ die transitiven Hüllen der $k$-Lookahead-Simulationen $\sqsubseteq^{k\text{-}x}$. Außerdem bezeichnet $\prec^{k\text{-}x} = \preceq^{k\text{-}x} \setminus (\prec^{k\text{-}x})^{-1}$ bzw. $\sqsubset^{x} = \sqsubseteq^{x} \setminus (\sqsubseteq^{x})^{-1}$ die asymmetrische Restriktion der entsprechenden Relation.

### 2.5.1 Reduktion durch Entfernen toter Zustände

Die erste recht einfache Methode zur Reduktion von Zuständen ist das Entfernen der toten Zustände aus dem Automaten. Dies kann ohne besondere Beachtung der akzeptierten Sprache geschehen, da die toten Zustände eines Automaten nach ihrer Definition nicht in akzeptierenden Pfaden des Automaten vorkommen und somit auch nicht die akzeptierte Sprache beeinflussen.

Abbildung 2.5 zeigt einen Automaten mit vier Zuständen, von denen zwei Zustände $q_0$ und $q_1$ lebendig sind, während die Zustände $q_2$ und $q_3$ tot sind und daher gestrichelt dargestellt werden. Der Zustand $q_2$ ist tot, da er nicht von einem initialen Zustand aus erreichbar ist und somit nicht Teil eines akzeptierenden Laufs sein kann. Der zweite tote Zustand $q_3$ ist von einem initialen Zustand aus erreichbar, in diesem Fall von $q_0$ aus, und selbst akzeptierend. Jeder Lauf, der $q_3$ erreicht, muss jedoch in $q_3$ enden und kann somit nicht akzeptierend sein. Da $q_2$ und $q_3$ tot sind,

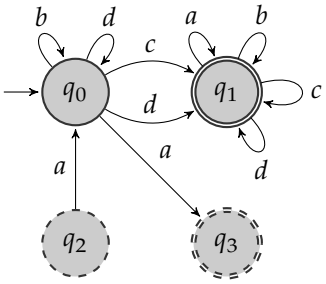



Abbildung 2.5: Grafische Repräsentation eines NBAs mit vier Zuständen. Die Zustände $q_0$ und $q_1$ sind lebendig, während die Zustände $q_2$ und $q_3$ tot sind. Die toten Zustände sind gestrichelt dargestellt.

können beide Zustände problemlos entfernt werden und das Ergebnis ist der bereits aus Abbildung 2.1 bekannte Automat, der dieselbe Sprache wie der Ausgangsautomat akzeptiert.

### 2.5.2 Reduktion durch Transitionsstutzung

Um einen NBA zu minimieren, bietet sich neben dem Entfernen von toten Zuständen auch an, die Anzahl der Transitionen des Automaten durch Transitionsstutzung zu verringern. Dies führt zu einem präziser beschriebenen Automaten, da für die Akzeptanz eines Wortes überflüssige Transitionen entfernt werden.

Die Idee der Transitionsstutzung ist es, eine Transition aus dem Automaten zu entfernen, wenn diese durch eine andere Transition überdeckt wird.

**Definition 2.16** Gestutzter Automat [CM13]
Sei $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ ein NBA und $P$ eine transitive und asymmetrische Relation auf $\Delta$. Der *gestutzte Automat* (engl. pruned automaton) ist dann definiert als $\text{Prune}(\mathcal{A}, P) := (\Sigma, Q, I, \Delta', F)$ mit

$$\Delta' = \{(q_i, \sigma, q_j) \in \Delta \mid \nexists (q_i', \sigma', q_j') \in \Delta : (q_i, \sigma, q_j)\ P\ (q_i', \sigma', q_j')\}.$$

Da wir die durch den Automaten akzeptierte Sprache durch die Transitionsstutzung erhalten wollen, muss die verwendete Pruning-Relation eine bestimmte Struktur aufweisen.

**Definition 2.17** Good for Pruning (GFP) [CM13]
Sei $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ ein beliebiger NBA. Eine Relation $P$ auf $\Delta$ ist für die Transitionsstutzung geeignet (engl. *Good for Pruning (GFP)*), gdw. $\text{Prune}(\mathcal{A}, P) \approx \mathcal{A}$ gilt.

Um nur geeignete Pruning-Relation zu erhalten, beschreiben wir nun eine Konstruktionsmethode für Pruning-Relationen, die für geeignete Eingaben GFP-Relationen produziert.

**Definition 2.18** Pruning-Relation [CM13]
Seien $R_b$ und $R_f$ zwei Binärrelationen auf $Q$, dann definieren wir die Relation $P(R_b, R_f) \subseteq \Delta \times \Delta$ mit

$$P(R_b, R_f) = \left\{ \left( (q_i, \sigma, q_j), (\hat{q}_i, \sigma, \hat{q}_j) \right) \,\middle|\, q_i \; R_b \; \hat{q}_i \wedge q_j \; R_f \; \hat{q}_j \right\}$$

Eine Transition $(\hat{q}_i, \sigma, \hat{q}_j)$ überdeckt nach obiger Definition eine Transition $(q_i, \sigma, q_j)$, gdw. sowohl die Ausgangszustände $q_i$ und $\hat{q}_i$ in Rückrichtung in einer rückgerichteten Simulation $R_b$ und die Zielzustände $q_j$ und $\hat{q}_j$ in Vorwärtsrichtung in einer Simulation $R_f$ stehen.

**Bemerkung 2** [CM13]
Für $R_b \in \{id, \sqsubset^{\text{bw}}, \subset^{\text{bw}}\}$ und $R_f \in \{id, \sqsubset^{\text{di}}, \subset^{\text{di}}, \sqsubset^{\text{de}}, \sqsubset^{\text{f}}\}$ induzieren $R_b$ und $R_f$ nach folgender Tabelle eine Relation $P(R_b, R_f)$, die GFP ist:

| $R_b \backslash R_f$ | $id$ | $\sqsubset^{\text{di}}$ | $\subset^{\text{di}}$ | $\sqsubset^{\text{de}}$ | $\sqsubset^{\text{f}}$ |
|---|---|---|---|---|---|
| $id$ | × | ✓ | ✓ | × | × |
| $\sqsubset^{\text{bw}}$ | ✓ | ✓ | ✓ | × | × |
| $\subset^{\text{bw}}$ | ✓ | ✓ | × | × | × |

Nähere Informationen und Beweise finden sich in [CM13].

Darüber hinaus ist auch die folgende Relation GFP [SB00]

$$R_t(\subset^{\mathrm{f}}) = P(id, \sqsubset^{\mathrm{di}}) \cup \{((q_i, \sigma_i, q_j), (q_i, \sigma_i, \hat{q}_j)) \mid (q_i, \sigma_i, \hat{q}_j) \text{ ist transient und } q_j \subset^{\mathrm{f}} \hat{q}_j\}.$$

Die Relation $R_t(\subset^{\mathrm{f}})$ hängt also von den transienten Transitionen des Automaten ab, das sind die Transitionen, die zwischen den starken Zusammenhangskomponenten des Automaten verlaufen und daher in einem Pfad nur einmal vorkommen können.

Da die $k$-Lookahead-Simulationen größenmäßig zwischen den entsprechenden Simulationen und Laufinklusionen liegen, überträgt sich die Eigenschaft GFP der Relationen aus Bemerkung 2 auch auf die Pruning-Relationen, wenn $k$-Lookahead-Simulationen statt Laufinklusionen als Grundlage dienen.

Abbildung 2.6 zeigt ein Beispiel für die Transitionsstutzung auf einem NBA. Vor der Stutzung sind sechs Transitionen vorhanden. Da der Zustand $q_1$ den Zustand $q_2$ sowohl direkt als auch rückgerichtet simuliert, können die Transitionen $(q_0, a, q_2)$ und $(q_2, a, q_3)$ entfernt werden, da sie durch die Transitionen $(q_0, a, q_1)$ bzw. $(q_1, a, q_3)$ überdeckt werden, ohne die akzeptierte Sprache zu ändern. Dadurch wird der Zustand $q_2$ zu einem toten Zustand und kann in einem nächsten Schritt entfernt werden.

### 2.5.3 Reduktion durch Quotientenbildung

Nachdem die Anzahl der Zustände durch Entfernen toter Zustände und die Anzahl der Transitionen durch Transitionsstutzung reduziert wurden, steht noch eine weitere Möglichkeit zur Verfügung, um die Anzahl der Zustände zu verringern. Diese besteht darin, äquivalente Zustände in dem Automaten zusammenzufassen und somit einen kleineren Automaten zu erhalten.

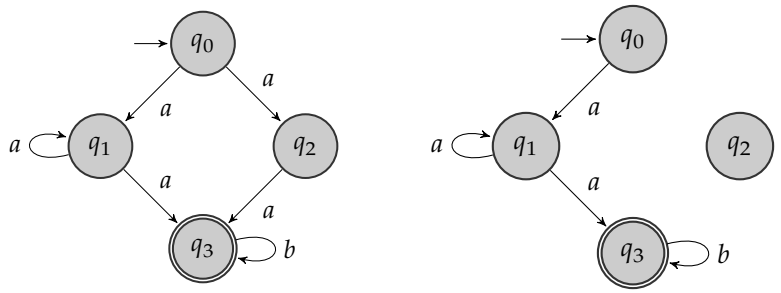



Abbildung 2.6: Beispiel für die Transitionsstutzung eines NBAs. Die Kantenfolge von $q_0$ über $q_2$ nach $q_3$ wurde entfernt, da sie durch die Kantenfolge von $q_0$ über $q_1$ nach $q_3$ vollständig überdeckt wird.

Im Folgenden betrachten wir einen NBA $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ und eine Quasiordnung $\sqsubseteq$ auf $Q$, diese induziert eine Äquivalenzrelation $\equiv = \sqsupseteq \cap \sqsubseteq$.

Sei $q \in Q$ ein Zustand aus $\mathcal{A}$, dann bezeichnet $[q]$ die Äquivalenzklasse von $q$ bzgl. $\equiv$. Für eine Teilmenge von Zuständen $P \subseteq Q$ bezeichnet $[P] = \{[p] \mid p \in P\}$ die Menge der Äquivalenzklassen.

**Definition 2.19** Quotientenautomat [CM13]
Der *Quotientenautomat* eines Automaten $\mathcal{A}$ bezüglich $\sqsubseteq$ ist definiert als $\mathcal{A}/\sqsubseteq := (\Sigma, Q', I', \Delta', F')$ mit

$$
\begin{aligned}
Q' &= [Q], \\
I' &= \{[q] \in Q' \mid \exists \hat{q} \in I : \hat{q} \in [q]\}, \\
F' &= \{[q] \in Q' \mid \exists \hat{q} \in F : \hat{q} \in [q]\} \text{ und} \\
\Delta' &= \{([q_i], \sigma, [q_j]) \in Q' \times \Sigma \times Q' \mid \exists \hat{q}_i \in [q_i], \hat{q}_j \in [q_j] : \\
&\qquad\qquad (\hat{q}_i, \sigma, \hat{q}_j) \in \Delta\}.
\end{aligned}
$$

Offensichtlich existiert für jeden Lauf $q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \cdots$ in $\mathcal{A}$ ein entsprechender Lauf $[q_0] \xrightarrow{\sigma_0} [q_1] \xrightarrow{\sigma_1} \cdots$ in $\mathcal{A}/\sqsubseteq$, dieser ist initial/fair,

wenn der Ausgangslauf initial/fair ist.

**Definition 2.20** Good for Quotienting (GFQ) [CM13]
Eine Quasiordnung $\sqsubseteq$ ist für die Quotientenbildung geeignet (engl. *Good for Quotienting (GFQ)*), gdw. $\mathcal{A}/\sqsubseteq \approx \mathcal{A}$ gilt.

Die direkte Simulationen $\sqsubseteq^{\mathrm{di}}$ und verzögerte Simulation $\sqsubseteq^{\mathrm{de}}$ sind PTIME-berechenbare GFQ Quasiordnungen [EWS01]. Die faire Simulation $\sqsubseteq^{\mathrm{f}}$ ist zwar PTIME-berechenbar nicht jedoch GFQ[HKR02]. Entsprechend ist die rückgerichtete Simulation $\sqsubseteq^{\mathrm{bw}}$ eine PTIME-berechenbare GFQ Quasiordnung [CM13]. Die direkte und die rückgerichtete Laufinklusion sind GFQ Quasiordnungen [Ete02; CM13]. Da bereits die faire Simulation nicht GFQ ist, ist auch die faire Laufinklusion nicht GFQ. Ebenso ist die verzögerte Laufinklusion nicht GFQ [Cle11]. Auch die direkte $k$-Lookahead-Simulation $\sqsubseteq^{k\text{-}\mathrm{di}}$ und die rückgerichtete $k$-Lookahead-Simulation $\sqsubseteq^{k\text{-}\mathrm{bw}}$ sind GFQ Quasiordnungen, da sie größenmäßig zwischen den entsprechenden Simulationen und Laufinklusionen liegen.

Abbildung 2.7 zeigt ein Beispiel für die Quotientenbildung auf einem NBA. Da die Zustände $q_1$ und $q_2$ sich gegenseitig direkt simulieren, sind sie äquivalent und können bei der Quotientenbildung bezüglich der direkten Simulation miteinander zu dem Zustand $[q_1]$ verschmolzen werden. Die übrigen Zustände $q_0$ und $q_3$ stehen in keiner Simulationsbeziehung zu anderen Zuständen und können daher auch nicht kombiniert werden. Sie werden im Quotientenautomaten zu $[q_0]$ und $[q_3]$.

### 2.5.4 Minimierungverfahren auf NBAs

Ausgehend von den in den vorherigen Unterabschnitten vorgestellten Minimierungstechniken werden in [CM13] verschiedene Vorgehen zur Minimierung von expliziten NBAs beschrieben, die die in den vorhergehenden Abschnitten erläuterten Techniken der Quotientenbildung und der Transitionsstutzung auf verschiedene Arten kombinieren. Zu-

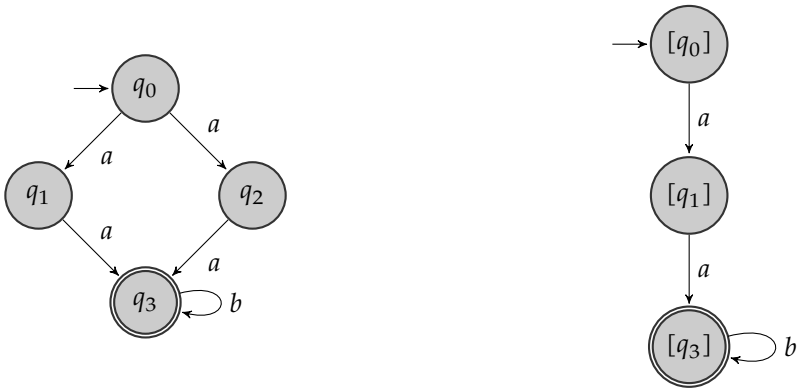



Abbildung 2.7: Beispiel für die Quotientenbildung auf einem NBA. Die Zustände $q_1$ und $q_2$ wurden verschmolzen zu einem Zustand $[q_1]$. Die Zustände $q_0$ und $q_3$ können nicht kombiniert werden.

dem werden die toten Zustände im Laufe der verschiedenen Verfahren entfernt.

Im Folgenden werden die $k$-Lookahead-Simulationen aus Unterabschnitt 2.4.1 als Unterapproximationen der in Abschnitt 2.3 beschriebenen Laufinklusionen verwendet, da diese nicht effizient berechnet werden können, gleichzeitig aber größere Simulationsrelationen benötigt werden, als mit den Simulationen aus Abschnitt 2.2 erreicht werden kann. Im Speziellen werden

- direkte $k$-Lookahead-Simulation $\preceq^{k\text{-di}}$ statt direkter Laufinklusion $\subseteq^{\text{di}}$,
- verzögerte $k$-Lookahead-Simulation $\preceq^{k\text{-de}}$ statt verzögerter $n$-Pebble-Simulation,
- faire $k$-Lookahead-Simulation $\preceq^{k\text{-f}}$ statt fairer Laufinklusion $\subseteq^{\text{f}}$ und
- rückgerichtete $k$-Lookahead-Simulation $\preceq^{k\text{-bw}}$ statt rückgerichteter Laufinklusion $\subseteq^{\text{bw}}$ verwendet.

Auf Basis dieser $k$-Lookahead-Simulationen können verschiedene Pruning-Relationen bestimmt werden.

- $P(id, \prec^{k\text{-di}})$ besagt, dass eine Transition $(q_i, \sigma, q_j)$ durch eine andere Transition $(q_i', \sigma, q_j')$ überdeckt wird, wenn die beiden Ausgangszustände übereinstimmen $q_i = q_i'$ und der Folgezustand $q_j$ durch den Zustand $q_j'$ mit Lookahead $k$ direkt simuliert wird $q_j \prec^{k\text{-di}} q_j'$.
- $P(\prec^{k\text{-bw}}, id)$ besagt, dass eine Transition $(q_i, \sigma, q_j)$ durch eine andere Transition $(q_i', \sigma, q_j')$ überdeckt wird, wenn die beiden Folgezustände übereinstimmen $q_j = q_j'$ und der Anfangszustand $q_i$ durch den Zustand $q_i'$ mit Lookahead $k$ rückgerichtet simuliert wird $q_j \prec^{k\text{-bw}} q_j'$.
- $P(\sqsubset^{\text{bw}}, \preceq^{k\text{-di}})$ besagt, dass eine Transition $(q_i, \sigma, q_j)$ durch eine andere Transition $(q_i', \sigma, q_j')$ überdeckt wird, wenn der Anfangszustand $q_i$ durch den Zustand $q_i'$ rückgerichtet simuliert wird $q_j \sqsubset^{\text{bw}} q_j'$ und der Folgezustand $q_j$ durch den Zustand $q_j'$ mit Lookahead $k$ direkt simuliert wird $q_j \prec^{k\text{-di}} q_j'$.
- $P(\preceq^{k\text{-bw}}, \sqsubset^{\text{di}})$ besagt, dass eine Transition $(q_i, \sigma, q_j)$ durch eine andere Transition $(q_i', \sigma, q_j')$ überdeckt wird, wenn der Anfangszustand $q_i$ durch den Zustand $q_i'$ mit Lookahead $k$ rückgerichtet simuliert wird $q_j \prec^{k\text{-bw}} q_j'$ und der Folgezustand $q_j$ durch den Zustand $q_j'$ direkt simuliert wird $q_j \sqsubset^{\text{di}} q_j'$.
- $R_t(\prec^{k\text{-f}})$ besagt, dass eine Transition $(q_i, \sigma, q_j)$ durch eine andere Transition $(q_i', \sigma, q_j')$ überdeckt wird, wenn $(q_i', \sigma, q_j')$ transient ist, also zwischen den starken Zusammenhangskomponenten des Automaten verläuft. Zusätzlich muss der Folgezustand $q_j$ durch den Zustand $q_j'$ fair simuliert werden $q_j \prec^{k\text{-di}} q_j'$.

Durch Verwendung dieser Relationen zur Stutzung der Transitionen werden alle überflüssigen Transitionen aus dem NBA entfernt, da diese nicht entscheidend für die Akzeptanz eines Wortes sind.

Für die Quotientenbildung werden verzögerte $k$-Lookahead-Simulation $\preceq^{k\text{-de}}$ und rückgerichtete $k$-Lookahead-Simulation $\preceq^{k\text{-bw}}$ verwendet. Zwei Zustände $q$ und $q'$ werden verschmolzen, wenn sie entweder in Vorwärtsrichtung oder in Rückwärtsrichtung nicht unterscheidbar sind $q \preceq^{k\text{-}x} q' \wedge q \succeq^{k\text{-}x} q'$.

Die so näher bestimmten Minimierungstechniken werden in [CM13] auf zwei Arten verknüpft und bilden somit zwei verschiedene Verfahren zur Minimierung expliziter NBAs. Die Korrektheit der Minimierungstechniken ergibt sich daraus, dass nur passende Quasiordnungen als Grundlage der Minimierung dienen. Für die Stutzung von Transitionen werden ausschließlich GFP Quasiordnungen $P$ verwendet, sodass für den gestutzten Automaten $\mathrm{Prune}(\mathcal{A}, P) \approx \mathcal{A}$ gilt. Da auch die Quotientenbildung ausschließlich mit GFQ Quasiordnungen $\sqsubseteq$ durchgeführt wird, gilt für den Quotientenautomaten immer $\mathcal{A}/\sqsubseteq \approx \mathcal{A}$. Das Entfernen toter Zustände kann aufgrund ihrer Definition keinen Einfluss auf die akzeptierte Sprache des Automaten haben. Daher gilt für alle Minimierungsverfahren, die ausschließlich die vorgestellten Techniken verwenden, dass sie in dem Sinne korrekt sind, dass sie die akzeptierte Sprache des Ausgangsautomaten nicht verändern.

### Heavy-$k$

Die Variante *Heavy-k* aus Algorithmus 2.1 wendet alle bisher vorgestellten Techniken wiederholt an, bis ein Fixpunkt erreicht wird. Der erreichte Fixpunkt stellt dabei nur ein lokales Minimum im Raum der möglichen Automaten dar, die dieselbe Sprache wie der Ausgangsautomat akzeptieren [CM13].

In jeder Iteration des Verfahrens werden

1. zuerst die toten Zustände enfernt,
2. werden die Transitionen des Automaten bzg. der GFP Relationen mit Lookahead $k$ gestutzt und abschließend
3. wird der Quotient des Automaten bzgl. $\preceq^{k\text{-de}}$ und $\preceq^{k\text{-bw}}$ gebildet.

**Algorithm 2.1** Minimierungsverfahren Heavy-$k$

**input:** $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$

**procedure** HEAVY-$k(\mathcal{A})$
  **repeat**
    $last \leftarrow \mathcal{A}$
    **for all** $q \in Q$ **do**
      **if** $q$ ist tot **then**
        $Q \leftarrow Q \setminus q$
      **end if**
    **end for**
    $\mathcal{A} \leftarrow \text{Prune}(\mathcal{A}, P(id, \prec^{k\text{-di}}))$
    $\mathcal{A} \leftarrow \text{Prune}(\mathcal{A}, P(\prec^{k\text{-bw}}, id))$
    $\mathcal{A} \leftarrow \text{Prune}(\mathcal{A}, P(\sqsubset^{\text{bw}}, \preceq^{k\text{-di}}))$
    $\mathcal{A} \leftarrow \text{Prune}(\mathcal{A}, P(\preceq^{k\text{-bw}}, \sqsubset^{\text{di}}))$
    $\mathcal{A} \leftarrow \text{Prune}(\mathcal{A}, R_t(\prec^{k\text{-f}}))$
    $\mathcal{A} \leftarrow \mathcal{A}/\preceq^{k\text{-de}}$
    $\mathcal{A} \leftarrow \mathcal{A}/\preceq^{k\text{-bw}}$
  **until** $\mathcal{A} = last$
  **return** $\mathcal{A}$
**end procedure**

Das Ergebnis der Minimierung kann mit den vorgestellten Techniken nicht weiter minimiert werden und stellt somit ein lokales Minimum für die Automatengröße dar.

### Light-$k$

Die Variante *Light-k* aus Algorithmus 2.2 wird in [CM13] nur beschrieben, um eine Vergleichbarkeit mit anderen in der Literatur beschriebenen Minimierungen zu erhalten. Das Verfahren durchläuft nur eine Iteration, in der

1. zuerst die toten Zustände entfernt werden und
2. danach der Quotient des Automaten bzgl. $\preceq^{k\text{-de}}$ bestimmt wird .

**Algorithm 2.2** Minimierungsverfahren Light-$k$

**input:** $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$

**procedure** LIGHT-$k$($\mathcal{A}$)
  **for all** $q \in Q$ **do**
    **if** $q$ ist tot **then**
      $Q \leftarrow Q \setminus q$
    **end if**
  **end for**
  $\mathcal{A} \leftarrow \mathcal{A}/\preceq^{k\text{-de}}$
  **return** $\mathcal{A}$
**end procedure**

Das Verfahren zeigt die Auswirkungen des $k$-Lookaheads auf die Minimierung, wenn nur ein einziger Quotientenbildungsdurchgang ausgeführt wird.

# 3 Symbolische Automaten und deren Minimierung

Im vorherigen Kapitel wurde das von Clemente und Mayr in [CM13] beschriebene Minimierungsframework eingeführt, dieses wollen wir in diesem Kapitel auf zwei Abwandlungen der bekannten nichtdeterministischen Büchi-Automaten übertragen.

In Abschnitt 3.1 werden wir zuerst die *nichtdeterministischen Büchi-Automaten mit symbolischer Transitionsbedingung* als Abwandlung der expliziten NBAs aus Abschnitt 2.1 definieren. Der Abschnitt 3.2 beschreibt, wie die verschiedenen Simulations- und $k$-Lookahead-Simulationsrelationen aus Abschnitt 2.2 und Unterabschnitt 2.4.1 auf die neu definierten Automaten übertragen werden können und ausgehend davon auch die Berechnung dieser Relationen. Aufbauend auf den so berechneten Relationen beschreibt Abschnitt 3.3 dann auch die Minimierung der NBAs mit symbolischen Transitionsbedingungen.

In dem darauf folgenden Abschnitt 3.4 werden *vollständig symbolisch repräsentierte nichtdeterministisch Büchi-Automaten* definiert. Diese weiten die symbolische Beschreibung von Automaten auch auf deren Zustandsmenge aus. Die Berechnung der Simulations- und $k$-Lookahead-Simulationsrelationen auf den so erweiterten Automaten wird in Abschnitt 3.5 beschrieben. In Abschnitt 3.6 wird dann auch das Minimierungsframework übertragen.

## 3.1 Nichtdeterministische Büchi-Automaten mit symbolischer Transitionsbedingung (SNBAs)

Wie bereits in der Einleitung erwähnt, weisen die Eingabealphabete der in der Verifikation verwendeten Automaten häufig eine bestimmte Struktur auf. Die symbolische Beschreibung der Transitionsbedingungen dieser Automaten nutzt die Struktur ihrer Eingabealphabete, um die Automaten deutlich präziser beschreiben zu können. Dies führt auch zu einer Steigerung der Effizienz ihrer Verarbeitung.

Daher werden wir in diesem Abschnitt die expliziten NBAs aus Abschnitt 2.1 so erweitern, dass die Bedingung für einen Zustandsübergang zwischen ihren Zuständen nicht mehr explizit durch Aufzählung der möglichen Eingabesymbole beschrieben wird. Stattdessen wird die Bedingung auf Basis einer symbolischen Repräsentation des Eingabealphabets durch eine Formel beschrieben. Automaten über symbolischen Alphabeten wurden bereits in [Vea13] definiert, dort jedoch nur für Automaten auf endlichen Worten.

Bevor wir jedoch die symbolisierten NBAs definieren, beschreiben wir in Unterabschnitt 3.1.1, wie eine beliebige endliche Menge über einer passenden Repräsentationsdomäne symbolisch beschrieben werden kann. Darauf aufbauend definieren wir dann in Unterabschnitt 3.1.2 die nichtdeterministischen Büchi-Automaten mit symbolischen Transitionsbedingungen.

### 3.1.1 Symbolische Mengenbeschreibung

In diesem Unterabschnitt beschreiben wir zuerst, wie eine beliebige endliche Menge $X$ über einer passenden Repräsentationsdomäne symbolisch repräsentiert werden kann. Außerdem veranschaulichen wir das Konzept in einem Beispiel, das uns für das weitere Vorgehen als Referenz dienen soll.

Im Folgenden bezeichnen wir mit $\mathbb{B}$ die zweiwertige booleschen Algebra über der Menge $\{\bot, \top\}$. Für eine Menge $P$ von aussagenlogischen Variablen bezeichnen wir mit $B(P)$ die Menge aller aussagenlogischen Formeln über $P$, die nach den üblichen Regeln gebildet werden. Mit $\mathfrak{B}(P)$ bezeichnen wir die Menge aller Belegungen $\beta : P \to \mathbb{B}$ der Variablen aus $P$. Jede $k$-stellige boolesche Funktion kann durch eine aussagenlogische Formel über $k$ Variablen beschrieben werden.

Jede endliche Teilmenge $T$ einer Menge $X$ kann symbolisch repräsentiert werden, dafür wird die *charakteristische Funktion* der Menge $T$ symbolisch beschrieben. Die charakteristische Funktion der Menge $T$ ist $\chi_T : X \to \mathbb{B}$ mit

$$\chi_T(x) = \begin{cases} \top & \text{falls } x \in T, \\ \bot & \text{sonst.} \end{cases}$$

Um alle Teilmengen von $X$ zu kodieren, benötigt man $n = \lceil \log_2 |X| \rceil$ boolesche Variablen $V_X = \{v_0, \ldots, v_{n-1}\}$. Mit $\mathbf{V}_X = \mathbb{B}^{|V_X|}$ bezeichnen wir die Menge der durch boolesche Funktionen über $V_X$ repräsentierbaren Elemente. Wir bezeichnen die Menge $V_X$ und die Reihenfolge ihrer Elemente auch als Repräsentationsdomäne von $X$.

Die injektive Funktion $c_X : X \to \mathbf{V}_X$ kodiert jedes Element von $X$ durch genau einen Bitvektor $(v_0, v_1 \ldots, v_{n-1}) \in \mathbf{V}_X$, den wir auch mit $\mathbf{x}$ bezeichnen. Mit $\hat{c}_X : X \to B(V_X)$ bezeichnen wir die Kodierungsfunktion, die jedes Element von $X$ durch genau eine aussagenlogische Formel $\varphi \in B(V_X)$ kodiert, für die gilt $c_X(x) \models \varphi$. Zur einfacheren Darstellung verwenden wir das Symbol $c_X$ für beide Kodierungsfunktionen, wenn sich die verwendete Funktion aus dem Kontext ergibt. Zudem erweitern wir die Kodierungsfunktion wie üblich auf Mengen:

$$c_X(M) = \{c_X(m) \mid m \in M\}$$

Um nun eine Teilmenge $T$ symbolisch zu repräsentieren, interpretieren wir die charakteristische Funktion $\chi_T$ als boolesche Funktion $f_T$ von der

Form $f_T : \mathbf{V}_X \to \mathbb{B}$. Die symbolische Funktion $f_T$ ergibt sich wie folgt aus der Mengenzugehörigkeit von $T$

$$f_T(\mathbf{x}) = f_T(v_0, \ldots, v_{n-1}) = \bigvee_{x \in X} (\chi_T(x) \wedge c_X(x)) = \bigvee_{t \in T} c_X(t).$$

**Beispiel 3.1.** Um alle Teilmengen von $X = \{a, b, c, d\}$ zu kodieren benötigen wir eine Repräsentationsdomäne $V_X = \{c_0, c_1\}$ mit zwei Variablen $c_0$ und $c_1$. Die Kodierungsfunktion $c_X$ ist dann wie folgt definiert

$$c_X(a) = (\bot, \bot) \qquad c_X(b) = (\bot, \top) \qquad c_X(c) = (\top, \bot) \qquad c_X(d) = (\top, \top)$$

$$\hat{c}_X(a) = \neg c_0 \wedge \neg c_1 \quad \hat{c}_X(b) = \neg c_0 \wedge c_1 \quad \hat{c}_X(c) = c_0 \wedge \neg c_1 \quad \hat{c}_X(d) = c_0 \wedge c_1$$

Der Teilmenge $T = \{a, b\} \subseteq X$ wird die charakteristische Funktion $\chi_T$ zugeordnet mit

$$\chi_T(a) = \top \quad \chi_T(b) = \top \quad \chi_T(c) = \bot \quad \chi_T(d) = \bot.$$

Die charakteristische Funktion von $T$ können wir durch die folgende boolesche Funktion symbolisch repräsentieren

$$\begin{aligned} f_T(c_0, c_1) &= c_X(a) \vee c_X(b) \\ &= (\neg c_0 \wedge \neg c_1) \vee (\neg c_0 \wedge c_1) \\ &= \neg c_0. \end{aligned}$$

Zur Probe betrachten wir die Auswertung von $f_T$ für verschiedene Belegungen von $V_X$

$$f_T(c_X(a)) = \top \quad f_T(c_X(b)) = \top \quad f_T(c_X(c)) = \bot \quad f_T(c_X(d)) = \bot.$$

### 3.1.2 Einführung von NBAs mit symbolischer Transitionsbedingung

Im Folgenden wollen wir die im vorherigen Unterabschnitt beschriebene Eigenschaft endlicher Mengen nutzen, um nichtdeterministische Büchi-Automaten mit symbolischen Transitionsbedingungen zu definieren. Diese sind dem bereits bekannten NBA aus Definition 2.1 sehr ähnlich, beschreiben die Transitionsbedingungen jedoch nicht explizit durch Aufzählen der möglichen Eingabesymbole, sondern durch eine aussagenlogische Formel, die die Teilmenge der möglichen Eingabesymbole symbolisch beschreibt. Jede Kante im symbolischen Automaten beschreibt daher eine Menge von Transitionen im expliziten Automaten.

**Definition 3.1** Symbolischer Nichtdeterministischer Büchi-Automat (SNBA)
Ein *symbolischer nichtdeterministischer Büchi-Automat (SNBA)* ist ein 6-Tupel $\mathcal{A} = (V_\Sigma, f_\Sigma, Q, I, \delta, F)$, wobei

- $V_\Sigma$ die Menge der *Repräsentationsvariablen* von $\Sigma$,
- $f_\Sigma : \mathfrak{B}(V_\Sigma) \to \mathbb{B}$ eine Funktion, die die *gültigen Belegungen* von $V_\Sigma$ beschreibt,
- $Q$ eine endliche Menge von *Zuständen*,
- $I \subseteq Q$ eine Menge von *initialen* Zuständen,
- $\delta : Q \times Q \to B(V_\Sigma)$ die Transitionsfunktion und
- $F \subseteq Q$ eine Menge von *akzeptierenden* Zuständen ist.

Ein SNBA $\mathcal{A}$ kann auch als Graph repräsentiert werden. Abbildung 3.1 zeigt ein Beispiel eines einfachen SNBAs $\mathcal{A}$ sowohl durch mathematische Strukturen beschrieben, als auch seine grafische Repräsentation. Unter Verwendung der Kodierungsfunktion aus Beispiel 3.1 akzeptiert der Automat nach Umkodierung auf explizite Worte die dieselbe Sprache wie der Automat aus Abbildung 2.1.

Befindet sich der Automat zu Beginn in Zustand $q_0$, dann verbleibt er in diesem Zustand, solange er Eingaben $\beta \in \mathfrak{B}(V_\Sigma)$ liest, für die $\beta(c_1) =$

$$
\begin{aligned}
V_\Sigma &= \{c_0, c_1\} \\
f_\Sigma(c_0, c_1) &= \text{true} \\
Q &= \{q_0, q_1\} \\
I &= \{q_0\} \\
\delta = \; & \{(q_0, q_0) \mapsto c_0, \\
& \; (q_0, q_1) \mapsto c_1, \\
& \; (q_1, q_1) \mapsto \text{true}\} \\
F &= \{q_1\}
\end{aligned}
$$

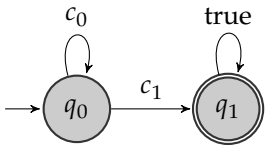



Abbildung 3.1: Beispiel für einen einfachen SNBA $\mathcal{A}$ in mathematischer und graphischer Darstellung. Unter Verwendung der Kodierungsfunktion aus Beispiel 3.1 akzeptiert der Automat nach Umkodierung auf explizite Worte die gleiche Sprache $\{uvw \in \Sigma^\omega \mid u \in \{b,d\}^*, v \in \{c,d\} \text{ und } w \in \Sigma^\omega\}$ wie der Automat aus Abbildung 2.1.

$\top$ gilt. Liest er hingegen eine Eingabe mit $\beta(c_0) = \top$, wechselt er von $q_0$ nach $q_1$. Gilt für eine Eingabe sowohl $\beta(c_0) = \top$ als auch $\beta(c_1) = \top$, so trifft der Automat nichtdeterministisch eine Wahl zwischen $q_0$ und $q_1$. Wenn stattdessen $\beta(c_0) = \bot$ und $\beta(c_1) = \bot$ gilt, verwirft der Automat die Eingabe und akzeptiert das Wort nicht. Befindet sich der Automat stattdessen in $q_1$, so verbleibt der Automat für jede mögliche Eingabe in $q_1$ und akzeptiert das Wort, da für alle $\beta \in \mathfrak{B}(V_\Sigma)$ gilt, dass $\beta \models \text{true}$.

Eine Transition zwischen zwei Zuständen $q_i$ und $q_j$ des Automaten existiert, gdw. eine Belegung $\beta \in \mathfrak{B}(V_\Sigma)$ existiert, für die $\beta(\delta(q_i, q_j)) \wedge f_\Sigma(\beta)$ gilt. Wir schreiben dann auch $q_i \xrightarrow{\varphi} q_j \in \delta$. Für $\beta \in \mathfrak{B}(V_\Sigma)$ ist eine $\beta$-Transition $q_i \xrightarrow{\beta} q_j$ eine Transition $q_i \xrightarrow{\varphi} q_j$, für die $\beta \models \varphi$ gilt.

Aus Gründen der einfacheren Darstellung nehmen wir an, dass die betrachteten Automaten *vorwärts und rückwärts vollständig* sind: Es existieren für jeden Zustand $q_i \in Q$ und jede Belegung $\beta \in \mathfrak{B}(V_\Sigma)$ Zustände

$q_h, q_j \in Q$, sodass $q_i \xrightarrow{\varphi} q_j \in \delta \wedge \beta \models \varphi$ und $q_h \xrightarrow{\varphi'} q_i \in \delta \wedge \beta \models \varphi'$ gilt. Auch ein symbolischer NBA kann durch Hinzufügen zweier Zustände und zweier Transitionen pro Zustand vervollständigt werden.

Ein *unendlicher Lauf* von $\mathcal{A}$ über einem unendlichen Wort $w = \beta_0\beta_1 \ldots \in \mathfrak{B}(V_\Sigma)^\omega$ beginnt in einem beliebigen Zustand $q_0 \in Q$ und ist eine unendliche Folge von Transitionen $\pi = q_0 \xrightarrow{\varphi_0} q_1 \xrightarrow{\varphi_1} \cdots$ mit $\forall i \geq 0 : \beta_i \models \varphi_i$.

Die Eigenschaften initial, fair und akzeptierend sind entsprechend dem expliziten NBA aus Definition 2.1 definiert. Der SNBA $\mathcal{A}$ akzeptiert die Sprache

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^\omega | \mathcal{A} \text{ hat mindestens einen akzeptierenden Lauf auf } w\}.$$

**Lemma 3.1**
*Für jeden expliziten NBA $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ existiert ein symbolischer SNBA $\dot{\mathcal{A}} = (V_\Sigma, f_\Sigma, \dot{Q}, \dot{I}, \delta, \dot{F})$ mit $c_\Sigma(\mathcal{L}(\mathcal{A})) = \mathcal{L}(\dot{\mathcal{A}})$.*

Für die folgenden Beweise erweitern wir die Funktion $c_\Sigma$ zu $c_\Sigma^\omega : \Sigma^\omega \to \mathfrak{B}(V_\Sigma)^\omega$ mit

$$c_\Sigma^\omega(xw) = c_\Sigma(x)c_\Sigma^\omega(w) \text{ mit } x \in \Sigma \text{ und } w \in \Sigma^\omega.$$

Wir verwenden jedoch das Symbol $c_\Sigma$ sowohl für die Funktion $c_\Sigma$ als auch für die Funktion $c_\Sigma^\omega$.

*Beweis.* Sei $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ ein beliebiger NBA. Wir konstruieren einen SNBA $\dot{\mathcal{A}} = (V_\Sigma, f_\Sigma, Q, I, \delta, F)$, indem wir die Mengen $Q$, $I$ und $F$ übernehmen und ansonsten wie folgt vorgehen:

1. Wir erzeugen eine Repräsentationsdomäne $V_\Sigma = \{v_0, v_1, \ldots, v_{n-1}\}$ mit $n = \lceil \log_2 |\Sigma| \rceil$, über der die $\sigma \in \Sigma$ repräsentiert werden. Die Funktion $c_\Sigma : \Sigma \to \mathfrak{B}(V_\Sigma)$ kodiert jedes $\sigma \in \Sigma$ eindeutig als Bitvektor über $V_\Sigma$.

2. Die gültigen Eingaben werden beschrieben durch

$$f_\Sigma = \bigvee_{\sigma \in \Sigma} c_\Sigma(\sigma).$$

3. Wir konstruieren die Transitionsfunktion $\delta : Q \times Q \to B(V_\Sigma)$ wie folgt:

$$\delta(q_i, q_j) = \bigvee_{(q_i, \sigma, q_j) \in \Delta} c_\Sigma(\sigma)$$

Wir zeigen zuerst $c_\Sigma(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\dot{\mathcal{A}})$:

Sei $w \in \mathcal{L}(\mathcal{A}) \subseteq \Sigma^\omega$ ein beliebiges Wort, das der Automat $\mathcal{A}$ akzeptiert. Daraus folgt, dass in $\mathcal{A}$ ein akzeptierender unendlicher Lauf $\rho = q_0 \xrightarrow{w(0)} q_1 \xrightarrow{w(1)} \cdots$ über $w$ mit $q_0 \in I$ und $\inf(\rho) \cap F \neq \emptyset$ existiert.

Dann existiert aber auch in $\dot{\mathcal{A}}$ ein Lauf $\dot{\rho} = q_0 \xrightarrow{\varphi_0} q_1 \xrightarrow{\varphi_1} \cdots$, für den $\forall i \geq 0 : c_\Sigma(w(i)) \models \varphi_i$ sowie $q_0 \in \dot{I}$ und $\inf(\rho) \cap \dot{F} \neq \emptyset$ gilt. Damit akzeptiert auch $\dot{\mathcal{A}}$ das Wort $c_\Sigma(w)$ und es gilt $w \in \mathcal{L}(\mathcal{A}) \implies c_\Sigma(w) \in \mathcal{L}(\dot{\mathcal{A}})$.

Jetzt zeigen wir noch $c_\Sigma(\mathcal{L}(\mathcal{A})) \supseteq \mathcal{L}(\dot{\mathcal{A}})$:

Sei $c_\Sigma(w) \in \mathcal{L}(\dot{\mathcal{A}}) \subseteq \mathfrak{B}(V_\Sigma)^\omega$ ein beliebiges Wort, das der Automat $\dot{\mathcal{A}}$ akzeptiert. Daraus folgt, dass in $\dot{\mathcal{A}}$ ein akzeptierender unendlicher Lauf $\dot{\rho} = q_0 \xrightarrow{\varphi_0} q_1 \xrightarrow{\varphi_1} \cdots$ existiert, für den $\forall i \geq 0 : c_\Sigma(w)(i) \models \varphi_i$ sowie $q_0 \in \dot{I}$ und $\inf(\dot{\rho}) \cap \dot{F} \neq \emptyset$ gilt.

Dann existiert aber auch in $\mathcal{A}$ ein Lauf $\rho = q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \cdots$, für den $\forall i \geq 0 : w(i) = \sigma_i$ sowie $q_0 \in I$ und $\inf(\rho) \cap F \neq \emptyset$ gilt. Damit akzeptiert auch $\mathcal{A}$ das Wort $w$ und es gilt $c_\Sigma(w) \in \mathcal{L}(\dot{\mathcal{A}}) \implies w \in \mathcal{L}(\mathcal{A})$.

Daraus folgt $c_\Sigma(\mathcal{L}(\mathcal{A})) = \mathcal{L}(\dot{\mathcal{A}})$. □

**Lemma 3.2**
*Für jeden symbolischen SNBA $\dot{\mathcal{A}} = (V_\Sigma, f_\Sigma, \dot{Q}, \dot{I}, \delta, \dot{F})$ existiert ein expliziter NBA $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ mit $c_\Sigma^{-1}(\mathcal{L}(\dot{\mathcal{A}})) = \mathcal{L}(\mathcal{A})$.*

*Beweis.* Sei $\dot{\mathcal{A}} = (V_\Sigma, f_\Sigma, Q, I, \delta, F)$ ein beliebiger SNBA. Wir konstruieren einen NBA $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$, indem wir die Mengen $Q$, $I$ und $F$ übernehmen und ansonsten wie folgt vorgehen:

1. Wir erzeugen ein Eingabealphabet aus der Potenzmenge $S = 2^{V_\Sigma}$. Die Funktion $c_\Sigma^{-1} : \mathfrak{B}(V_\Sigma) \to S$ kodiert jede Belegung von $V_\Sigma$ eindeutig als ein Element von $S$. Das Eingabealphabet $\Sigma$ wird dann bestimmt durch

$$\Sigma = \{s \in S \mid f_\Sigma(c_\Sigma(s))\}$$

2. Wir konstruieren die Transitionsrelation $\Delta \subseteq Q \times \Sigma \times Q$ wie folgt:

$$\Delta = \{(q_i, \sigma_i, q_j) \mid c_\Sigma(\sigma_i) \models \delta(q_i, q_j)\}$$

Wir zeigen zuerst $c_\Sigma^{-1}(\mathcal{L}(\dot{\mathcal{A}})) \subseteq \mathcal{L}(\mathcal{A})$:

Sei $w \in \mathcal{L}(\dot{\mathcal{A}}) \subseteq \mathfrak{B}(V_\Sigma)^\omega$ ein beliebiges Wort, das der Automat $\dot{\mathcal{A}}$ akzeptiert. Daraus folgt, dass in $\dot{\mathcal{A}}$ ein akzeptierender unendlicher Lauf $\dot{\rho} = q_0 \xrightarrow{\varphi_0} q_1 \xrightarrow{\varphi_1} \cdots$ existiert, für den $\forall i \geq 0 : w(i) \models \varphi_i$ sowie $q_0 \in \dot{I}$ und $\inf(\dot{\rho}) \cap \dot{F} \neq \emptyset$ gilt.

Dann existiert aber auch in $\mathcal{A}$ ein Lauf $\rho = q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \cdots$, für den $\forall i \geq 0 : c_\Sigma^{-1}(w(i)) = \sigma_i$ sowie $q_0 \in I$ und $\inf(\rho) \cap F \neq \emptyset$ gilt. Damit akzeptiert auch $\mathcal{A}$ das Wort $c_\Sigma^{-1}(w)$ und es gilt $w \in \mathcal{L}(\dot{\mathcal{A}}) \implies c_\Sigma^{-1}(w) \in \mathcal{L}(\mathcal{A})$.

Jetzt zeigen wir noch $c_\Sigma^{-1}(\mathcal{L}(\dot{\mathcal{A}})) \supseteq \mathcal{L}(\mathcal{A})$:

Sei $c_\Sigma^{-1}(w) \in \mathcal{L}(\mathcal{A}) \subseteq \Sigma^\omega$ ein beliebiges Wort, das der Automat $\mathcal{A}$ akzeptiert. Daraus folgt, dass in $\mathcal{A}$ ein akzeptierender unendlicher Lauf $\rho = q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \cdots$ über $c_\Sigma^{-1}(w)$ existiert mit $\forall i \geq 0 : c_\Sigma^{-1}(w(i)) = \sigma_i$ sowie $q_0 \in I$ und $\inf(\rho) \cap F \neq \emptyset$.

Dann existiert aber auch in $\dot{\mathcal{A}}$ ein Lauf $\dot{\rho} = q_0 \xrightarrow{\varphi_0} q_1 \xrightarrow{\varphi_1} \cdots$, für den $\forall i \geq 0 : w(i) \models \varphi_i$ sowie $q_0 \in \dot{I}$ und $\inf(\rho) \cap \dot{F} \neq \emptyset$ gilt. Damit akzeptiert auch $\dot{\mathcal{A}}$ das Wort $w$ und es gilt $c_\Sigma^{-1}(w) \in \mathcal{L}(\mathcal{A}) \implies w \in \mathcal{L}(\dot{\mathcal{A}})$.

Daraus folgt $c_\Sigma^{-1}(\mathcal{L}(\dot{\mathcal{A}})) = \mathcal{L}(\mathcal{A})$. □

**Theorem 3.3**
*NBAs und SNBAs sind gleichmächtig.*

Der Beweis folgt direkt aus Lemma 3.1 und Lemma 3.2.

## 3.2 $k$-Lookahead-Simulationen auf symbolischen nichtdeterministischen Büchi-Automaten

In diesem Abschnitt beschreiben wir, wie die Simulations- und $k$-Lookahead-Simulationsrelationen aus Abschnitt 2.2 und Unterabschnitt 2.4.1 auf symbolische NBAs übertragen werden.

Wir werden jedoch die Simulationsrelationen aus Abschnitt 2.2 auf symbolischen NBAs nicht gesondert betrachten, da sie auf symbolische $k$-Lookahead-Simulationen mit Parameter $k = 1$ abbildbar sind.

Weiterhin betrachten wir die symbolischen Laufinklusionsrelationen nicht, da bereits die Laufinklusionsrelationen aus Abschnitt 2.3 auf expliziten NBAs nicht effizient berechenbar sind.

Stattdessen übertragen wir die $k$-Lookahead-Simulationen, die in Unterabschnitt 2.4.1 auf NBAs definiert werden, auf SNBAs. Dabei machen wir uns zunutze, dass die Definitionen im Wesentlichen nur von den während der Simulationsspiele besuchten Zuständen des Automaten abhängen. Danach werden wir auch die Vorgänger-Operatoren aus Unterabschnitt 2.4.2 entsprechend übertragen.

### 3.2.1 Übertragung der Definitionen

Da sich viele Definitionen auf expliziten NBAs ohne Anpassung auf symbolische NBAs übertragen lassen, werden wir im Folgenden nur

die Definitionen erneut aufführen, die auf SNBAs angepasst werden müssen.

Wie zuvor kann die symbolische $k$-Lookahead-Simulationsbeziehung zweier Zustände $q_0$ und $\hat{q}_0$ formal als ein Spiel zwischen zwei Spielern beschrieben werden. Das symbolische $k$-Lookahead-Grundspiel $G_{\mathcal{A}}(q_0, \hat{q}_0)$ ändert das explizite $k$-Lookahead-Grundspiel aus Definition 2.10 so ab, dass Spoiler und Duplikator symbolische Transitionsfolgen mit passenden Belegungen $\beta_j$ wählen anstatt expliziter Transitionsfolgen.

**Definition 3.2** Symbolisches $k$-Lookahead-Grundspiel
Sei $\mathcal{A} = (V_\Sigma, f_\Sigma, Q, I, \delta, F)$ ein beliebiger SNBA. Dann wird das *symbolische k-Lookahead-Grundspiel* $G_{\mathcal{A}}(q_0, \hat{q}_0)$ auf $\mathcal{A}$ gespielt, indem Spoiler und Duplikator symbolische Transitionsfolgen mit passenden Belegungen $\beta_j$ wählen.

Zu Beginn von Runde $i$ befindet sich das Spiel in der Konfiguration $(q_i, \hat{q}_i)$ und Spoiler beginnt:

1. Spoiler wählt eine Folge von $k$ zusammenhängenden symbolischen Transitionen $q_i \xrightarrow{\varphi_i} q_{i+1} \xrightarrow{\varphi_{i+1}} \cdots \xrightarrow{\varphi_{i+k-1}} q_{i+k}$ sowie $k$ passende Belegungen $\beta_j$ mit $\forall i \leq j \leq i+k : \beta_j \models \varphi_j$ aus.
2. Duplikator wählt eine Zahl $1 \leq m \leq k$ aus und antwortet mit einer passenden Folge von $m$ Transitionen $\hat{q}_i \xrightarrow{\hat{\varphi}_i} \hat{q}_{i+1} \xrightarrow{\hat{\varphi}_{i+1}} \cdots \xrightarrow{\hat{\varphi}_{i+m-1}} \hat{q}_{i+m}$ mit $\forall i \leq j \leq i+m : \beta_j \models \hat{\varphi}_j$.
3. Die verbleibenden, von Spoiler gewählten, Transitionen werden ignoriert und die neue Konfiguration ist $(q_{i+m}, \hat{q}_{i+m})$.

Entweder endet das Grundspiel vorzeitig und Spoiler gewinnt das Spiel, oder es läuft unendlich lange. Dabei werden zwei unendliche symbolische Läufe $\rho = q_0 \xrightarrow{\varphi_0} q_1 \xrightarrow{\varphi_1} \cdots$ und $\hat{\rho} = \hat{q}_0 \xrightarrow{\hat{\varphi}_0} \hat{q}_1 \xrightarrow{\hat{\varphi}_1} \cdots$ produziert, die sich aus den gewählten Transitionen ergeben. Wie zuvor bezeichnen wir $(\rho, \hat{\rho})$ auch als das Resultat des Grundspiels.

Da der Automat als vollständig angenommen wird, müssen die von Spoiler gewählten Belegungen für $V_\Sigma$ nicht zwingend gültig sein. Ungültige Belegungen können nur in die Vorwärtssenke $\bot_f$ führen, die dafür gewählte Transition steht allerdings auch Duplikator immer zur Verfügung, daher hat dies keine Auswirkungen auf die berechnete Simulationsrelation.

Das symbolische rückgerichtete $k$-Lookahead-Grundspiel $G_\mathcal{A}(q_0, \hat{q}_0)$ ändert das explizite rückgerichtete $k$-Lookahead-Grundspiel aus Definition 2.11 so ab, dass Spoiler und Duplikator symbolische Transitionsfolgen in Rückrichtung mit passenden Belegungen $\beta_j$ wählen anstatt expliziter Transitionfolgen.

**Definition 3.3** Symbolisches rückgerichtetes $k$-Lookahead-Grundspiel
Sei $\mathcal{A} = (V_\Sigma, f_\Sigma, Q, I, \delta, F)$ ein beliebiger SNBA. Dann wird das *symbolische rückgerichtete $k$-Lookahead-Grundspiel* $G_\mathcal{A}(q_0, \hat{q}_0)$ auf $\mathcal{A}$ gespielt, indem Spoiler und Duplikator symbolische Transitionsfolgen in Rückrichtung mit passenden Belegungen $\beta_j$ wählen.

Zu Beginn von Runde $i$ befindet sich das Spiel in der Konfiguration $(q_i, \hat{q}_i)$ und Spoiler beginnt:

1. Spoiler wählt eine Folge von $k$ zusammenhängenden symbolischen rückwärtsgerichteten Transitionen $q_i \xleftarrow{\varphi_i} q_{i+1} \xleftarrow{\varphi_{i+1}} \cdots \xleftarrow{\varphi_{i+k-1}} q_{i+k}$ sowie $k$ passende Belegungen $\beta_j$ mit $\forall i \leq j \leq i+k : \beta_j \vDash \varphi_j$ aus.

2. Duplikator wählt eine Zahl $1 \leq m \leq k$ aus und antwortet mit einer passenden Folge von $m$ rückwärtsgerichteten Transitionen $\hat{q}_i \xleftarrow{\hat{\varphi}_i} \hat{q}_{i+1} \xleftarrow{\hat{\varphi}_{i+1}} \cdots \xleftarrow{\hat{\varphi}_{i+m-1}} \hat{q}_{i+m}$ mit $\forall i \leq j \leq i+m : \beta_j \vDash \hat{\varphi}_j$.

3. Die verbleibenden, von Spoiler gewählten, Transitionen werden ignoriert und die neue Konfiguration ist $(q_{i+m}, \hat{q}_{i+m})$.

Für das rückgerichtete Grundspiel gelten die gleichen Hinweise zu ungültigen Belegungen wie auch für das vorwärtsgerichtete Grundspiel.

Hier ist jedoch die Rückwärtssenke $\bot_b$ unerheblich für die berechnete Simulationsrelation.

Aufbauend auf dem symbolischen $k$-Lookahead-Grundspiel können nun wie schon zuvor die symbolischen $x$-Simulationsspiele definiert werden. Wie bereits in Unterabschnitt 2.4.1 können die Definitionen 2.3, 2.5 und 2.6 unverändert auch auf die $k$-Lookahead-Simualtionen auf SNBAs übertragen werden.

Auf die gleiche Weise sind auch $k$-Lookahead-Simulationen aus Definition 2.12 ohne Anpassung auf SNBAs übertragbar.

### 3.2.2 Berechnung der $k$-Lookahead-Simulationen

Nachdem wir im vorherigen Abschnitt die Simulationen und $k$-Lookahead-Simulationen auf SNBAs übertragen haben, beschreiben wir nun, wie diese Relationen auf SNBAs berechnet werden können.

Wie zuvor beschrieben, gehen wir davon aus, dass die betrachteten Automaten vollständig sind. Sollte der Automat nicht vollständig sein, kann er wie folgt vervollständigt werden.

Sei $\mathcal{A} = (V_\Sigma, f_\Sigma, Q, I, \delta, F)$ ein beliebiger SNBA. Der vollständige SNBA kann berechnet werden, indem zwei Zustände $\bot_b$ und $\bot_f$ zur Zustandsmenge und entsprechende Transitionen von und zu allen anderen Zuständen hinzugefügt werden. Dann ergibt sich für den vollständigen SNBA $\dot{\mathcal{A}} := (V_\Sigma, f_\Sigma, Q', I, \delta', F)$ mit

$$Q' = Q \cup \{\bot_b, \bot_f\}$$

und $\delta' : Q' \times Q' \to B(V_\Sigma)$ mit

$$\delta'(q_i, q_j) = \delta(q_i, q_j) \vee (q_i = \bot_b) \vee (q_j = \bot_f)$$

Wir beginnen damit, den expliziten Vorgänger-Operator $\mathrm{CPre}^{\mathrm{di}} : 2^{Q^2} \to 2^{Q^2}$ aus Definition 2.13 in Unterabschnitt 2.4.2 auf SNBAs zu übertragen, dafür übertragen wir den $\mathrm{CPre}^{\mathrm{di}}$-Operator auf symbolische Transitonsfunktionen.

**Definition 3.4** Direkter Vorgänger-Operator auf SNBAs
Sei $\mathcal{A} = (V_\Sigma, Q, I, \delta, F)$ ein beliebiger SNBA und $X$ eine beliebige Menge von Konfigurationen im symbolischen direkten Simulationsspiel auf $\mathcal{A}$. Dann ist der *direkte Vorgänger-Operator auf SNBAs* $\mathrm{CPre}_\varphi^{\mathrm{di}} : 2^{Q^2} \to 2^{Q^2}$ definiert durch:

$$\begin{aligned}\mathrm{CPre}_\varphi^{\mathrm{di}}(X) = \Big\{ (q_0, \hat{q}_0) \;\Big|\; \exists \left( q_0 \xrightarrow{\beta_0} q_1 \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_{k-1}} q_k \right) : \\ \forall 0 < m \leq k : \\ \forall \left( \hat{q}_0 \xrightarrow{\beta_0} \hat{q}_1 \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_{m-1}} \hat{q}_m \right) : \\ (\exists 0 \leq j \leq m : q_j \in F \wedge \hat{q}_j \notin F) \\ \vee (q_m, \hat{q}_m) \in X \Big\}\end{aligned}$$

Der $\mathrm{CPre}_\varphi^{\mathrm{di}}$-Operator liefert die Menge der Konfigurationen des symbolischen direkten Simulationsspiels auf $\mathcal{A}$, aus denen Spoiler innerhalb von einer Runde das direkte Simulationsspiel gewinnt oder es in eine der Konfigurationen aus $X$ zwingen kann, ohne das Spiel zu verlieren.

Für die Berechnung des $\mathrm{CPre}_\varphi^{\mathrm{bw}}$-Operators auf SNBAs übertragen wir wie zuvor den expliziten Operator $\mathrm{CPre}^{\mathrm{bw}}$ aus Definition 2.14 auf symbolische Transitionen.

**Definition 3.5** Rückgerichteter Vorgänger-Operator auf SNBAs
Sei $\mathcal{A} = (V_\Sigma, Q, I, \delta, F)$ ein beliebiger SNBA und $X$ eine beliebige Menge von Konfigurationen des symbolischen direkten Simulationsspiels auf $\mathcal{A}$. Dann ist der *rückgerichtete Vorgänger-Operator auf SNBAs* $\mathrm{CPre}_\varphi^{\mathrm{bw}}$ :

$2^{Q^2} \to 2^{Q^2}$ definiert durch:

$$\begin{aligned}\mathrm{CPre}^{\mathrm{bw}}_{\varphi}(X) = \Big\{ (q_0, \hat{q}_0) \;\Big|\; \exists \Big( q_0 \xleftarrow{\beta_0} q_1 \xleftarrow{\beta_1} \cdots \xleftarrow{\beta_{k-1}} q_k \Big) : \\ \forall 0 < m \leq k : \\ \forall \Big( \hat{q}_0 \xleftarrow{\beta_0} \hat{q}_1 \xleftarrow{\beta_1} \cdots \xleftarrow{\beta_{m-1}} \hat{q}_m \Big) : \\ (\exists 0 \leq j \leq m : q_j \in F \wedge \hat{q}_j \notin F) \\ \vee (\exists 0 \leq j \leq m : q_j \in I \wedge \hat{q}_j \notin I) \\ \vee (q_m, \hat{q}_m) \in X \Big\}\end{aligned}$$

Der $\mathrm{CPre}^{\mathrm{bw}}_{\varphi}$-Operator berechnet die Menge der Konfigurationen des rückgerichteten Simulationsspiels auf SNBAs, aus denen Spoiler innerhalb von einer Runde das rückgerichtete Simulationsspiel gewinnt oder es in eine der Konfigurationen aus $X$ zwingen kann, ohne das Spiel zu verlieren.

Somit ergibt sich auch weiterhin für $x \in \{\mathrm{di}, \mathrm{bw}\}$ die Fixpunktgleichung

$$W^x = \mu(W \mapsto \mathrm{CPre}^x_{\varphi}(W)).$$

Abschließend übertragen wir auch den expliziten CPre Operator aus Definition 2.15 auf symbolische Transitionen. Auch der symbolische CPre Operator erweitert die beiden bisher betrachteten symbolischen Vorgänger-Operatoren um zwei Parameter $Y$ und $Z$.

**Definition 3.6** Vorgänger-Operator auf SNBAs
Sei $\mathcal{A} = (V_\Sigma, Q, I, \delta, F)$ ein beliebiger SNBA und $X$, $Y$ und $Z$ beliebige Mengen von Konfigurationen eines symbolischen Simulationsspiels auf $\mathcal{A}$. Dann ist der *Vorgänger-Operator auf SNBAs* $\mathrm{CPre}_{\varphi} : 2^{Q^2} \times 2^{Q^2} \times$

$2^{Q^2} \to 2^{Q^2}$ definiert durch:

$$\begin{aligned}\mathrm{CPre}_\varphi(X,Y,Z) = \Bigg\{(q_0,\hat{q}_0) \;\Bigg|\; \exists \left(q_0 \xrightarrow{\beta_0} q_1 \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_{k-1}} q_k\right) : \\ \forall 0 < m \leq k : \\ \forall \left(\hat{q}_0 \xrightarrow{\beta_0} \hat{q}_1 \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_{m-1}} \hat{q}_m\right) : \\ \left(\begin{array}{l} \exists 0 \leq j \leq m : q_j \in F \\ \wedge \forall 0 \leq j \leq m : \hat{q}_j \notin F \\ \wedge (q_m, \hat{q}_m) \in X \end{array}\right) \\ \vee \left(\begin{array}{l} \forall 0 \leq j \leq m : \hat{q}_j \notin F \\ \wedge (q_m, \hat{q}_m) \in Y \end{array}\right) \\ \vee \quad (q_m, \hat{q}_m) \in Z\Bigg\}\end{aligned}$$

Da sich nur die Form der betrachteten Transitionsfunktion geändert hat, gelten die Erklärungen zum expliziten CPre-Operator aus Unterabschnitt 2.4.2 unverändert auch für den symbolischen $\mathrm{CPre}_\varphi$-Operator. Daher gilt die Fixpunktgleichung der fairen $k$-Lookahead-Simulation weiterhin wie folgt

$$W^{\mathrm{f}} = \mu(Z \mapsto \nu(X \mapsto \mu(Y \mapsto \mathrm{CPre}_\varphi(X,Y,Z)))).$$

Und auch für die verzögerte $k$-Lookahead-Simulation bleibt die Fixpunktgleichung bestehen mit

$$W^{\mathrm{de}} = \mu(W \mapsto \mathrm{CPre}^1_\varphi(\nu(X \mapsto \mathrm{CPre}^2_\varphi(X,W)),W)).$$

Somit können wir nun auch auf SNBAs die $k$-Lookahead-Simulationen berechnen.

## 3.3 Minimierung von symbolischen nichtdeterministischen Büchi-Automaten mithilfe symbolischer $k$-Lookahead-Simulationen

Im vorherigen Unterabschnitt wurde die Berechnung der $k$-Lookahead-Simulationen auf SNBAs beschrieben. Um SNBAs zu minimieren fehlen noch die restlichen Definitionen und Verfahren aus Kapitel 2, daher werden wir in diesem Unterabschnitt die noch fehlenden Bausteine übertragen und orientieren uns dabei am Verlauf des Heavy-$k$ Verfahrens aus Abschnitt 2.5.4.

Die folgenden Reduktionsmethoden für symbolische NBAs ändern die Gültigkeit der verschiedenen Belegungen für $V_\Sigma$ nicht, daher wird $f_\Sigma$ wie auch $V_\Sigma$ in allen Definitionen unverändert übernommen.

### 3.3.1 Entfernen toter Zustände

Genauso wie bei einem expliziten NBA können wir auch aus einem symbolischen NBA die toten Zustände entfernen, ohne die Sprache des Automaten zu ändern.

In Kapitel 2 haben wir einen toten Zustand wie folgt definiert:

> Wir bezeichnen einen Zustand $q \in Q$ als *tot*, falls kein Pfad von einem initialen Zustand zu $q$ existiert oder von $q$ aus kein Pfad zu einem akzeptierenden Kreis in $\mathcal{A}$ existiert.

Offensichtlich hängt die Lebendigkeit eines Zustands nicht von den Transitionsbedinungen der vorhandenen Pfade ab, sondern nur von der Existenz einer Transitionsfolge mit erfüllbaren Bedingungen. Daher betrachten wir für die Lebendigkeit der Zustände nur die Kanten der Transitionen und nicht deren Bedingungen.

**Definition 3.7** $\delta$-Kanten
Sei $\mathcal{A} = (V_\Sigma, f_\Sigma, Q, I, \delta, F)$ ein beliebiger SNBA, dann ist $\delta_e \subseteq Q \times Q$ die Menge der *$\delta$-Kanten* von $\mathcal{A}$ ohne Beachtung der Kantenbeschriftung definiert als

$$\delta_e = \{(q_i, q_j) \mid \exists \beta \in \mathfrak{B}(V_\Sigma) : \beta \models \delta(q_i, q_j)\}.$$

Die Lebendigkeit von Zuständen des Automaten ist von der Existenz bestimmter Pfade abhängig, daher können wir sie auf die transitive Hülle $\delta_e^+$ der $\delta$-Kanten des Automaten zurückführen. Wenn $(q_i, q_j) \in \delta_e^+$, dann existiert in $\mathcal{A}$ ein Pfad von $q_i$ zu $q_j$ und wenn $(q_i, q_i) \in \delta_e^+$, dann existiert in $\mathcal{A}$ ein Pfad von $q_i$ zu $q_i$, der einen Kreis bildet.

Damit können wir die Menge $Q_l \subseteq Q$ der lebendigen Zustände eines Automaten $\mathcal{A} = (V_\Sigma, f_\Sigma, Q, I, \delta, F)$ wie folgt beschreiben:

$$\begin{aligned} Q_l = \{q_i \in Q \mid \quad & \exists q_j \in Q : (\, q_j \in F \wedge (q_j, q_j) \in \delta_e^+ \wedge \\ & ((q_i, q_j) \in \delta_e^+ \vee q_i = q_j)) \\ & \wedge \exists q_h \in Q : (q_h \in I \wedge ((q_h, q_i) \in \delta_e^+ \vee q_i = q_h)\}. \end{aligned} \tag{3.1}$$

Die erste Bedingung beschreibt, dass ein Pfad von $q_i$ zu einem akzeptierenden Zustand $q_j$ existiert und dass $q_j$ Teil eines Kreises ist. Die zweite Bedingung sagt aus, dass $q_i$ von einem initialen Zustand $q_h$ aus erreichbar ist.

Nachdem mit $Q_l$ die lebendigen Zustände des Automaten bestimmt sind, können wir den lebendigen Automanten als den Automaten ohne die toten Zustände definieren.

**Definition 3.8** Lebendiger Automat
Sei $\mathcal{A} = (V_\Sigma, f_\Sigma, Q, I, \delta, F)$ ein beliebiger SNBA. Der *symbolische lebendige Automat* ist dann definiert als $\text{Living}(\mathcal{A}) := (V_\Sigma, f_\Sigma, Q', I', \delta', F')$ mit

$$\begin{aligned} Q' &= Q_l \\ I' &= I \cap Q_l \\ F' &= F \cap Q_l \end{aligned}$$

und $\delta' : Q' \times Q' \to B(V_\Sigma)$ mit

$$\delta'(q_i, q_j) = \delta(q_i, q_j).$$

### 3.3.2 Stutzen der Transitionen

In Unterabschnitt 2.5.2 haben wir die Transitionen der expliziten Automaten gestutzt, indem wir ganze Transitionstupel aus der Transitionsrelation entfernt haben. Wir werden symbolische Transitionen stutzen, indem wir die Anzahl der Modelle der Transitionsbedingungen reduzieren.

Da die Pruning-Relationen aus Definition 2.18 auf den Tupeln der Transitionsrelation $\Delta$ definiert sind, müssen wir bei der Übertragung auf SNBAs die symbolischen Transitionen des Automaten auf die durch sie beschriebenen $\beta$-Transitionen abbilden. Im Folgenden beschreiben wir mit $\delta_\beta \subseteq Q \times Q \times \mathfrak{B}(V_\Sigma)$ die Menge der durch $\delta$ beschriebenen $\beta$-Transitionen mit

$$\delta_\beta = \{(q_i, q_j, \beta_i) \mid \beta_i \models \delta(q_i, q_j)\}.$$

Somit können wir nun eine Pruning-Relation über den $\beta$-Transitionen des Automaten definieren.

**Definition 3.9** Pruning-Relation
Sei $\mathcal{A} = (V_\Sigma, f_\Sigma, Q, I, \delta, F)$ ein beliebiger SNBA und $R_b$ und $R_f$ zwei Binärrelationen auf $Q$, dann ist eine *Pruning-Relation* definiert als die Relation $P(R_b, R_f) \subseteq \delta_\beta \times \delta_\beta$ mit

$$P(R_b, R_f) = \left\{ \Big( (q_i, q_j, \beta), (\hat{q}_i, \hat{q}_j, \beta) \Big) \;\Big|\; q_i \; R_b \; \hat{q}_i \wedge q_j \; R_f \; \hat{q}_j \right\}.$$

Zwei $\beta$-Transitionen stehen in einer Pruning-Relation, wenn die erste Transition durch die zweite Transition überdeckt wird. Dann kann die erste Transition entfernt werden, ohne dass sich Auswirkungen auf die akzeptierte Sprache des Automaten ergeben.

**Definition 3.10** Symbolischer gestutzter Automat
Sei $\mathcal{A} = (V_\Sigma, f_\Sigma, Q, I, \delta, F)$ ein beliebiger SNBA und $P$ eine transitive und asymmetrische Pruning-Relation auf den $\beta$-Transitionen von $\delta$. Der *symbolische gestutzte Automat* ist dann definiert durch

$$\text{Prune}(\mathcal{A}, P) := (V_\Sigma, f_\Sigma, Q, I, \delta', F) \text{ mit}$$

$$\delta'(q_i, q_j) = \bigvee_{\beta \in B} c_\Sigma(\beta) \text{ mit } B = \{\beta \in \mathfrak{B}(V_\Sigma) \mid \beta \models \delta(q_i, q_j) \wedge \nexists(\hat{q}_i, \hat{q}_j, \hat{\beta}) \in \delta_\beta : (q_i, q_j, \beta)\ P\ (\hat{q}_i, \hat{q}_j, \hat{\beta})\}.$$

### 3.3.3 Quotientenbildung

Auch die Quotientenbildung lässt sich einfach auf die symbolischen NBAs übertragen, um so die Anzahl der Knoten und Kanten des Automaten zu reduzieren.

Wie bereits in Unterabschnitt 2.5.3 betrachten wir im Folgenden einen SNBA $\mathcal{A} = (V_\Sigma, f_\Sigma, Q, I, \delta, F)$ und eine beliebige Quasiordnung $\sqsubseteq \subseteq Q \times Q$. Durch die Quasiordnung $\sqsubseteq$ wird eine Äquivalenzrelation $\equiv = \sqsubseteq \cap \sqsupseteq$ induziert. Damit lässt sich auch auf SNBAs der Quotient eines Automaten bestimmen.

**Definition 3.11** Quotientenautomat
Sei $\mathcal{A} = (V_\Sigma, f_\Sigma, Q, I, \delta, F)$ ein beliebiger SNBA und $\sqsubseteq$ eine Quasiordnung auf $Q$. Der *Quotientenautomat* zu $\mathcal{A}$ bezüglich $\sqsubseteq$ ist dann definiert als $\mathcal{A}/\sqsubseteq = (V_\Sigma, f_\Sigma, Q', I', \delta', F')$ mit

$$\begin{aligned} Q' &= [Q], \\ I' &= \{[q] \in Q' \mid \exists q' \in I : q' \in [q]\}, \\ F' &= \{[q] \in Q' \mid \exists q' \in F : q' \in [q]\} \end{aligned}$$

und für $\delta' : Q' \times Q' \to B(V_\Sigma)$ gilt:

$$\delta'([q_i],[q_j]) = \bigvee_{\varphi \in \Phi} \varphi \text{ mit } \Phi = \{\varphi \in B(V_\Sigma) \mid \exists q_i' \in [q_i], q_j' \in [q_j] : \delta(q_i', q_j') = \varphi\}$$

### 3.3.4 Minimierungsverfahren auf SNBAs

Nachdem in den vorherigen Abschnitten die Berechnung von symbolischen $k$-Lookahead-Simulationen beschrieben und die verschiedenen Verfahren zur Reduktion von NBAs auf SNBAs übertragen wurden, wird in diesem Abschnitt die Minimierung symbolischer NBAs beschrieben.

Grundlegend werden die Verfahren aus Unterabschnitt 2.5.4 übernommen, jedoch dienen die symbolischen Varianten der $k$-Lookahead-Simulationen als Grundlage:

- symbolische direkte $k$-Lookahead-Simulation $\preceq^{k\text{-di}}$
- symbolische verzögerter $k$-Lookahead-Simulation $\preceq^{k\text{-de}}$
- symbolische faire $k$-Lookahead-Simulation $\preceq^{k\text{-f}}$
- symbolische rückgerichtete $k$-Lookahead-Simulation $\preceq^{k\text{-bw}}$

Ebenso werden die, auf SNBAs übertragenen, Minimierungstechniken statt der Techniken für explizite NBAs verwendet.

Für die Quotientenbildung werden die symbolische verzögerte $k$-Lookahead-Simulation $\preceq^{k\text{-de}}$ und symbolische rückgerichtete $k$-Lookahead-Simulation $\preceq^{k\text{-bw}}$ verwendet. Ansonsten ist das Vorgehen sowohl für Heavy-$k$ als auch für Light-$k$ identisch mit den Verfahren für explizite NBAs.

Die Hinweise zur Korrektheit der Minimierungstechniken aus Unterabschnitt 2.5.4 gelten entsprechend auch für die Minimierung symbolischer NBAs, da sich nur die Form der Transitionsbedingungen geändert hat.

## 3.4 Vollständig symbolisch repräsentierte nichtdeterministische Büchi-Automaten (FSNBAs)

In Abschnitt 3.1 haben wir beschrieben, wie die Transitionsfunktion eines NBAs durch eine symbolische Repräsentation der Transitionsbedingungen prägnanter spezifiziert werden kann.

Die Berechnung der $k$-Lookahead-Simulationen auf den symbolischen NBAs hängt jedoch von den $\beta$-Transitionen des Automaten ab. Diese entsprechen genau den Transitionen des expliziten NBAs aus $\Delta$, wodurch ihre Berechnung einer Explizitmachung des SNBAs gleichkommt. Die Explizitmachung der Transitionsfunktion macht jedoch den Vorteil der präzisen Beschreibbarkeit gegenüber der expliziten Transitionsrelation zunichte. In diesem Abschnitt wollen wir daher die symbolische Mengenrepräsentation auch auf die Zustände des Automaten ausweiten und erhalten dadurch ein Automatenmodell, das nur noch durch aussagenlogische Formeln beschrieben wird. Dies ermöglicht es uns, Techniken für ihre Verarbeitung einzusetzen, die bereits aus dem symbolischen Model-Checking bekannt sind [McM93].

Um einen SNBA $\mathcal{A} = (V_\Sigma, f_\Sigma, Q, I, \delta, F)$ vollständig symbolisch repräsentieren zu können, benötigen wir neben der Repräsentationsdomäne $V_\Sigma = \{c_0^0, c_1^0, \ldots, c_{m-1}^0\}$ mit $|V_\Sigma| = m$ noch zwei weitere Repräsentationsdomänen $V_Q^0 = \{v_0^0, v_1^0, \ldots, v_{n-1}^0\}$ und $V_Q^1 = \{v_0^1, v_1^1, \ldots, v_{n-1}^1\}$ mit $|V_Q^0| = |V_Q^1| = \lceil \log_2 |Q| \rceil = n$ Variablen, um die Zustände aus $Q$ zu beschreiben. Dabei ist zu beachten, dass alle Zustände über allen Repräsentationsdomänen in gleicher Weise kodiert werden. Im Folgenden beschreiben wir mit $\mathbf{q}_0$ und $\mathbf{q}_1$ Elemente aus $\mathbf{V}_Q^0$ bzw. $\mathbf{V}_Q^1$ und mit $\mathbf{c}$ Elemente aus $\mathbf{V}_\Sigma$.

Da $|Q| \leq 2^n$ gilt, können durch $V_Q^0$ und $V_Q^1$ potentiell mehr Zustände beschrieben werden als überhaupt in $Q$ vorhanden sind. Daher muss auch in diesem Fall eine boolesche Funktion $f_Q : \mathbf{V}_Q^0 \to \mathbb{B}$ angegeben werden, die die gültigen Belegungen für $V_Q^0$ bzw. $V_Q^1$ beschreibt. Die

Funktion wird bestimmt als

$$f_Q(\mathbf{q}_0) = f_Q(v_0^0, v_1^0, \dots, v_{n-1}^0) = \bigvee_{q \in Q} c_Q^0(q).$$

Zuerst wollen wir die Menge der initialen Zustände $I$ symbolisch beschreiben und bestimmen dafür die boolesche Funktion $f_I : \mathbf{V}_Q^0 \to \mathbb{B}$ mit

$$f_I(\mathbf{q}_0) = f_I(v_0^0, v_1^0, \dots, v_{n-1}^0) = \bigvee_{q \in I} c_Q^0(q).$$

Die Funktion $f_I$ liefert für jedes über $V_Q^0$ kodierte Element von $Q$ den Wert der charakteristischen Funktion $\chi_I$ von $I$, so dass $\chi_I(q) \iff f_I(c_Q(q))$ gilt.

Auf gleiche Weise können wir die Teilmenge der akzeptierenden Zustände $F$ von $Q$ durch die boolesche Funktion $f_F : \mathbf{V}_Q^0 \to \mathbb{B}$ beschreiben:

$$f_F(\mathbf{q}_0) = f_F(v_0^0, v_1^0, \dots, v_{n-1}^0) = \bigvee_{q \in F} c_Q^0(q)$$

Auch hier gilt $\chi_F(q) \iff f_F(c_Q(q))$.

Um die Transitionsfunktion $\delta : Q \times Q \to B(V_\Sigma)$ zu beschreiben, betrachten wir wieder die durch $\delta$ beschriebenen $\beta$-Transitionen $\delta_\beta$ als Teilmenge von $Q \times Q \times \mathfrak{B}(V_\Sigma)$. Dadurch können wir nun wie zuvor die boolesche Funktion $f_\delta : \mathbf{V}_Q^0 \times \mathbf{V}_Q^1 \times \mathbf{V}_\Sigma \to \mathbb{B}$ definieren mit

$$f_\delta(\mathbf{q}_0, \mathbf{q}_1, \mathbf{c}) = \bigvee_{q_0, q_1 \in Q} c_Q^0(q_0) \wedge c_Q^1(q_1) \wedge \delta(q_0, q_1).$$

Somit können wir nun den vollständig symbolisch repräsentierten NBA definieren.

**Definition 3.12** Vollsymbolischer nichtdeterministischer Büchi-Automat
Ein *vollsymbolischer nichtdeterministischer Büchi-Automat (FSNBA)* ist ein 8-Tupel $\mathcal{A} = (V_\Sigma^0, V_Q^0, V_Q^1, f_\Sigma, f_Q, f_I, f_\delta, f_F)$, wobei

- $V_\Sigma^0$ die *Repräsentationsdomäne* von $\Sigma$,
- $V_Q^0$ die erste Repräsentationsdomäne der *Zustände*,

- $V_Q^1$ die zweite Repräsentationsdomäne der *Zustände*,
- $f_\Sigma : \mathbf{V}_\Sigma \to \mathbb{B}$ eine boolesche Funktion, die die *gültigen Belegungen* von $V_\Sigma$ beschreibt,
- $f_Q : \mathbf{V}_Q^0 \to \mathbb{B}$ eine boolesche Funktion, die die *gültigen Belegungen* von $V_Q^0$ und $V_Q^1$ beschreibt,
- $f_I : \mathbf{V}_Q^0 \to \mathbb{B}$ eine boolesche Funktion, die die *initialen* Zustände von $\mathcal{A}$ beschreibt,
- $f_\delta : \mathbf{V}_Q^0 \times \mathbf{V}_Q^1 \times \mathbf{V}_\Sigma^0 \to \mathbb{B}$ eine boolesche Funktion, die die Transitionen des Automaten $\mathcal{A}$ beschreibt, und
- $f_F : \mathbf{V}_Q^0 \to \mathbb{B}$ eine boolesche Funktion ist, die die *akzeptierenden* Zustände von $\mathcal{A}$ beschreibt.

Wie bereits NBAs und SNBAs können FSNBAs grafisch repräsentiert werden, jedoch geht die intuitive Verständlichkeit der Repräsentation verloren. Abbildung 3.2 zeigt den bereits bekannten Automaten, diesmal jedoch als FSNBA. Die Belegungen von $V_Q^0$ werden als Zustände des Automaten interpretiert und die Transitionsbedingungen bleiben wie beim SNBA bestehen. Wir verwenden $x$ bzw. $\overline{x}$ um eine Belegung $\beta$ mit $\beta(x) = \top$ bzw. $\beta(x) = \bot$ darzustellen.

Die Anfangskonfiguration $\left(\overline{v_0^0}\right)$ des Automaten beschreibt den ersten Zustand und $f_\delta$ kann für $\beta(v_0^0) = \bot$ auf vier Arten erfüllt werden. Gültige Belegungen für $V_Q^0$, $V_Q^1$ und $V_\Sigma^0$ sind

$$\left(\overline{v_0^0}, \overline{v_1^0}, c_0^0, \overline{c_1^0}\right), \left(\overline{v_0^0}, \overline{v_1^0}, c_0^0, c_1^0\right), \left(\overline{v_0^0}, v_1^0, \overline{c_0^0}, c_1^0\right) \text{ und } \left(\overline{v_0^0}, v_1^0, c_0^0, c_1^0\right).$$

Ist die Konfiguration hingegen $\left(v_0^0\right)$, so sind die Belegungen, die $f_\delta$ erfüllen

$$\left(v_0^0, v_1^0, \overline{c_0^0}, \overline{c_1^0}\right), \left(v_0^0, v_1^0, c_0^0, \overline{c_1^0}\right), \left(v_0^0, v_1^0, \overline{c_0^0}, c_1^0\right) \text{ und } \left(v_0^0, v_1^0, c_0^0, c_1^0\right).$$

Der Automat kann ausgeführt werden, indem in $f_\delta$ die Variablen in $V_Q^0$ und $V_\Sigma^0$ fest belegt werden und die verbleibenden freien Variablen aus $V_Q^1$ als Beschreibung der Folgekonfiguration interpretiert werden.

$$V_\Sigma = \{c_0^0, c_1^0\}$$

$$\begin{aligned}
V_Q^0 &= \{v_0^0\} \\
V_Q^1 &= \{v_0^1\} \\
f_\Sigma(c_0^0, c_1^0) &= \text{true} \\
f_Q(v_0^0) &= \text{true} \\
f_I(v_0^0) &= \neg v_0^0 \\
f_\delta(v_0^0, v_0^1, c_0^0, c_1^0) &= \begin{array}{r} (\neg v_0^0 \wedge \neg v_0^1 \wedge c_0^0) \\ \vee (\neg v_0^0 \wedge \quad v_0^1 \wedge c_1^0) \\ \vee (\quad v_0^0 \wedge \quad v_0^1) \end{array} \\
f_F(v_0^0) &= v_0^0
\end{aligned}$$

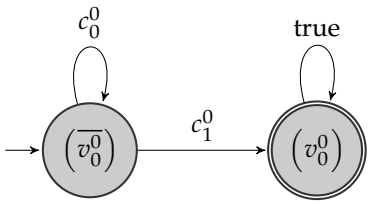



Abbildung 3.2: Beispiel für einen einfachen FSNBA $\mathcal{A}$ in vollsymbolischer und graphischer Darstellung. Unter Verwendung der Kodierungsfunktion aus Beispiel 3.1 akzeptiert der Automat nach Umkodierung auf explizite Worte die gleiche Sprache $\{uvw \in \Sigma^\omega \mid u \in \{b,d\}^*, v \in \{c,d\} \text{ und } w \in \Sigma^\omega\}$ wie der Automat aus Abbildung 2.1 und Abbildung 3.2.

$$V_\Sigma = \{c_0^0, c_1^0\}$$
$$V_Q^0 = \{v_0^0, v_1^0\}$$
$$V_Q^1 = \{v_0^1, v_1^1\}$$
$$f_\Sigma(c_0^0, c_1^0) = \neg c_0^0 \vee \neg c_1^0$$
$$f_Q(v_0^0, v_1^0) = \neg v_0^0 \vee \neg v_1^0$$
$$f_I(v_0^0, v_1^0) = \neg v_0^0 \wedge \neg v_1^0$$
$$f_F(v_0^0) = \neg v_0^0$$

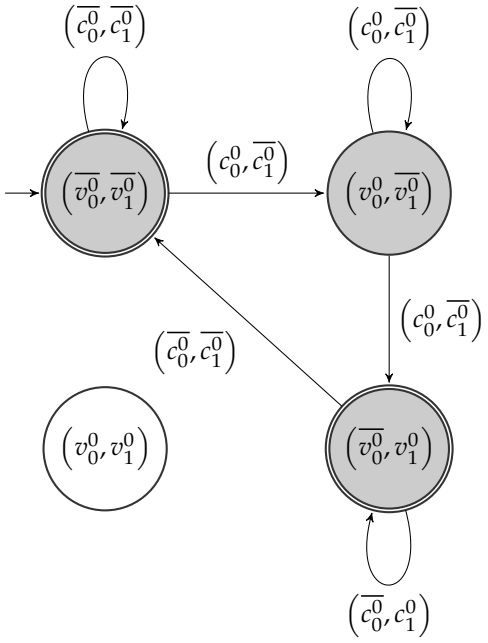



Abbildung 3.3: Ein FSNBA $\mathcal{A}$ in vollsymbolischer und grafischer Repräsentation, der Ausgabesequenzen eines unendlichen Modulo-3-Zählers akzeptiert. Die Beschreibung der Transitionsfunktion entfällt aus Gründen der Übersichtlichkeit, sie ergibt sich aus der grafischen Repräsentation. Der Knoten $\left(v_0^0, v_1^0\right)$ stellt keinen Zustand des Automaten dar.

Der Automat in Abbildung 3.3 verdeutlicht noch einmal die Begründung für $f_\Sigma$ und $f_Q$. Der Automat akzeptiert für

$$c_\Sigma(0) = (\bot, \bot) \quad c_\Sigma(1) = (\top, \bot) \quad c_\Sigma(2) = (\bot, \top)$$

die Ausgabesequenzen eines unendlichen Modulo-3-Zählers. Da keine Eingabe existiert, die durch $(\top, \top)$ kodiert wird, erfüllt diese Belegung $f_\Sigma$ nicht und muss daher bei der Rückübersetzung in NBAs nicht betrachtet werden. Gleiches gilt für die Belegung $(\top, \top)$ für $V_Q^0$, die einen nicht vorhandenen Zustand des Automaten kodiert.

## 3.5 Berechnung der $k$-Lookahead-Simulationen auf FSNBAs.

Nachdem in Unterabschnitt 3.2.1 die $k$-Lookahead-Simulationen auf SNBAs übertragen wurden, widmen wir uns in diesem und den folgenden Abschnitten der vollständig symbolischen Berechnung dieser Simulationsrelationen.

Um die Vorgänger-Operatoren vollsymbolisch zu beschreiben, werden wir sie durch Formeln der quantifizierten Aussagenlogik beschreiben [Bub09]. Quantifizierte Aussagenlogik erweitert die bekannte Aussagenlogik um die Möglichkeit über den Werten von aussagenlogischen Variablen zu quantifizieren.

### 3.5.1 Direkte und rückgerichtete $k$-Lookahead-Simulation

Um für $x \in \{\mathrm{di}, \mathrm{bw}\}$ die Relation $\sqsubseteq^{k\text{-}x}$ symbolisch zu berechnen, müssen wir die Komplement-Relation $W^x = (Q \times Q) \setminus \sqsubseteq^{k\text{-}x}$ symbolisch berechnen. Dafür definieren wir zunächst den vollsymbolischen Vorgänger-Operator $\mathrm{CPre}^x_f(f_X)$ für eine beliebige Funktion $f_X : \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}$, die eine Menge von Konfigurationen beschreibt. Intuitiv können wir $\mathrm{CPre}^x_f(f_X)$ verstehen als die symbolische Funktion $f_{\mathrm{CPre}^x_\varphi(X)}$, die die charakteristische Funktion von $\mathrm{CPre}^x_\varphi(X)$ symbolisch beschreibt, wenn $f_X$ die Menge $X$ beschreibt.

Wir betrachten noch einmal den Vorgänger-Operator auf SNBAs $\mathrm{CPre}^{\mathrm{di}}_\varphi$

aus Definition 3.4.

$$\mathrm{CPre}^{\mathrm{di}}_{\varphi}(X) = \Bigg\{ (q_0, \hat{q}_0) \;\Bigg|\; \exists \left( q_0 \xrightarrow{\beta_0} q_1 \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_{k-1}} q_k \right) : \forall 0 < m \leq k : \forall \left( \hat{q}_0 \xrightarrow{\beta_0} \hat{q}_1 \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_{m-1}} \hat{q}_m \right) : \underbrace{(\exists 0 \leq j \leq m : q_j \in F \wedge \hat{q}_j \notin F)}_{e_1} \vee \underbrace{(q_m, \hat{q}_m) \in X}_{e_2} \Bigg\}$$

$e_3$ (über $\vee$), $e_4$ (über $\forall$)

Um den Operator zu symbolisieren, werden wir die charakteristische Funktion schrittweise umwandeln. Um die Zustände $q_0, q_1, \ldots, q_k \in Q$ korrekt und unabhängig voneinander kodieren zu können, führen wir weitere Repräsentationsdomänen $V^2_Q, V^3_Q, \ldots, V^k_Q$ ein. Dasselbe gilt für die Domänen $\hat{V}^0_Q, \hat{V}^1_Q, \ldots, \hat{V}^k_Q$, die die Zustände $\hat{q}_0, \hat{q}_1, \ldots, \hat{q}_k \in Q$ kodieren. Weiterhin führen wir $V^0_\Sigma, V^2_\Sigma, \ldots, V^{k-1}_\Sigma$ mit $V_\Sigma = V^0_\Sigma$ ein, um die Belegungen $\beta_i \in \mathfrak{B}(V_\Sigma)$ zu repräsentieren.

Wir beginnen mit der Umwandlung von Ausdruck $e_1$, in dem die $q_j$ und $\hat{q}_j$ frei vorkommen.

$$e_1 = \exists 0 \leq j \leq m : q_j \in F \wedge \hat{q}_j \notin F \tag{3.2}$$

Da $m$ endlich ist, können wir den Existenzquantor über dem Index $j$ eliminieren, indem wir ihn durch eine Disjunktion über dem Index $j$ ersetzen.

$$= \bigvee_{j=0}^{m} q_j \in F \wedge \hat{q}_j \notin F$$

Abschließend ersetzen wir die $q_j, \hat{q}_j$ und die charakteristische Funktion von $F$ durch ihre symbolischen Entsprechungen und erhalten:

$$\tilde{e}_1 = \bigvee_{j=0}^{m} f_F(\mathbf{q}_j) \wedge \neg f_F(\hat{\mathbf{q}}_j) \tag{3.3}$$

In der aussagenlogischen Formel $\tilde{e_1}$ kommen die $\mathbf{q}_j$ und $\hat{\mathbf{q}}_j$ frei vor.

Als nächstes wandeln wir den Ausdruck $e_2$ um, in dem $q_m$ und $\hat{q}_m$ frei vorkommen.

$$e_2 = (q_m, \hat{q}_m) \in X \tag{3.4}$$

Es reicht aus, die charakteristische Funktion von $X$ und $q_m$ bzw. $\hat{q}_m$ zu ersetzen.

$$\tilde{e_2} = f_X(\mathbf{q}_m, \hat{\mathbf{q}}_m) \tag{3.5}$$

Wir erhalten die aussagenlogische Formel $\tilde{e_2}$, in der $\mathbf{q}_m$ und $\hat{\mathbf{q}}_m$ frei vorkommen.

Darauf aufbauend wandeln wir den Ausdruck $e_3$ um, der sowohl $e_1$ als auch $e_2$ enthält. In $e_3$ kommen die $q_j$ sowie $\hat{q}_0$ frei vor.

$$e_3 = \forall 0 < m \leq k : \forall \left( \hat{q}_0 \xrightarrow{\beta_0} \hat{q}_1 \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_{m-1}} \hat{q}_m \right) : e_1 \vee e_2 \tag{3.6}$$

Wir beginnen, indem wir die Transitionsfolge $\left(\hat{q}_0 \rightarrow^{\beta_0} \cdots \rightarrow^{\beta_{m-1}} \hat{q}_m\right)$ aufbrechen und durch eine Konjunktion ersetzen.

$$= \forall 0 < m \leq k : \forall \hat{q}_1, \ldots, \hat{q}_m \in Q : \left( \bigwedge_{i=1}^{m} \hat{q}_{i-1} \xrightarrow{\beta_{i-1}} \hat{q}_i \in \delta \right) \rightarrow (e_1 \vee e_2)$$

Danach ersetzen wir die Notation $\hat{q}_{i-1} \rightarrow^{\beta_{i-1}} \hat{q}_i$ durch $\beta_{i-1} \models \delta(\hat{q}_{i-1}, \hat{q}_i)$.

$$= \forall 0 < m \leq k : \forall \hat{q}_1, \ldots, \hat{q}_m \in Q : \left( \bigwedge_{i=1}^{m} \beta_{i-1} \models \delta(\hat{q}_{i-1}, \hat{q}_i) \right) \rightarrow (e_1 \vee e_2)$$

Jetzt können wir den Allquantor über $m$ durch eine Konjunktion ersetzen, da auch $k$ endlich ist.

$$= \bigwedge_{m=1}^{k} \forall \hat{q}_1, \ldots, \hat{q}_m \in Q : \left( \bigwedge_{i=1}^{m} \beta_{i-1} \models \delta(\hat{q}_{i-1}, \hat{q}_i) \right) \rightarrow (e_1 \vee e_2)$$

Abschließend ersetzen wir die $q_j$, $\hat{q}_j$, $e_1$ und $e_2$ durch ihre symbolischen Entsprechungen.

$$\tilde{e}_3 = \bigwedge_{m=1}^{k} \forall \hat{\mathbf{q}}_1, \dots, \hat{\mathbf{q}}_m : \left( \bigwedge_{i=1}^{m} f_\delta(\hat{\mathbf{q}}_{i-1}, \hat{\mathbf{q}}_i, \mathbf{c}_{i-1}) \right) \rightarrow (\tilde{e}_1 \vee \tilde{e}_2) \tag{3.7}$$

Wir erhalten die aussagenlogische Formel $\tilde{e}_3$, in der die $\mathbf{q}_j$ sowie $\hat{\mathbf{q}}_0$ frei vorkommen.

Abschließend können wir den Ausdruck $e_4$ umwandeln, der $q_0$ und $\hat{q}_0$ frei enthält.

$$e_4 = \exists \left( q_0 \xrightarrow{\beta_0} q_1 \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_{k-1}} q_k \right) : e_3 \tag{3.8}$$

Auch hier beginnen wir damit, die Transitionsfolge $\left( q_0 \rightarrow^{\beta_0} \cdots \rightarrow^{\beta_{k-1}} q_k \right)$ durch eine Konjunktion zu ersetzen.

$$\begin{aligned} = \; & \exists q_1, q_2, \dots, q_k \in Q : \exists \beta_0, \beta_1, \dots, \beta_k \in \mathfrak{B}(V_\Sigma) : \\ & \left( \bigwedge_{i=1}^{k} \beta_{i-1} \vDash \delta(q_{i-1}, q_i) \right) \wedge e_3 \end{aligned}$$

Nun ersetzen wir auch in $e_4$ die Vorkommen der expliziten Variablen und Funktionen durch ihre symbolischen Entsprechungen und erhalten:

$$\begin{aligned} \tilde{e}_4 = \; & \exists \mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_k : \exists \mathbf{c}_0, \mathbf{c}_1, \dots, \mathbf{c}_k : \\ & \left( \bigwedge_{i=1}^{k} f_\delta(\mathbf{q}_{i-1}, \mathbf{q}_i, \mathbf{c}_{i-1}) \right) \wedge \tilde{e}_3 \end{aligned} \tag{3.9}$$

Nach der Umwandlung enthält die aussagenlogische Formel $\tilde{e}_4$ die beiden Bitvektoren $\mathbf{q}_0$ und $\hat{\mathbf{q}}_\mathbf{0}$ ungebunden.

Auf Basis der Ergebnisse aus (3.3), (3.5), (3.7) und (3.9) definieren wir den vollsymbolischen Vorgänger-Operator $\mathrm{CPre}_f^{\mathbf{di}}$.

**Definition 3.13** Vollsymbolischer direkter Vorgänger-Operator
Sei $\mathcal{A} = (V_\Sigma, V_Q^0, V_Q^1, f_\Sigma, f_Q, f_I, f_\delta, f_F)$ ein beliebiger FSNBA. Dann ist

der *vollsymbolische direkte Vorgänger-Operator* $\mathrm{CPre}_f^{\mathrm{di}} : (\mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}) \to (\mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B})$ definiert als

$$\begin{aligned}\mathrm{CPre}_f^{\mathrm{di}}(f_X) = \exists \mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_k : \exists \mathbf{c}_0, \mathbf{c}_1, \dots, \mathbf{c}_k : \\ \left(\bigwedge_{i=1}^{k} f_\delta(\mathbf{q}_{i-1}, \mathbf{q}_i, \mathbf{c}_{i-1})\right) \wedge \\ \bigwedge_{m=1}^{k} \forall \hat{\mathbf{q}}_1, \dots, \hat{\mathbf{q}}_m : \left(\bigwedge_{i=1}^{m} f_\delta(\hat{\mathbf{q}}_{i-1}, \hat{\mathbf{q}}_i, \mathbf{c}_{i-1})\right) \to \\ \left(\bigvee_{j=0}^{m} f_F(\mathbf{q}_j) \wedge \neg f_F(\hat{\mathbf{q}}_j)\right) \\ \vee \quad f_X(\mathbf{q}_m, \hat{\mathbf{q}}_m)\end{aligned}$$

Für die Bestimmung des vollsymbolischen rückgerichteten Vorgänger-Operators $\mathrm{CPre}_f^{\mathrm{bw}}$-Operators betrachten wir noch einmal den Operator $\mathrm{CPre}_\varphi^{\mathrm{bw}}$ aus Definition 2.14 auf symbolischen NBAs.

$$\begin{aligned}\mathrm{CPre}_\varphi^{\mathrm{bw}}(X) = \Big\{(q_0, \hat{q}_0) \;\Big|\; \exists \left(q_0 \xleftarrow{\beta_0} q_1 \xleftarrow{\beta_1} \cdots \xleftarrow{\beta_{k-1}} q_k\right) : \\ \forall 0 < m \leq k : \\ \forall \left(\hat{q}_0 \xleftarrow{\beta_0} \hat{q}_1 \xleftarrow{\beta_1} \cdots \xleftarrow{\beta_{m-1}} \hat{q}_m\right) : \\ \underbrace{(\exists 0 \leq j \leq m : q_j \in F \wedge \hat{q}_j \notin F)}_{e_1} \\ \vee \underbrace{\left(\exists 0 \leq j \leq m : q_j \in I \wedge \hat{q}_j \notin I\right)}_{e2} \\ \vee \underbrace{(q_m, \hat{q}_m) \in X}_{e_3}\Big\} \\ \underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{e_4} \\ \underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{e_5}\end{aligned}$$

Wir können die Erkenntnisse über (3.2) und (3.4) aus $\mathrm{CPre}_f^{\mathrm{di}}$ mit geringen

Anpassungen wiederverwenden und müssen nur für $e_2$, $e_4$ und $e_5$ die symbolischen Entsprechungen bestimmen.

Wir beginnen mit dem Ausdruck $e_2$, in dem die $q_j$ und $\hat{q}_j$ frei vorkommen.

$$e_2 = \exists 0 \leq j \leq m : q_j \in I \wedge \hat{q}_j \notin I$$

Da $m$ endlich ist, können wir den Existenzquantor über dem Index $j$ eliminieren, indem wir ihn durch eine Disjunktion über dem Index $j$ ersetzen.

$$= \bigvee_{j=0}^{m} q_j \in I \wedge \hat{q}_j \notin I$$

Abschließend ersetzen wir die $q_j, \hat{q}_j$ und die charakteristische Funktion von $I$ durch ihre symbolischen Entsprechungen und erhalten:

$$\tilde{e}_2 = \bigvee_{j=0}^{m} f_I(\mathbf{q}_j) \wedge \neg f_I(\hat{\mathbf{q}}_j) \tag{3.10}$$

In $\tilde{e}_2$ kommen die $\mathbf{q}_j$ und $\hat{\mathbf{q}}_j$ frei vor.

Darauf und auf den Ergebnissen aus (3.2) und (3.4) aufbauend wandeln wir den Ausdruck $e_4$ um, der $e_1$, $e_2$ und $e_3$ enthält. In $e_4$ kommen die $q_j$ sowie $\hat{q}_0$ frei vor.

$$e_4 = \forall 0 < m \leq k : \forall \left( \hat{q}_0 \xleftarrow{\beta_0} \hat{q}_1 \xleftarrow{\beta_1} \cdots \xleftarrow{\beta_{m-1}} \hat{q}_m \right) : e_1 \vee e_2 \vee e_3$$

Wir beginnen, indem wir die Transitionsfolge $\left( \hat{q}_0 \leftarrow^{\beta_0} \cdots \leftarrow^{\beta_{m-1}} \hat{q}_m \right)$ aufbrechen und durch eine Konjunktion ersetzen.

$$\begin{aligned} = \forall 0 < m \leq k : \forall \hat{q}_1, \ldots, \hat{q}_m \in Q : \\ \left( \bigwedge_{i=1}^{m} \beta_{i-1} \vDash \delta(\hat{q}_i, \hat{q}_{i-1}) \right) \rightarrow (e_1 \vee e_2 \vee e_3) \end{aligned}$$

Jetzt können wir den Allquantor über $m$ durch eine Konjunktion ersetzen, da auch $k$ endlich ist.

$$= \bigwedge_{m=1}^{k} \forall \hat{q}_1, \dots, \hat{q}_m \in Q : \left( \bigwedge_{i=1}^{m} \beta_{i-1} \vDash \delta(\hat{q}_i, \hat{q}_{i-1}) \right) \rightarrow (e_1 \vee e_2 \vee e_3)$$

Abschließend ersetzen wir die $q_j$, $\hat{q}_j$, $e_1$ und $e_2$ durch ihre symbolischen Entsprechungen.

$$\tilde{e}_4 = \bigwedge_{m=1}^{k} \forall \hat{\mathbf{q}}_1, \dots, \hat{\mathbf{q}}_m : \left( \bigwedge_{i=1}^{m} f_\delta(\hat{\mathbf{q}}_i, \hat{\mathbf{q}}_{i-1}, \mathbf{c}_{i-1}) \right) \rightarrow (\tilde{e}_1 \vee \tilde{e}_2 \vee \tilde{e}_3) \quad (3.11)$$

Wir erhalten die aussagenlogische Formel $\tilde{e}_4$, in der die $\mathbf{q}_j$ sowie $\hat{\mathbf{q}}_0$ frei vorkommen.

Abschließend können wir den Ausdruck $e_5$ umwandeln, der $q_0$ und $\hat{q}_0$ frei enthält.

$$e_5 = \exists \left( q_0 \xleftarrow{\beta_0} q_1 \xleftarrow{\beta_1} \cdots \xleftarrow{\beta_{k-1}} q_k \right) : e_4$$

Auch hier beginnen wir damit, die Transitionsfolge $\left( q_0 \leftarrow^{\beta_0} \cdots \leftarrow^{\beta_{k-1}} q_k \right)$ durch eine Konjunktion zu ersetzen.

$$= \exists q_1, q_2, \dots, q_k \in Q : \exists \beta_0, \beta_1, \dots, \beta_k \in \mathfrak{B}(V_\Sigma) : \left( \bigwedge_{i=1}^{k} \beta_{i-1} \vDash \delta(q_i, q_{i-1}) \right) \wedge e_4$$

Nun ersetzen wir auch in $e_4$ die Vorkommen der expliziten Variablen und Funktionen durch ihre symbolischen Entsprechungen und erhalten:

$$\tilde{e}_5 = \exists \mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_k : \exists \mathbf{c}_0, \mathbf{c}_1, \dots, \mathbf{c}_k : \left( \bigwedge_{i=1}^{k} f_\delta(\mathbf{q}_i, \mathbf{q}_{i-1}, \mathbf{c}_{i-1}) \right) \wedge \tilde{e}_5 \quad (3.12)$$

Nach der Umwandlung erhalten wir die aussagenlogische Formel $\tilde{e}_5$, in der $\mathbf{q}_0$ und $\hat{\mathbf{q}}_\mathbf{0}$ ungebunden vorkommen.

Kombinieren wir die Ergebnisse aus (3.3), (3.5), (3.10), (3.11) und (3.12) erhalten wir für den vollsymbolischen rückgerichteten Vorgänger-Operator $\mathrm{CPre}_f^{\mathrm{bw}}$ folgende Definition.

**Definition 3.14** Vollsymbolischer rückgerichteter Vorgänger-Operator
Sei $\mathcal{A} = (V_\Sigma, V_Q^0, V_Q^1, f_\Sigma, f_Q, f_I, f_\delta, f_F)$ ein beliebiger FSNBA. Dann ist der *vollsymbolische rückgerichtete Vorgänger-Operator* $\mathrm{CPre}_f^{\mathrm{bw}} : (\mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}) \to (\mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B})$ definiert als

$$\begin{aligned}\mathrm{CPre}_f^{\mathrm{bw}}(f_X) = \exists \mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_k : \exists \mathbf{c}_0, \mathbf{c}_1, \dots, \mathbf{c}_k : \\ \left( \bigwedge_{i=1}^{k} f_\delta(\mathbf{q}_i, \mathbf{q}_{i-1}, \mathbf{c}_{i-1}) \right) \wedge \\ \bigwedge_{m=1}^{k} \forall \hat{\mathbf{q}}_1, \dots, \hat{\mathbf{q}}_m : \left( \bigwedge_{i=1}^{m} f_\delta(\hat{\mathbf{q}}_i, \hat{\mathbf{q}}_{i-1}, \mathbf{c}_{i-1}) \right) \to \\ \left( \bigvee_{j=0}^{m} f_F(\mathbf{q}_j) \wedge \neg f_F(\hat{\mathbf{q}}_j) \right) \\ \vee \left( \bigvee_{j=0}^{m} f_I(\mathbf{q}_j) \wedge \neg f_I(\hat{\mathbf{q}}_j) \right) \\ \vee \ f_X(\mathbf{q}_m, \hat{\mathbf{q}}_m)\end{aligned}$$

Für $f_X = \text{false}$ ist $\mathrm{CPre}_f^x(f_X)$ die boolesche Funktion, die die Menge der Konfigurationen des $x$-Simulationsspiels kodiert, aus denen Spoiler in maximal einer Runde gewinnt. Somit kodiert $\mathrm{CPre}_f^x(f_X)$ genau die Menge $\mathrm{CPre}^x(X)$, wenn $f_X$ die Menge $X$ beschreibt. Daher ergibt sich für $x \in \{\mathrm{di}, \mathrm{bw}\}$ der Zusammenhang

$$f_W^x = \mu(f_W \mapsto \mathrm{CPre}_f^x(f_W)).$$

Wobei $f_W^x$ genau die Menge $W^x$ der Konfigurationen kodiert, aus denen Spoiler eine Gewinnstrategie in dem direkten bzw. rückgerichteten $k$-Lookahead-Simulationsspiel hat. Die vollsymbolische direkte bzw. rückgerichtete $k$-Lookahead-Simulation sind daher definiert durch

$$f_{\sqsubseteq}^{k\text{-di}} = \neg f_W^{\mathrm{di}} \quad \text{und} \quad f_{\sqsubseteq}^{k\text{-bw}} = \neg f_W^{\mathrm{bw}}.$$

### 3.5.2 Verzögerte und faire $k$-Lookahead-Simulation

Auch für den vollsymbolischen Vorgänger-Operator $\mathrm{CPre}_f$ betrachten wir noch einmal den Vorgänger-Operator auf SNBAs $\mathrm{CPre}_\varphi$ aus Definition 3.6.

$$
\begin{aligned}
\mathrm{CPre}_\varphi(X,Y,Z) = \Big\{ (q_0,\hat{q}_0) \;\Big|\; & \exists \left( q_0 \xrightarrow{\beta_0} q_1 \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_{k-1}} q_k \right) : \\
& \forall 0 < m \leq k : \\
& \forall \left( \hat{q}_0 \xrightarrow{\beta_0} \hat{q}_1 \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_{m-1}} \hat{q}_m \right) : \\
& \underbrace{\left( \begin{array}{l} \exists 0 \leq j \leq m : q_j \in F \\ \wedge \forall 0 \leq j \leq m : \hat{q}_j \notin F \\ \wedge (q_m,\hat{q}_m) \in X \end{array} \right)}_{e_1} \\
& \vee \underbrace{\left( \begin{array}{l} \forall 0 \leq j \leq m : \hat{q}_j \notin F \\ \wedge (q_m,\hat{q}_m) \in Y \end{array} \right)}_{e_2} \\
& \vee \underbrace{(q_m,\hat{q}_m) \in Z}_{e_3} \Big\} \\
& \underbrace{\qquad\qquad\qquad\qquad}_{e_4} \\
\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{e_5} &
\end{aligned}
$$

Wir werden den $\mathrm{CPre}_\varphi$-Operator schrittweise umformen und beginnen mit dem Ausdruck $e_1$, in dem die $q_j$ und $\hat{q}_j$ frei vorkommen.

$$
\begin{aligned}
e_1 = \quad & (\exists 0 \leq j \leq m : q_j \in F) \\
& \wedge (\forall 0 \leq j \leq m : \hat{q}_j \notin F) \\
& \wedge (q_m,\hat{q}_m) \in X
\end{aligned}
$$

Da $m$ endlich ist, ersetzen wir zuerst die Quantoren über $j$ durch eine Disjunktion bzw. Konjunktion über $j$.

$$= \left( \bigvee_{j=0}^{m} q_j \in F \right) \wedge \left( \bigwedge_{j=0}^{m} \hat{q}_j \notin F \right) \wedge (q_m, \hat{q}_m) \in X$$

Abschließend ersetzen wir alle Vorkommen von freien Variablen und die charakteristischen Funktionen von $F$ und $X$ durch ihre symbolischen Entsprechungen.

$$\tilde{e}_1 = \left( \bigvee_{j=0}^{m} f_F(\mathbf{q}_j) \right) \wedge \left( \bigwedge_{j=0}^{m} \neg f_F(\hat{\mathbf{q}}_j) \right) \wedge f_X(\mathbf{q}_m, \hat{\mathbf{q}}_m) \tag{3.13}$$

Wir erhalten die aussagenlogische Formel $\tilde{e}_1$, in der die $\mathbf{q}_j$ und $\hat{\mathbf{q}}_j$ frei vorkommen.

Als nächstes wandeln wir den Ausdruck $e_2$ um, in dem $q_m$ und die $\hat{q}_j$ frei vorkommen.

$$e_2 = (\forall 0 \leq j \leq m : \hat{q}_j \notin F)) \wedge (q_m, \hat{q}_m) \in Y$$

Da $m$ endlich ist, ersetzen wir zuerst den Quantor über $j$ durch eine Konjunktion über $j$.

$$= \left( \bigwedge_{j=0}^{m} \hat{q}_j \notin F) \right) \wedge (q_m, \hat{q}_m) \in Y$$

Abschließend ersetzen wir alle Vorkommen von freien Variablen und die charakteristischen Funktionen von $F$ und $Y$ durch ihre symbolischen Entsprechungen.

$$\tilde{e}_2 = \left( \bigwedge_{j=0}^{m} \neg f_F(\hat{\mathbf{q}}_j) \right) \wedge f_Y(\mathbf{q}_m, \hat{\mathbf{q}}_m) \tag{3.14}$$

Wir erhalten die aussagenlogische Formel $\tilde{e}_2$, in der $\mathbf{q}_m$ und die $\hat{\mathbf{q}}_j$ frei vorkommen.

Und auch den Ausdruck $e_3$ wandeln wir wie bereits bekannt um. In $e_3$ kommen $q_m$ und $\hat{q}_m$ frei vor.

$$e_3 = (q_m, \hat{q}_m) \in Z$$

In diesem Ausdruck müssen wir nur die freien Variablen $q_m$ und $\hat{q}_m$ sowie die charakteristische Funktion von $Z$ durch ihre symbolischen Entsprechungen ersetzen und erhalten:

$$\tilde{e}_3 = f_Z(\mathbf{q}_m, \hat{\mathbf{q}}_m) \tag{3.15}$$

Nach der Umwandlung erhalten wir die aussagenlogische Formel $\tilde{e}_3$, in der $\mathbf{q}_m$ und $\hat{\mathbf{q}}_m$ frei vorkommen.

Für die Umformung des Ausdrucks $e_4$ orientieren wir uns an der Umformung des Ausdrucks $e_3$ aus (3.6). In $e_4$ kommen die $q_j$ und $\hat{q}_0$ frei vor.

$$e_4 = \forall 0 < m \leq k : \forall \left( \hat{q}_0 \xrightarrow{\beta_0} \hat{q}_1 \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_{m-1}} \hat{q}_m \right) : (e_1 \vee e_2 \vee e_3)$$

Wenn wir die Umformungen nachvollziehen, erhalten wir:

$$= \bigwedge_{m=1}^{k} \forall \hat{q}_1, \ldots, \hat{q}_m \in Q : \left( \bigwedge_{i=1}^{m} \beta_{i-1} \vDash \delta(\hat{q}_{i-1}, \hat{q}_i) \right) \rightarrow (e_1 \vee e_2 \vee e_3)$$

Abschließend ersetzen wir alle Vorkommen von freien Variablen und die $e_1$, $e_2$ und $e_3$ durch ihre symbolischen Entsprechungen.

$$\tilde{e}_4 = \bigwedge_{m=1}^{k} \forall \hat{\mathbf{q}}_1, \ldots, \hat{\mathbf{q}}_m : \left( \bigwedge_{i=1}^{m} f_\delta(\hat{\mathbf{q}}_{i-1}, \hat{\mathbf{q}}_i, \mathbf{c}_{i-1}) \right) \rightarrow (\tilde{e}_1 \vee \tilde{e}_2 \vee \tilde{e}_3) \tag{3.16}$$

Nach der Umwandlung erhalten wir die aussagenlogische Formel $\tilde{e}_4$, in der die $\mathbf{q}_j$ und $\hat{\mathbf{q}}_0$ frei vorkommen.

Auch den Ausdruck $e_5$ formen wir um, wie bereits aus der Umformung des Ausdrucks $e_4$ aus (3.8) bekannt.

$$e_5 = \exists \left( q_0 \xrightarrow{\beta_0} q_1 \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_{k-1}} q_k \right) : e_4$$

Wenn wir die Umformungen nachvollziehen, erhalten wir:

$$\begin{aligned} = \; & \exists q_1, q_2, \ldots, q_k \in Q : \exists \beta_0, \beta_1, \ldots, \beta_k \in \mathfrak{B}(V_\Sigma) : \\ & \left( \bigwedge_{i=1}^{k} \beta_{i-1} \vDash \delta(q_{i-1}, q_i) \right) \wedge e_4 \end{aligned}$$

Abschließend ersetzen wir alle Vorkommen von freien Variablen und $e_4$ durch ihre symbolischen Entsprechungen.

$$\tilde{e}_5 = \exists \mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_k : \exists \mathbf{c}_0, \mathbf{c}_1, \dots, \mathbf{c}_k : \left( \bigwedge_{i=1}^{k} f_\delta(\mathbf{q}_{i-1}, \mathbf{q}_i, \mathbf{c}_{i-1}) \right) \wedge \tilde{e}_4 \tag{3.17}$$

Wir erhalten die aussagenlogische Formel $\tilde{e}_5$, in der $\mathbf{q}_0$ und $\hat{\mathbf{q}}_0$ frei vorkommen.

Wir kombinieren die Ergebnisse aus (3.13), (3.14), (3.15), (3.16) und (3.17) zu dem vollsymbolischen Vorgänger-Operator $\mathrm{CPre}_f$ für beliebige zweistellige boolesche Funktionen $f_X, f_Y, f_Z : \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}$.

**Definition 3.15** Vollsymbolischer Vorgänger-Operator
Sei $\mathcal{A} = (V_\Sigma, V_Q^0, V_Q^1, f_\Sigma, f_Q, f_I, f_\delta, f_F)$ ein beliebiger FSNBA, dann ist der *vollsymbolische Vorgänger-Operator* $\mathrm{CPre}_f : (\mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B})^3 \to (\mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B})$ definiert als

$$\begin{aligned}
\mathrm{CPre}_f(f_X, f_Y, f_Z) = \exists \mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_k : \exists \mathbf{c}_0, \mathbf{c}_1, \dots, \mathbf{c}_k : \\
&\left( \bigwedge_{i=1}^{k} f_\delta(\mathbf{q}_{i-1}, \mathbf{q}_i, \mathbf{c}_{i-1}) \right) \wedge \\
&\bigwedge_{m=1}^{k} \forall \hat{\mathbf{q}}_1, \dots, \hat{\mathbf{q}}_m : \left( \bigwedge_{i=1}^{m} f_\delta(\hat{\mathbf{q}}_{i-1}, \hat{\mathbf{q}}_i, \mathbf{c}_{i-1}) \right) \rightarrow \\
&\quad \left( \left( \bigvee_{j=0}^{m} f_F(\mathbf{q}_j) \right) \wedge \left( \bigwedge_{j=0}^{m} \neg f_F(\hat{\mathbf{q}}_j) \right) \wedge f_X(\mathbf{q}_m, \hat{\mathbf{q}}_m) \right) \\
&\quad \vee \left( \left( \bigwedge_{j=0}^{m} \neg f_F(\hat{\mathbf{q}}_j) \right) \wedge f_Y(\mathbf{q}_m, \hat{\mathbf{q}}_m) \right) \\
&\quad \vee \; f_Z(\mathbf{q}_m, \hat{\mathbf{q}}_m) \Big\}
\end{aligned}$$

Für das faire Simulationsspiel gilt: Spoiler gewinnt, gdw. er unendlich viele akzeptierende Zustände besucht, während Duplikator nur endlich

viele akzeptierende Zustände besucht. Daher ergibt sich für die faire $k$-Lookahead-Simulation folgender Zusammenhang:

$$f_W^{\mathrm{f}} = \mu(f_Z \mapsto \nu(f_X \mapsto \mu(f_Y \mapsto \mathrm{CPre}_f(f_X, f_Y, f_Z))))$$

Wobei $f_W^{\mathrm{f}}$ genau die Menge $W^{\mathrm{f}}$ der Konfigurationen kodiert, aus denen Spoiler eine Gewinnstrategie in dem fairen $k$-Lookahead-Simulationsspiel hat. Die vollsymbolische faire $k$-Lookahead-Simulation ist daher definiert als

$$f_{\sqsubseteq}^{k\text{-f}} = \neg f_W^{\mathrm{f}}.$$

Spoiler gewinnt das verzögerte Simulationsspiel, wenn nach endlich vielen Runden die folgenden beiden Bedingungen erfüllt sind:

1. Spoiler besucht einen akzeptierenden Zustand und
2. Spoiler verhindert, dass Duplikator in der Zukunft einen akzeptierenden Zustand besucht.

Daraus lassen sich die beiden folgenden symbolischen Operatoren ableiten:

$$\mathrm{CPre}_f^1(f_X, f_Y) := \mathrm{CPre}_f(f_X, f_Y, f_Y)$$

und

$$\mathrm{CPre}_f^2(f_X, f_Y) := \mathrm{CPre}_f(f_X, f_X, f_Y)$$

Somit gilt für die verzögerte $k$-Lookahead-Simulation folgender Zusammenhang:

$$f_W^{\mathrm{de}} = \mu(f_W \mapsto \mathrm{CPre}_f^1(\nu(f_X \mapsto \mathrm{CPre}_f^2(f_X, f_W)), f_W))$$

Wobei $f_W^{\mathrm{de}}$ genau die Menge $W^{\mathrm{de}}$ der Konfigurationen kodiert, aus denen Spoiler eine Gewinnstrategie in dem verzögerten $k$-Lookahead-Simulationsspiel hat. Die vollsymbolische verzögerte $k$-Lookahead-Simulation ist daher definiert als

$$f_{\sqsubseteq}^{k\text{-de}} = \neg f_W^{\mathrm{de}}.$$

## 3.6 Minimierung vollsymbolischer NBAs

Im vorherigen Abschnitt haben wir beschrieben, wie wir $k$-Lookahead-Simulationen auf vollsymbolischen NBAs berechnen können. Auf dieser Basis werden wir nun die verbliebenen Bausteine der Minimierung vollsymbolischer NBAs vollständig symbolisch berechnen. Dafür übertragen wir auch diese Komponenten der Minimierung in die quantifizierte Aussagenlogik.

### 3.6.1 Entfernen toter Zustände

Auch in vollsymbolischen NBAs können die toten Zustände entfernt werden, indem die Anzahl der gültigen Belegungen für $V_Q^0$ und $V_Q^1$ verringert wird. Wir werden im Folgenden die notwendigen Schritte für diese Reduktion auf FSNBAs beschreiben.

Wir beginnen damit, die $\delta$-Kanten eines vollsymbolischen Automaten vollsymbolisch zu repräsentieren.

**Definition 3.16** $\delta$-Kanten Funktion
Sei $\mathcal{A} = (V_\Sigma, V_Q^0, V_Q^1, f_\Sigma, f_Q, f_I, f_\delta, f_F)$ ein beliebiger FSNBA, dann ist $f_{\delta_e} : V_Q^0 \times V_Q^1 \to \mathbb{B}$ die boolesche Funktion, die die *$\delta$-Kanten* von $\mathcal{A}$ ohne Beachtung der Kantenbeschriftung beschreibt mit

$$f_{\delta_e}(\mathbf{q}_0, \mathbf{q}_1) = \exists \mathbf{c}_0 : f_\delta(\mathbf{q}_0, \mathbf{q}_1, \mathbf{c}_0)$$

Als nächstes geben wir den Algorithmus 3.1 an. Dieser berechnet für eine gegebene boolesche Funktion $f_R : \mathbf{V}_X^0 \times \mathbf{V}_X^1 \to \mathbb{B}$, die eine Binärrelation $R \subseteq X \times X$ auf der Menge $X$ beschreibt, eine boolesche Funkton $f_R^+ : \mathbf{V}_X^0 \times \mathbf{V}_X^1 \to \mathbb{B}$. Die berechnete Funktion $f_R^+$ beschreibt die transitive Hülle $R^+$ der Binärrelation $R$.

In Schleifendurchlauf $i$ bestimmt der Algorithmus die Funktion $f_R^{\leq 2i}$, die die Relation $R^{\leq 2i}$ beschreibt. Dies wird solange wiederholt, bis $f_R^{\leq 2i} = f_R^{\leq 2(i+1)}$ gilt. Die Funktion, die die transitive Hülle von $R$ beschreibt,

ist dann durch $f_R^+ = f_R^{\leq 2i}$ bestimmt. Der Algorithmus benötigt für die Berechnung der Funktion $f_R^+$ maximal $O(|V_X|)$ viele Schleifendurchläufe, da über der Repräsentationsdomäne $V_X$ maximal $2^{|V_X|}$ viele Elemente von $X$ kodiert werden können und in jedem Schleifendurchlauf die Länge der maximalen Relationsverkettung ohne Kreise verdoppelt wird.

**Algorithm 3.1** TransitiveClosure($f_R$)

```
   input: f_R : V_X^0 × V_X^1 → 𝔹

3: procedure TransitiveClosure(f_R)
       f_R^+ ← f_R
       repeat
6:         f_last ← f_R^+
           f_R^+ ← f_R^+ ∨ ∃z : f_R^+(x, z) ∧ f_R^+(z, y)
       until f_R^+ = f_last
9:     return f_R^+
   end procedure
```

Da wir nun mithilfe von Algorithmus 3.1 die Funktion $f_{\delta_e}^+$ bestimmen können, können wir darauf aufbauend die lebendigen Zustände eines vollsymbolischen Automaten $\mathcal{A} = (V_\Sigma^0, V_Q^0, V_Q^1, f_\Sigma, f_Q, f_I, f_\delta, f_F)$ durch die Funktion $f_{Q_l} : \mathbf{V}_Q \to \mathbb{B}$ beschreiben mit

$$\begin{aligned} f_{Q_l}(\mathbf{q}_i) = \quad & \exists \mathbf{q}_j : (f_F(\mathbf{q}_j) \wedge f_{\delta_e}^+(\mathbf{q}_j, \mathbf{q}_j) \wedge (f_{\delta_e}^+(\mathbf{q}_i, \mathbf{q}_j) \vee \mathbf{q}_i = \mathbf{q}_j)) \\ & \wedge \exists \mathbf{q}_h : (f_I(\mathbf{q}_h) \wedge (f_{\delta_e}^+(\mathbf{q}_h, \mathbf{q}_i) \vee \mathbf{q}_i = \mathbf{q}_h) \end{aligned}$$

Die Funktion beschreibt vollsymbolisch die Menge der lebendigen Zustände aus (3.1).

Nachdem wir die lebendigen Zustände bestimmen können, können wir den lebendigen Automaten als den Automaten ohne die toten Zustände definieren.

**Definition 3.17** Vollsymbolischer lebendiger Automat
Sei $\mathcal{A} = (V_\Sigma^0, V_Q^0, V_Q^1, f_\Sigma, f_Q, f_I, f_\delta, f_F)$ ein beliebiger FSNBA. Der *voll-*

*symbolische lebendige Automat* ist dann definiert als

$$\text{Living}(\mathcal{A}) := (V_\Sigma^0, V_Q^0, V_Q^1, f_\Sigma, f'_Q, f'_I, f'_\delta, f'_F) \text{ mit}$$

$$\begin{aligned} f'_Q &= f_{Q_l} \\ f'_I &= f_I \wedge f'_Q \\ f'_F &= f_F \wedge f'_Q \end{aligned}$$

und $\delta' : \mathbf{V}_Q^0 \times \mathbf{V}_Q^1 \times \mathbf{V}_\Sigma^0 \to \mathbb{B}$ mit

$$f'_\delta(\mathbf{q}_i, \mathbf{q}_j, \mathbf{c}) = f'_Q(\mathbf{q}_i) \wedge f'_Q(\mathbf{q}_j) \wedge f_\delta(\mathbf{q}_i, \mathbf{q}_j, \mathbf{c})$$

Somit können wir nun die Modellmengen der booleschen Funktionen vollsymbolisch repräsentierter NBAs einschränken, sodass nur noch lebendige Zustände im zugrundeliegenden Automaten berücksichtigt werden.

### 3.6.2 Stutzen der Transitionen

Nachdem wir die überflüssigen toten Zustände der FSNBAs entfernen können, wollen wir auch überflüssige Transitionen aus den FSNBAs entfernen können. Daher übertragen wir nun auch die zum Pruning von Transitionen notwendigen Definitionen auf vollsymbolische NBAs.

Wir müssen keine gesonderte Funktion $f_{\delta_\beta} : \mathbf{V}_Q^0 \times \mathbf{V}_Q^1 \times \mathbf{V}_\Sigma^0 \to \mathbb{B}$ mehr definieren, da die Funktion $f_\delta$ bereits die durch $\delta$ definierten $\beta$-Transitionen beschreibt. Somit können wir nun eine Pruning-Relation über den $\beta$-Transitionen eines vollsymbolischen Automaten definieren.

**Definition 3.18** Vollsymbolische Pruning-Relation
Sei $\mathcal{A} = (V_\Sigma^0, V_Q^0, V_Q^1, f_\Sigma, f_Q, f_I, f_\delta, f_F)$ ein beliebiger FSNBA und $f_{R_b}$ und $f_{R_f}$ zwei boolesche Funktionen, die Binärrelationen auf $Q$ beschreiben, dann ist eine *vollsymbolisch Pruning-Relation* definiert als die Funktion

$$f_P(f_{R_b}, f_{R_f}) : \mathbf{V}_Q^0 \times \mathbf{V}_Q^1 \times \mathbf{V}_\Sigma^0 \times \hat{\mathbf{V}}_Q^0 \times \hat{\mathbf{V}}_Q^1 \times \hat{\mathbf{V}}_\Sigma^0 \to \mathbb{B} \text{ mit}$$

$$f_P(f_{R_b}, f_{R_f})(\mathbf{q}_0, \mathbf{q}_1, \mathbf{c}_0, \hat{\mathbf{q}}_0, \hat{\mathbf{q}}_1, \hat{\mathbf{c}}_0) \coloneqq \mathbf{c}_0 = \hat{\mathbf{c}}_0 \wedge f_{R_b}(\mathbf{q}_i, \hat{\mathbf{q}}_i) \wedge f_{R_f}(\mathbf{q}_j, \hat{\mathbf{q}}_j).$$

Ist $f_P(f_{R_b}, f_{R_f})(\mathbf{q}_0, \mathbf{q}_1, \mathbf{c}_0, \hat{\mathbf{q}}_0, \hat{\mathbf{q}}_1, \hat{\mathbf{c}}_0)$ für Belegungen $\mathbf{q}_0 \in \mathbf{V}_Q^0$, $\mathbf{q}_1 \in \mathbf{V}_Q^1$, $\mathbf{c}_0 \in \mathbf{V}_\Sigma^0$, $\hat{\mathbf{q}}_0 \in \hat{\mathbf{V}}_Q^0$, $\hat{\mathbf{q}}_1 \in \hat{\mathbf{V}}_Q^1$ und $\hat{\mathbf{c}}_0 \in \hat{\mathbf{V}}_\Sigma^0$ erfüllt, so können wir $f_\delta$ anpassen, sodass $\mathbf{q}_0$, $\mathbf{q}_1$ und $\mathbf{c}_0$ kein Modell mehr für $f_\delta$ beschreiben.

**Definition 3.19** Vollsymbolisch gestutzter Automat
Sei $\mathcal{A} = (V_\Sigma^0, V_Q^0, V_Q^1, f_\Sigma, f_Q, f_I, f_\delta, f_F)$ ein beliebiger FSNBA und $f_P$ eine boolesche Funktion, die eine transitive und asymmetrische Pruning-Relation auf den $\beta$-Transitionen von $\delta$ beschreibt. Der *vollsymbolische gestutzte Automat* ist dann definiert als

$$\text{Prune}(\mathcal{A}, f_P) \coloneqq (V_\Sigma^0, V_Q^0, V_Q^1, f_\Sigma, f_Q, f_I, f'_\delta, f_F) \text{ mit}$$

$$f'_\delta(\mathbf{q}_i, \mathbf{q}_j, \mathbf{c}_i) = f_\delta(\mathbf{q}_i, \mathbf{q}_j, \mathbf{c}_i) \wedge \nexists \hat{\mathbf{q}}_i, \hat{\mathbf{q}}_j, \hat{\mathbf{c}}_i : f_P(\mathbf{q}_i, \mathbf{q}_j, \mathbf{c}_i, \hat{\mathbf{q}}_i, \hat{\mathbf{q}}_j, \hat{\mathbf{c}}_i).$$

### 3.6.3 Bilden des Quotienten

Nachdem wir bereits vollsymbolisch tote Zustände entfernen und Transitionen stutzen können, werden wir in diesem Abschnitt abschließend beschreiben, wie der Quotient eines vollsymbolischen NBAs berechnet werden kann.

Da der Quotientenautomat aus Definition 3.11 die Quotientenmenge $[Q]$ als Zustandsmenge des SNBAs verwendet, kann dieser nicht direkt vollständig symbolisch beschrieben werden. Dies ist dadurch begründet, dass über der Repräsentationsdomäne $V_Q$ nur Zustände $q \in Q$ kodiert werden können und nicht Teilmengen von $Q$. Stattdessen werden wir einen Automaten definieren, der ein eindeutiges Repräsentantensystem der bisherigen Zustandsmenge als neue Zustandsmenge verwendet.

**Definition 3.20** Vollständiges Repräsentantensystem
Sei $M$ eine beliebige Menge, $[M]$ ihre Quotientenmenge und $V \subseteq M$

eine beliebige Teilmenge von $M$. Falls für jede Äquivalenzklasse $[m] \in [M]$ genau ein $v \in V$ mit $v \in [m]$ existiert, dann nennt man $V$ ein *vollständiges Repräsentantensystem* für die durch die Äquivalenzklassen beschriebene Äquivalenzrelation $\equiv$.

Die Existenz eines vollständigen Repräsentantensystems können wir uns zunutze machen und die eindeutige Repräsentantenmenge als Basis für weitere Definitionen beschreiben.

**Definition 3.21** Eindeutige Repräsentantenmenge
Sei $M$ eine beliebige Menge und $\leq$ eine beliebige totale Ordnung auf $M$. Dann ist die *Menge der eindeutigen Repräsentanten* $[M]!$ von $M$ definiert als

$$[M]! := \{m \in M \mid \forall x \in M : m \equiv x \rightarrow m \leq x\}$$

Sei $\sqsubseteq \subseteq Q \times Q$ eine beliebige Quasiordnung auf den Zuständen eines Automaten und $f_{\sqsubseteq} : \mathbf{V}_Q^0 \times \mathbf{V}_Q^1 \rightarrow \mathbb{B}$ eine boolesche Funktion, die sie vollsymbolisch beschreibt. Dann kann die vollsymbolische Funktion für die Äquivalenzrelation $f_{\equiv} : \mathbf{V}_Q^0 \times \mathbf{V}_Q^1 \rightarrow \mathbb{B}$ wie folgt berechnet werden:

$$f_{\equiv}(\mathbf{q}_0, \mathbf{q}_1) = f_{\sqsubseteq}(\mathbf{q}_0, \mathbf{q}_1) \wedge f_{\sqsubseteq}(\mathbf{q}_1, \mathbf{q}_0)$$

Die Funktion $f_{\leq} : \mathbb{B}^+ \times \mathbb{B}^+ \rightarrow \mathbb{B}$, beschreibt eine totale Ordnung auf den Belegungen einer Domäne. Die Funktion ist für Tupel der Länge eins rekursiv wie folgt definiert:

$$f_{\leq}((x_0), (y_0)) = x_0 \rightarrow y_0$$

Für Tupel der Länge $m > 1$ ist die Funktion entsprechend definiert:

$$\begin{aligned} f_{\leq}((x_0, x_1, \ldots, x_{m-1}), & \\ (y_0, y_1, \ldots, y_{m-1})) &= (\neg x_{m-1} \wedge y_{m-1}) \vee (x_{m-1} \leftrightarrow y_{m-1} \wedge \\ & \qquad f_{\leq}((x_0, x_1, \ldots, x_{m-2}), (y_0, y_1, \ldots, y_{m-2}))) \end{aligned}$$

Somit kann die Funktion $f_{[M]!} : \mathbf{V}_Q^0 \to \mathbb{B}$, die die eindeutige Repräsentantenmenge $[M]!$ einer durch $f_M : \mathbf{V}_Q^0 \to \mathbb{B}$ beschriebenen Menge $M$ beschreibt, wie folgt berechnet werden:

$$f_{[M]!}(\mathbf{q}_0) = f_M(\mathbf{q}_0) \wedge \forall \mathbf{q}_1 : f_{\equiv}(\mathbf{q}_0, \mathbf{q}_1) \to f_{\leq}(\mathbf{q}_0, \mathbf{q}_1)$$

**Definition 3.22** Vollsymbolischer Repräsentantenautomat
Sei $\mathcal{A} = (V_\Sigma^0, V_Q^0, V_Q^1, f_\Sigma, f_Q, f_I, f_\delta, f_F)$ ein beliebiger FSNBA und $f_\sqsubseteq : \mathbf{V}_Q^0 \times \mathbf{V}_Q^1 \to \mathbb{B}$ eine boolesche Funktion, die eine beliebige Quasiordnung auf den zugrunde liegenden Zuständen des Automaten beschreibt. Dann ist der *vollsymbolische Repräsentantenautomat* bezüglich $f_\sqsubseteq$ definiert als $A/f_\sqsubseteq = (V_\Sigma^0, V_Q^0, V_Q^1, f_\Sigma, f'_Q, f'_I, f'_\delta, f'_F)$ mit

$$\begin{aligned}
f'_Q &= f_{[Q]!} \\
f'_I(\mathbf{q}_0) &= f'_Q(\mathbf{q}_0) \wedge \exists \mathbf{q}_j : f_I(\mathbf{q}_j) \wedge f_\equiv(\mathbf{q}_i, \mathbf{q}_j) \\
f'_F(\mathbf{q}_0) &= f'_Q(\mathbf{q}_0) \wedge \exists \mathbf{q}_j : f_F(\mathbf{q}_j) \wedge f_\equiv(\mathbf{q}_i, \mathbf{q}_j) \\
f'_\delta(\mathbf{q}_i, \mathbf{q}_j, \mathbf{c}_i) &= f'_Q(\mathbf{q}_i) \wedge f'_Q(\mathbf{q}_j) \wedge f_\Sigma(\mathbf{c}_i) \wedge \\
&\quad \exists \hat{\mathbf{q}}_i, \hat{\mathbf{q}}_j : f_\equiv(\mathbf{q}_i, \hat{\mathbf{q}}_i) \wedge f_\equiv(\mathbf{q}_j, \hat{\mathbf{q}}_j) \wedge f_\delta(\hat{\mathbf{q}}_i, \hat{\mathbf{q}}_j, \mathbf{c}_i)
\end{aligned}$$

Der so definierte vollsymbolische Repräsentantenautomat erfüllt dieselbe Rolle wie der Quotientenautomat, verwendet jedoch statt der Äquivalenzklassen von $Q$ eindeutige Repräsentanten der Klassen als Zustände. Daher kann der Repräsentantenautomat vollsymbolisch repräsentiert werden.

### 3.6.4 Minimierungsverfahren auf FSNBAs

Auch die vollsymbolischen Minimierungsverfahren bauen auf den Verfahren aus Unterabschnitt 2.5.4 auf. Statt der symbolischen $k$-Lookahead-Simulationen werden jedoch die oben beschriebenen vollsymbolischen Simulationen verwendet:

- vollsymbolische rückgerichtete $k$-Lookahead-Simulation $f_{\sqsubseteq}^{k\text{-bw}}$ statt symbolischer rückgerichteter $k$-Lookahead-Simulation $\sqsubseteq^{k\text{-bw}}$
- vollsymbolische direkte $k$-Lookahead-Simulation $f_{\sqsubseteq}^{k\text{-di}}$ statt symbolischer direkter $k$-Lookahead-Simulation $\sqsubseteq^{k\text{-di}}$
- vollsymbolische verzögerte $k$-Lookahead-Simulation $f_{\sqsubseteq}^{k\text{-de}}$ statt symbolischer verzögerter $k$-Lookahead-Simulation $\sqsubseteq^{k\text{-de}}$
- vollsymbolische faire $k$-Lookahead-Simulation $f_{\sqsubseteq}^{k\text{-f}}$ statt symbolischer fairer $k$-Lookahead-Simulation $\sqsubseteq^{k\text{-f}}$

Auf Basis der vollsymbolischen Simulationen erfolgt die Entfernung der lebendigen Zustände und das Stutzen der Transitionen auch vollsymbolisch. Im Unterschied zur Minimierung expliziter NBAs wird jedoch nicht der Quotientenautomat berechnet, da dieser nicht durch dieselben Strukturen wie der Ausgangsautomat beschrieben werden kann. Stattdessen wird der Repräsentantenautomat des Ausgangsautomaten bzgl. der verzögerten bzw. rückgerichteten vollsymbolischen $k$-Lookahead-Simulation vollständig symbolisch berechnet.

Ansonsten erfolgt die Minimierung vollsymbolischer NBAs analog zur Minimierung expliziter und symbolischer NBAs. Daher gelten auch für die vollsymbolische Minimierung die Hinweise zur Korrektheit aus Unterabschnitt 2.5.4.

# 4 Realisierung des vollsymbolischen Minimierungsframeworks

In diesem Kapitel werden wir die Realisierung des in Abschnitt 3.3 und Abschnitt 3.6 entworfenen Minimierungsframeworks beschreiben.

Dafür wird zuerst in Abschnitt 4.1 die Automaten- und Logikbibliothek RltlConv beschrieben, in deren Kontext das vollsymbolische Minimierungsframework implementiert wurde. Danach werden wir in Abschnitt 4.2 die verschiedenen Alternativen für die Umsetzung vorstellen und bewerten. Die Grundlagen für die gewählte Realisierungsform werden dann in Abschnitt 4.3 beschrieben. Abschließend beschreiben wir in Abschnitt 4.4 die einzelnen Aspekte der Implementierung.

## 4.1 Die Automaten- und Logikbibliothek RltlConv

In diesem Abschnitt beschreiben wir kurz die Automaten- und Logikbibliothek RltlConv, die die Umwandlung von verschiedenen Automatenmodellen in verschiedene andere Modelle erlaubt. Zu den unterstützten Automatenmodellen gehören:

- Automaten auf endlichen Worten
  - Nichtdeterministische endliche Automaten (NFA)
  - Moore-Automaten
  - Mealy-Automaten
- Zwei-Wege-Automaten auf endlichen Worten

    * Nichtdeterministische endliche Zwei-Wege-Automaten (2NFA)
    * Alternierende Zwei-Wege-Mealy-Automaten

- Automaten auf unendlichen Worten

    * Nichtdeterministische Büchi-Automaten (NBA)
    * Alternierende Büchi-Automaten (ABA)
    * Alternierende Paritätsautomaten (APA)

- Zwei-Wege-Automaten auf unendlichen Worten

    * Nichtdeterministische Zwei-Wege-Büchi-Automaten (2NBA)
    * Alternierende Zwei-Wege-Büchi-Automaten (2ABA)
    * Alternierende Zwei-Wege-Paritätsautomaten (2APA)

Weiterhin unterstützt RltlConv verschiedene Logiken:

- Reguläre Ausdrücke mit Vergangenheit
- $\omega$-reguläre Ausdrücke mit Vergangenheit
- Reguläre Linearzeit-Temporallogik (RLTL)
- Linearzeit-Temporallogik mit Vergangenheit (LTL)

Zudem enthält die Bibliothek verschiedene Umwandlungen, die die Logiken in andere Logiken überführen oder in eines der unterstützten Automatenmodelle übersetzen. So können beispielsweise $\omega$-reguläre Ausdrücke in RLTL überführt werden. Die RLTL-Formeln können wiederum in alternierende Paritätsautomaten übersetzt werden.

Der Grundstein für die RltlConv-Bibliothek wurde im Rahmen von [Sch12] gelegt und seitdem in verschiedenen Arbeiten erweitert.

### 4.1.1 Beschreibung der Scala-Plattform

Die Bibliothek wurde überwiegend in der Programmiersprache Scala[1] entwickelt. Scala ist eine Mehrparadigmensprachen, das heißt, sie unterstützt imperative und funktionale Programmierstile. Außerdem sind alle Werte in Scala Objekte, auch primitive Datentypen und Funktionen, daher ist Scala rein objektorientiert. Scala-Code wird auf der Java Virtual Machine[2] (JVM) ausgeführt und ist vollständig kompatibel zu Java-Code, daher können sowohl Java-Bibliotheken in Scala als auch Scala-Bibliotheken in Java verwendet werden.

Für die Implementierung der verschiedenen Umwandlungen existieren in RltlConv bereits Klassen- und Objektstrukturen, die für die Repräsentation der verschiedenen Automaten geeignet sind. Dies begründet, dass auch das vollsymbolische Minimierungsframework in Scala implementiert wurde.

Die vorhandenen Klassen zur Modellierung der verschiedenen Automaten werden im Folgenden vorgestellt. Wir werden die grafische Modellierungssprache Unified Modeling Language[3] (UML) zur Beschreibung der Scala-Konstrukte verwenden und orientieren uns dabei an der UML-Notation für Scala aus [Rac09, Anhang C], die UML um Mittel zur Beschreibung von Scala-Sprachelementen erweitert, die standardmäßig nur unzureichend beschrieben werden können. Die Notation lässt sich zusammenfassen zu:

- Alle nicht privaten Attribute definieren implizit Getter-Methoden.
- Generische Klassen werden mit abstrakten Typen annotiert. Die Annotation verwendet die in Scala übliche Notation.
- Traits werden durch abstrakte Klassen dargestellt, die zusätzlich mit <<trait>> annotiert sind.

[1] www.scala-lang.org
[2] docs.oracle.com/javase/specs/
[3] www.uml.org

– Scalas Singleton-Objekte werden durch Klassen mit dem <<singleton>>-Stereotypen repräsentiert.

Zusätzlich bietet Scala noch das Konzept von Case-Klassen und -Objekten. Das sind Klassen und Objekte, deren Konstruktorparameter öffentlich abrufbar sind und für die automatisch `equals`-, `hashcode`- und `unapply`-Methoden auf Basis ihrer Konstruktorparameter erzeugt werden. Dadurch können sie für das sehr mächtige funktionale Sprachfeature Pattern Matching verwendet werden. Pattern Matching erweitert die aus Java bekannte `switch`-Anweisung und erlaubt so auch die Verwendung von Objektstrukturen als Bedingungen. Die Case-Klassen übernehmen somit in Scala die Funktion von algebraischen Datentypen, wie sie aus anderen funktionalen Programmiersprachen bekannt sind [Hud+07]. Wir werden Case-Klassen und -Objekte durch den UML-Stereotypen <<datatype>> kenntlich machen.

### 4.1.2 Beschreibung der vorhandenen Bibliothek

RltlConv enthält bereits unterschiedliche Klassen, die der Repräsentation von Automaten, Zuständen und Eingabesymbolen dienen. Abbildung 4.1 zeigt eine Auswahl der grundlegenden Klassen. Die Klasse `Automata` dient allen in RltlConv enthaltenen Automatenmodellen als abstrakte Basisklasse. Da die verschiedensten Modelle enthalten sind, kann Automata keine allgemeingültigen Attribute oder Methoden für alle Automaten-Implementierungen vorschreiben und dient daher nur als sehr allgemeiner abstrakter Ausgangspunkt für die entsprechenden Definitionen. Die Klassen `Automaton` und `AutomatonSpecular` sind abstrakte Unterklassen von `Automata` und dienen der Beschreibung eines Automaten oder eines Automatenpaares. Ein solches Automatenpaar beinhaltet dann zwei sich gegenseitig ergänzende Automaten zur Beschreibung bestimmter Eigenschaften.

Die Klassen `State` und `DirectedState` sind zwei Case-Klassen, die der Repräsentation von Zuständen der Automaten dienen. Sie werden von den später beschriebenen konkreten Automatenimplementierungen verwendet. Die Klasse `State` repräsentiert einen einfachen

Zustand, der nur durch einen Namen beschrieben wird. Dies reicht aus um NBAs oder NFAs zu modellieren. Für die Modellierung von 2NFAs oder 2NBA wird jedoch die Klasse `DirectedState` verwendet, da sie den Zustand noch um eine Richtungsangabe für die Bewegung auf der Eingabe erweitert. Diese Richtungsangaben werden durch die abstrakte `Direction`-Klasse repräsentiert. Es existieren drei Case-Objekte als Unterobjekte von `Direction`, die die drei Bewegungsmöglichkeiten auf der Eingabe des Automaten kodieren.

Für die Kodierung von Transitionsbedingungen in den expliziten Automatenmodellen werden die Unterklassen von `Sign` verwendet. Die Klasse `Epsilon` kodiert $\varepsilon$-Transitionen in den Automaten. Durch `All` werden Transitionen modelliert, die für alle Eingabesymbole aktiviert werden können. Um einzelne Eingabesymbole zu kodieren, wird die Klasse `Text` verwendet, die das Eingabesymbol als `String`-Attribut speichert.

Abbildung 4.2 zeigt die konkreten Klassen, die zur Modellierung von expliziten 2NBAs, 2NFAs und 2APAs in RltlConv verwendet werden. Diese Automatenmodelle werden in der Bibliothek sehr nahe an den mathematischen Definitionen der Automaten repräsentiert. So werden die Transitionsfunktionen von `Nba` und `Nfa` durch eine `Map` realisiert, die von einem Tupel aus einem Zustand und einem Zeichen auf eine Liste von Zuständen und einer Richtung abbildet. Der 2APA verwendet stattdessen eine `Map`, die auf eine positive boolesche Kombination von Zuständen und Richtungen abbildet.

## 4.2 Verschiedene alternative Ansätze für die Realisierung des Minimierungsframeworks

In Abschnitt 3.3 und Abschnitt 3.6 wurde beschrieben, wie das in Abschnitt 2.5 vorgestellte Minimierungsframework auf symbolische nichtdeterministischen Büchi-Automaten übertragen werden kann. In diesem Abschnitt wollen wir die verschiedenen alternativen Ansätze für die Realisierung dieses Minimierungsframeworks betrachten und bewerten.

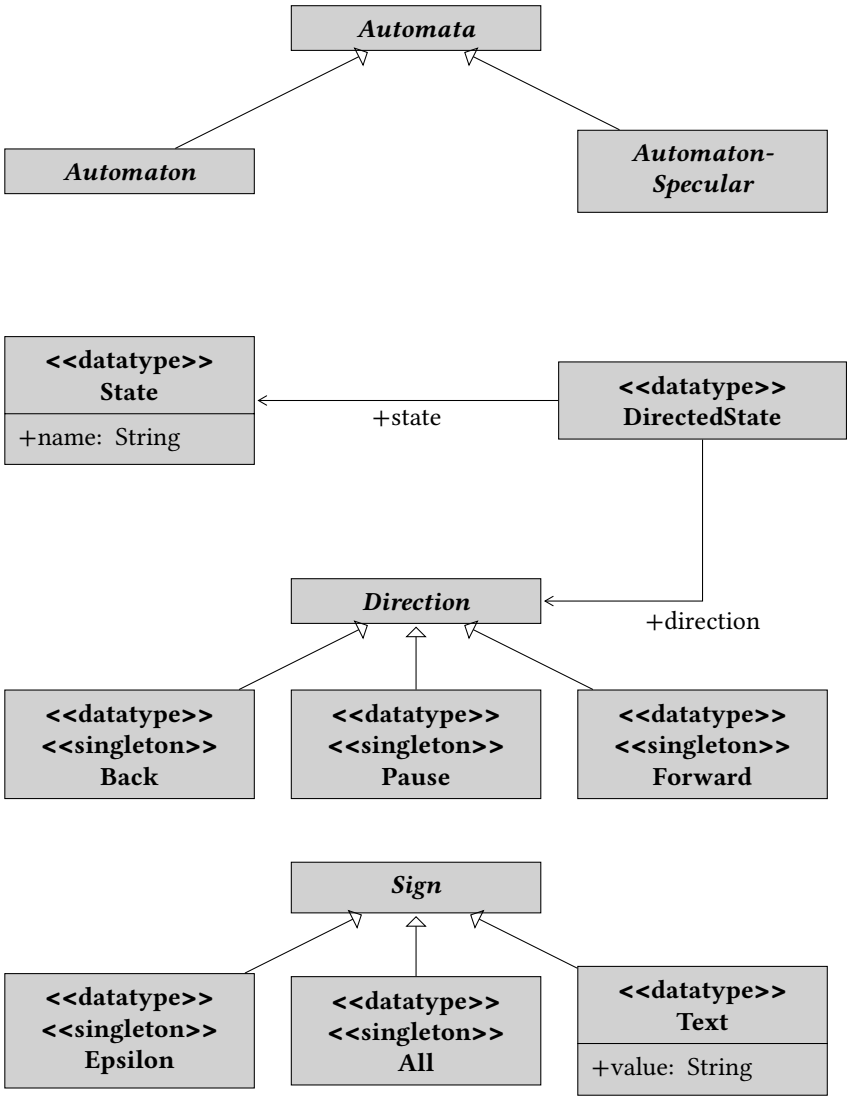



Abbildung 4.1: Klassendiagramm der Modellklassen für Automaten in RltlConv. Diese Klassen und Objekte bilden die Grundlage für die Repräsentation aller Automatenmodelle in RltlConv.

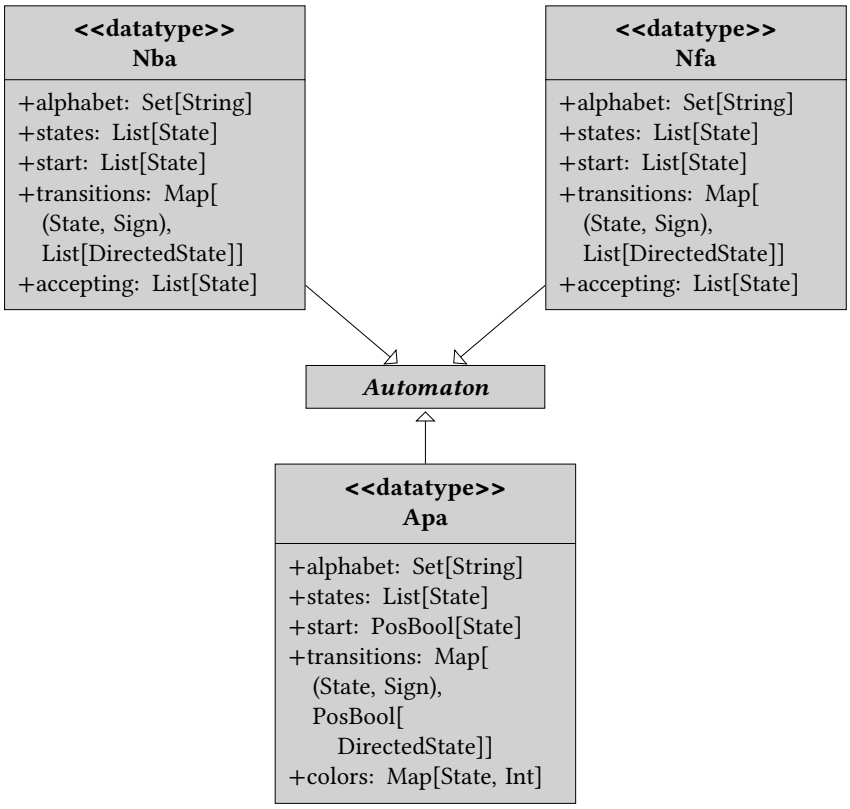



Abbildung 4.2: Das Klassendiagramm zeigt die Modellklassen der expliziten 2NBA, 2NFA und APA in RltlConv.

### 4.2.1 Realisierung durch explizite Modellierung

Die erste Realisierungsmöglichkeit besteht darin, die in Abschnitt 3.3 verwendeten Methoden und Verfahren explizit zu modellieren. Dies bedeutet insbesondere die Zustände und die Transitionen des Automaten durch entsprechende Klassen- und Objekthierarchien zu beschreiben. Darüber hinaus müssen für die Berechnung der Vorgänger-Operatoren die Pfade der Länge $k$, die Spoiler wählen kann, und die Pfade der Länge $m \leq k$, die Duplikator wählen kann, explizit beschrieben werden.

Dies bedeutet eine enorme Menge an Objektstrukturen, die in jeder Iteration der Minimierung erneut aufgebaut werden muss und sich für jede Ausprägung der $k$-Lookahead-Simulation unterscheidet. Eine Wiederverwendung der bereits berechneten Pfadbeschreibungen wäre daher nicht möglich.

Ein weiterer Punkt, der gegen diese Realisierungform spricht, ist die Abhängigkeit der $k$-Lookahead-Simulationen von den -Transitionen des Automaten. Die symbolischen Transitionsbedingungen des Automaten müssten bei dieser Realisierungsform vollständig materialisiert und durch die Menge ihrer Belegungen ausgedrückt werden. Dies käme faktisch einer Übersetzung eines SNBAs in einen expliziten NBA gleich. Dadurch würde der Vorteil durch die Verwendung von SNBAs verloren gehen.

### 4.2.2 Realisierung mithilfe von SAT-Solvern

Statt der expliziten Modellierung der Minimierungsverfahren erscheint es aus oben genannten Gründen sinnvoller, symbolische Verfahren zur Berechnung der Vorgänger-Operatoren zu verwenden. In Abschnitt 3.6 wurden bereits die Vorarbeiten zu dieser Form der Implementierung geleistet, indem die Vorgänger-Operatoren mithilfe von quantifizierter Aussagenlogik ausgedrückt wurden.

Die symbolische Berechnung der Simulationsrelationen und der verschiedenen Minimierungstechniken richtet sich nach dem Vorgehen aus dem

symbolischen Model Checking [McM93]. Dabei wird das Verhalten der betrachteten Systeme durch Transitionssysteme beschrieben. Diese Transitionssysteme wiederum werden gemeinsam mit den zu verifizierenden Eigenschaften durch Formeln in verschiedenen Logiken ausgedrückt, zum Beispiel auch durch aussagenlogische Formeln. Die Verifikation des Systems basiert dann darauf, die resultierenden Formeln auf ihre Erfüllbarkeit zu prüfen.

Verschiedene Verfahren zur symbolischen Berechnung von Systemabstraktionen werden in [LBC03] betrachtet. Dort werden Realisierungen mit binären Entscheidungsdiagrammen und mithilfe von SAT-Solvern verglichen. In [Cla+05] wird beschrieben, wie der Einsatz von SAT-Solvern bestehende symbolische Verifikationsverfahren verbessern kann.

Die Realisierung auf Basis von SAT-Solvern stößt jedoch spätestens an ihre Grenzen, wenn es um die Ausführung der Automaten auf Eingabesequenzen geht. Da SAT-Solver standardmäßig nur die Erfüllbarkeit einer aussagenlogischen Formel überprüfen, können durch sie keine nur teilweise ausgewerteten Formeln bestimmt werden. Zudem verlangen sie häufig die Eingabe der Formel in einem bestimmten Format, wie zum Beispiel in konjunktiver Normalform.

### 4.2.3 Realisierung mithilfe von binären Entscheidungsdiagrammen

Statt der Realisierung auf Basis von SAT-Solvern wurde das vollsymbolische Minimierungsframework mithilfe von *binären Entscheidungsdiagrammen* (BDD) implementiert, da diese nicht die oben genannten Einschränkungen aufweisen und ein BDD nach Umbenennung der Variablen als Eingabe für andere Berechnungen dienen kann.

Ein binäres Entscheidungsdiagramm beschreibt eine boolesche Funktion durch einen gerichteten, azyklischen Graphen. Wir verwenden für unsere Implementierung Reduced Ordered BDDs (ROBDD). ROBDDs

erweitern das Konzept von binären Entscheidungsdiagrammen dahingehend, dass eine Ordnung auf den Variablen der BDDs [Bry86] definiert ist und isomorphe Teilgraphen identifiziert wurden [Bry92]. Da ROBDDs die vorherrschende Variante von BDDs sind, werden wir sie auch kurz als BDDs bezeichnen. Um eine effiziente Implementierung zu erhalten, werden wir darüber hinaus eine BDD-Bibliothek verwenden, die BDDs mit geteilten Strukturen anbietet [BRB90], d. h. mehrere BDDs verwenden dieselben Datenstrukturen zur Darstellung gleicher Teilgraphen.

In [Bry92] werden auch Verfahren zur symbolischen Repräsentation von Mengen und Relationen beschrieben. Diese Verfahren werden wir in Abschnitt 4.4 nutzen, um das vollsymbolische Minimierungsframework zu implementieren. Dabei wird für jede Menge des Automaten und die Transitionsfunktion ein BDD aufgebaut, der diese Mengen beschreibt.

## 4.3 Beschreibung der BDD-Bibliothek JavaBDD

In diesem Kapitel wird die Java-Bibliothek JavaBDD[4] beschrieben. Die Bibliothek bietet eine objektorientierte API zur Manipulation von BDDs und orientiert sich an der API von BuDDy[5], einer BDD-Bibliothek für die Sprache C.

JavaBDD ist so entworfen worden, dass es als Wrapper für verschiedene andere BDD-Bibliotheken dienen kann, darunter

- das bereits genannte BuDDy,
- CUDD[6],
- CAL[7] und
- JDD[8].

[4] `javabdd.sourceforge.net/`
[5] `buddy.sourceforge.net`
[6] `vlsi.colorado.edu/~fabio/CUDD`
[7] `embedded.eecs.berkeley.edu/Research/cal_bdd/`
[8] `javaddlib.sourceforge.net/jdd/`

Außerdem enthält JavaBDD auch eine eigene, rein java-basierte Implementierung der BDD-Datenstrukturen und Algorithmen. Im Folgenden wollen wir kurz die wichtigsten Klassen der Bibliothek aus Abbildung 4.3 vorstellen.

Wir beginnen mit der wichtigsten Klasse BDD, diese repräsentiert einen BDD innerhalb der Bibliothek und bietet Möglichkeiten, diesen zu bearbeiten und mit anderen BDDs zu verrechnen. So kann durch die Methoden `var`, `high` und `low` der zugrunde liegende Entscheidungsbaum vollständig durchsucht werden. Neben den Methoden `and`, `or` und `not` stehen noch weitere Methoden zur Verfügung, die zwei BDDs miteinander kombinieren und einen neuen BDD liefern, der das Ergebnis der Operation beschreibt. Ein BDD kann durch `exist` bzw. `forAll` existentiell oder universell quantifiziert werden. Die Methode `restrict` erlaubt es, bestimmte Variablen innerhalb eines BDDs auf konstante Werte zu setzen. Durch `replace` können ausgewählte Variablen innerhalb eines BDDs durch andere BDDs ersetzt werden. Dadurch ist es möglich, die Variablen zu ändern, von denen der BDD abhängig ist.

Die zweite Klasse, die wir betrachten, ist BDD`Factory`, die als zentrale Schnittstelle für die Erzeugung und Manipulation von BDDs fungiert. Bevor die Bibliothek verwendet werden kann, muss sie durch den Aufruf von `init` initialisiert werden, dabei wird die initiale BDD-Knotenanzahl und Cache-Größe übergeben. Die Methode `ithVar` liefert einen BDD, der genau die Variable mit dem übergebenen Index beschreibt. Durch `universe` und `zero` erhält man den konstant wahren bzw. konstant falschen BDD. Durch `extDomain` wird eine neue Repräsentationsdomäne erzeugt, die bis zu `size` Elemente beschreiben kann. Nach der Nutzung der Bibliothek wird diese durch Aufruf von `done` deinitialisiert.

BDD`Pairing` beschreibt Zuordnungen zwischen zwei Variablen bzw. einer Variable und einem BDD. Diese werden benötigt, um Ersetzungen von Variablen in BDDs vorzunehmen.

Die Klasse BDD`Domain` beschreibt eine Repräsentationsdomäne, über deren Variablen die BDDs definiert werden. Wie bereits in Unterabschnitt 3.1.1 beschrieben, werden Repräsentationsdomänen verwendet,

um endliche Mengen symbolisch zu beschreiben. Die Methode `ithVar` liefert einen BDD zurück, der den $i$-ten Wert der repräsentierten endlichen Menge kodiert, und durch `set` erhält man ein BDD`Varset`, das die Variablen der Domäne enthält.

Allgemeiner beschreibt ein BDD`VarSet` eine Menge von BDD-Variablen, die an verschiedenen Stellen der Bibliothek verwendet werden. Durch die Methoden `intersect` und `union` können der Schnitt bzw. die Vereinigung solcher Variablen-Mengen bestimmt werden.

## 4.4 Implementierung des vollsymbolischen Minimierungsframeworks

Dieser Abschnitt beschreibt, wie das in Abschnitt 3.6 entwickelte Minimierungsframework in Scala realisiert und in die in Abschnitt 4.1 beschriebene Logik- und Automaten-Bibliothek RltlConv integriert wurde. Die Implementierung verwendet die im vorherigen Abschnitt beschriebene BDD-Bibliothek.

Wie Abbildung 4.4 zu entnehmen ist, sind die bisher beschriebenen Automaten-Klassen innerhalb von RltlConv alle dem Paket `automata` zugeordnet. Der Großteil, der in dieser Arbeit erstellten Implementierung, wurde in dem Paket `symbolic` angesiedelt. Wir werden später noch genauer auf die einzelnen Klassen und Objekte aus `symbolic` eingehen.

### 4.4.1 Modellklassen zur Automatenrepräsentation

Wir werden in diesem Unterabschnitt zuerst die Klassen- und Objektstrukturen vorstellen, die zur Modellierung der symbolischen und vollsymbolischen NBAs dienen.

Bei der Implementierung der theoretischen Konzepte wurde darauf geachtet, nach Möglichkeit mit unveränderbaren Datenstrukturen zu arbeiten und somit die Arbeit mit den Klassen zu vereinfachen.

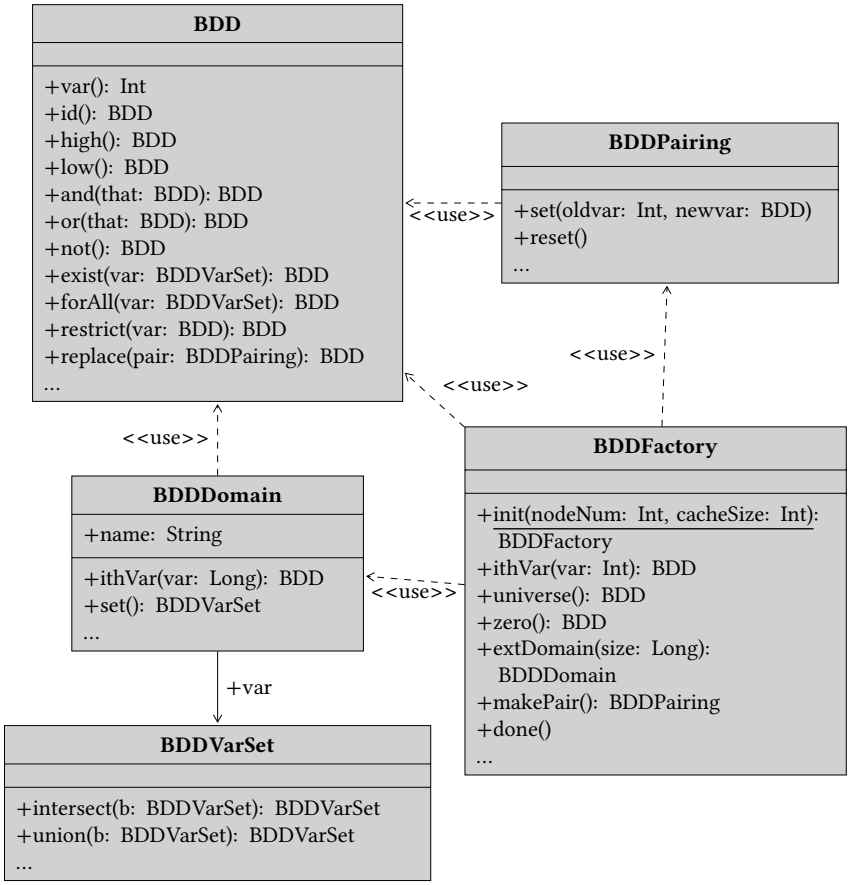



Abbildung 4.3: UML-Klassendiagramm für die wichtigsten Klassen der BDD-Bibliothek JavaBDD.

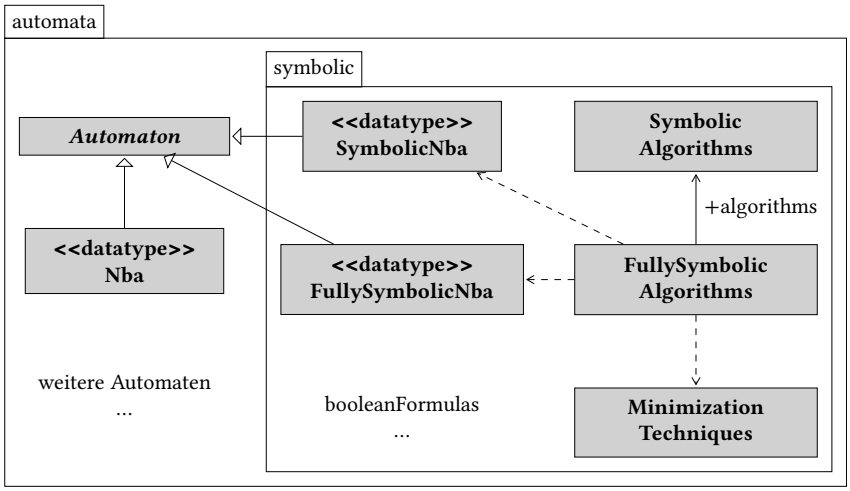



Abbildung 4.4: UML-Paketdiagramm zur groben Einordnung der im Rahmen dieser Arbeit erstellten Klassenstrukturen.

### Repräsentation aussagenlogischer Formeln

Wie in Unterabschnitt 3.1.2 beschrieben, werden die Transitionsbedingungen in SNBAs durch aussagenlogische Formeln über einer Menge von Variablen $V_\Sigma$ beschrieben, daher wurden zuerst die Klassen und Objekte aus Abbildung 4.5 erstellt. Diese beschreiben aussagenlogische Formeln über einer Menge von Elementen und orientieren sich dabei an der bereits vorhandenen Klassenhierarchie zur Modellierung positiver boolescher Kombinationen.

Zentrales Element ist der Trait `Bool`, welcher verschiedene Methoden für alle implementierenden Klassen und Objekte vorschreibt, die technischen Zwecken dienen. `Bool` ist durch den Typ-Parameter A parametrisiert, dies erlaubt es, aussagenlogische Formeln über beliebigen Scala-Typen zu erzeugen. In der vorhandenen Implementierung wird dies jedoch ausschließlich für den `String`-Typen verwendet.

Die Case-Objekte `True` und `False` modellieren die beiden Konstanten true bzw. false der aussagenlogischen Formeln. Sie binden beide den

Typ-Parameter A der abstrakten Klasse `Atom` an den Typen `Nothing`. `Nothing` stellt in Scala einen besonderen Typen dar, da er ein Untertyp jedes anderen Scala-Typen ist. Damit liegt er am anderen Ende des Spektrums von `Any`, das, ähnlich wie `Object` in Java, Obertyp jedes anderen Typen ist.

Die Case-Klasse `Element` beschreibt ein einzelnes Element der zugrundeliegenden Menge $P$ von Variablen. Die Case-Klassen `Not`, `And` und `Or` modellieren die entsprechenden Operatoren der aussagenlogischen Formeln. Die ihnen zugeordneten Kompagnon-Objekte bieten verschiedene Methoden zur Verarbeitung von `Bool`-Objekten an, die aus rein technischen Gründen existieren.

Damit bleibt nur noch das Objekt `SimpleBool` zu beschreiben. Während die bisher beschriebenen Klassen und Objekte ausschließlich dazu dienen, einen abstrakten Syntaxbaum der aussagenlogischen Formeln zu bilden, nehmen die Methoden in `SimpleBool` während der Erzegung bereits Vereinfachungen der Struktur vor. Dadurch werden bestimmte algebraische Vereinfachungen bereits frühzeitig durchgeführt. Dieses Konzept ist aus anderen funktionalen Programmiersprachen auch als „Smart Constructors“ bekannt[Ada93].

### Repräsentation von SNBAs und FSNBAs

Abbildung 4.6 zeigt die zur Modellierung von SNBAs und FSNBAs verwendeten Klassen `SymbolicNba` und `FullySymbolicNba`. Zusätzlich ist auch noch einmal die Klasse `Nba` aufgeführt, die der Repräsentation expliziter 2NBAs dient. Vergleicht man die Struktur der drei Klassen, so erkennt man sofort die Ähnlichkeiten.

Die Case-Klasse `SymbolicNba` stellt symbolische NBAs ganz ähnlich der Klasse `Nba` dar. Die Attribute `states`, `start` und `accepting` beschreiben die Zustände des Automaten durch `State`-Mengen. Statt eines Alphabets wird eine sortierte Menge von Propositionen gespeichert. Daher beschreibt `propositions` die Repräsentationsdomäne

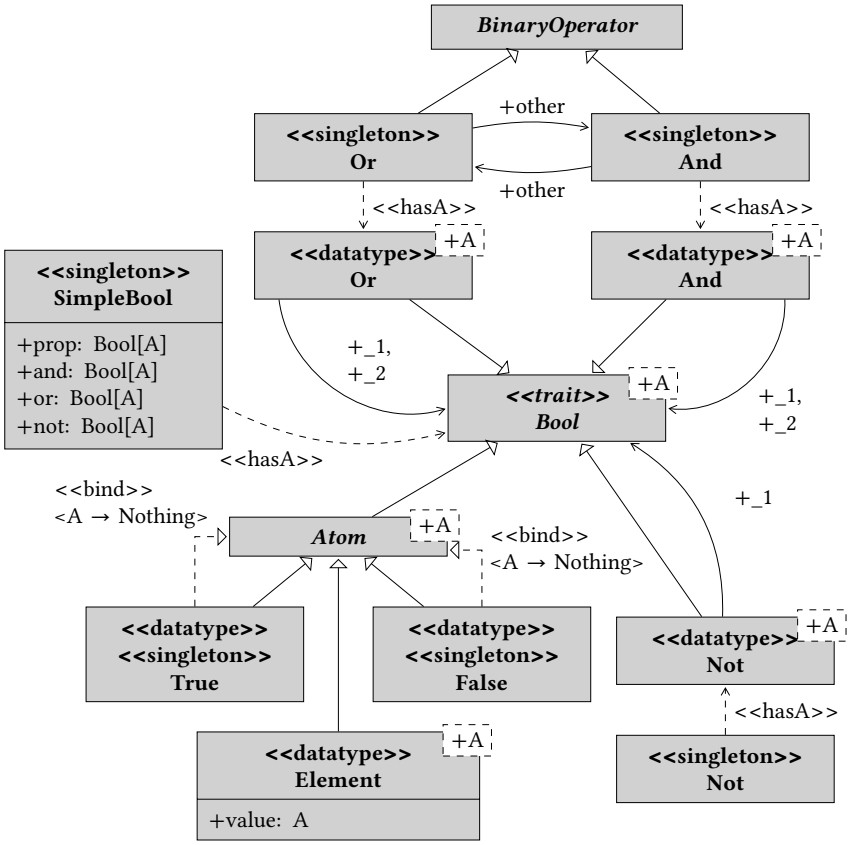



Abbildung 4.5: UML-Klassendiagramm für die Modellklassen, die aussagenlogische Formeln über einer Menge $P$ aussagenlogischer Variablen modellieren.

$V_\Sigma$ und $f_\Sigma$ wird durch das Attribut `validPropositions` beschrieben. Die Transitionsfunktion $\delta$ wird sehr direkt durch `transitions` beschrieben. Das <<lazy>> Attribut `complete` liefert den Automaten korrekt vervollständigt. Es wurde ein <<lazy>> Attribut für die Realisierung verwendet, da ein Automat ohne weitere Parameter vervollständigt werden kann, gleichzeitig aber nicht jeder Automat vollständig sein muss.

Wie bereits gesagt, verwendet die erstellte Realisierung die BDD-Bibliothek JavaBDD. Dies zeigt sich sehr deutlich in der Case-Klasse `FullySymbolicNba`, die der Modellierung von FSNBAs dient. Die drei Repräsentationsdomänen $V_\Sigma^0$, $V_Q^0$ und $V_Q^1$ wurden direkt auf BDD`Domain` abgebildet und die restlichen booleschen Funktionen $f_\Sigma$, $f_Q, f_I, f_\delta$ und $f_F$ wurden durch `BoolFun1` bis `BoolFun3` realisiert.

Die `BoolFun` Case-Klassen beschreiben boolesche Funktionen, wobei die Quellmenge der booleschen Funktionen durch BDD`Domains` und die Graphen der Funktionen durch BDD beschrieben werden. Da die Verwendung von BDDs ein manuelles Speichermanagement notwendig macht, implementieren die Case-Klassen das Java-Interface `Closeable`, wodurch ihre Verwendung im Kontext von automatischem Ressourcen-Management möglich wird. Genauso implementieren alle Datenstrukturen, die auf ihnen aufbauen, das `Closeable`-Interface.

Da die BDD-Klassen in ihrem Kern veränderbar sind, mussten bei der Implementierung der `FullySymbolicNba`-Klasse besondere Schritte ergriffen werden, um die beiden Ansätze zu vereinen. Dem ist auch das Vorhandensein der Methode `id` geschuldet, die eine exakte Kopie des Automaten zurück liefert, die vollkommen unabhängig von dem ursprünglichen Automaten ist. Auch die `close`-Methode lässt sich auf die manuellen Speicherverwaltungsbedürfnisse zurückführen. Und auch für den FSNBA existiert ein `complete`-Attribut, das ganz analog zu dem SNBA realisiert wurde.

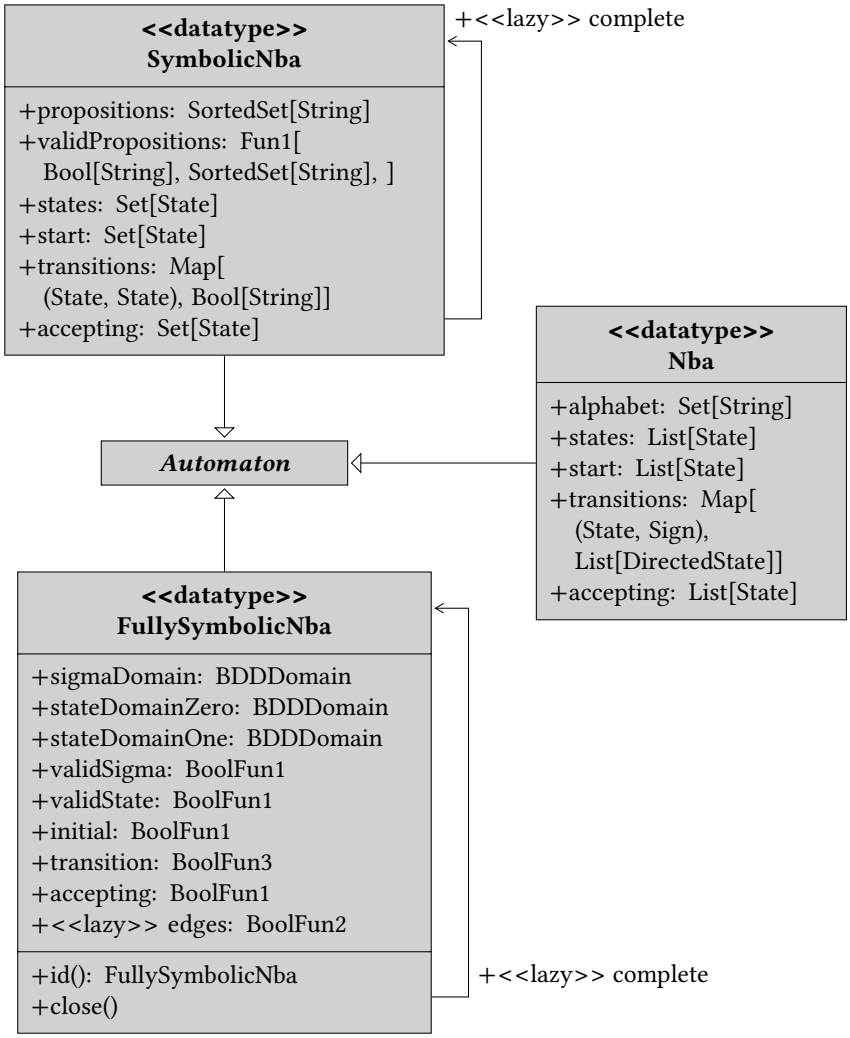



Abbildung 4.6: UML-Klassendiagramm der Modellklassen zur Repräsentation von SNBAs und FSNBAs. Zum Vergleich ist auch die Modellklasse für NBAs dargestellt.

### 4.4.2 Klassenstruktur zur Realisierung der Algorithmen

Nachdem im vorherigen Unterabschnitt die Klassen zur Repräsentation von Automaten vorgestellt wurden, werden wir in diesem Abschnitt beschreiben, wie die in dieser Arbeit entworfenen Algorithmen implementiert wurden.

#### Beschreibung der automatenunabhängigen Algorithmen

Zunächst beschreiben wir die Klasse `SymbolicAlgorithms` und das zugehörige Kompagnon-Objekt aus Abbildung 4.7, welche als zentrale Schnittstelle zur Minimierung von Automaten dienen. Da es vorgesehen ist, mehrere Automaten gleichzeitig bzw. direkt nacheinander zu minimieren, wurde dieser Ansatz gewählt. Dadurch können die Initialisierungsschritte des Minimierungsframeworks für die Minimierung verschiedener Automaten wiederverwendet werden. Die Methode `apply` erwartet daher die maximalen Kennwerte der zu minimierenden Automaten, ihre Zustandsanzahl und die Repräsentationsvariablenanzahl sowie den maximal betrachteten Lookahead. Als Ergebnis wird ein Objekt vom Typ `SymbolicAlgorithms` zurückgeliefert, das mit einer ausreichenden Anzahl an korrekt dimensionierten Repräsentationsdomänen initialisiert ist.

Zusätzlich enthält das Kompagnon-Objekt noch verschiedene Hilfsmethoden: Die boolesche Funktion, die die transitive Hülle einer Binärrelation beschreibt, kann mithilfe von `transitiveClosure` berechnet werden. Ebenso berechnet `asymmetricRestriction` die boolesche Funktion, die die asymmetrische Restriktion einer Binärrelation beschreibt. Mithilfe der Methode `fixPointFun2` wiederum kann ein Fixpunkt einer Funktion berechnet werden, die eine zweistellige boolesche Funktion verarbeitet und das Ergebnis zurückliefert.

Ein Objekt vom Typ `SymbolicAlgorithms` hält wie bereits beschrieben die benötigten Repräsentationsdomänen in seinen Attributen vor. Außerdem wird der maximal unterstützte Lookahead in dem Attribut `maxLookahead` gespeichert.

Mit `createAutomatonSpecificAlgorithms` wird ein Objekt vom Typ `FullySymbolicNbaAlgorithms` erzeugt, das die Algorithmen mit spezifischem Bezug zu einem bestimmten FSNBA implementiert.

Neben der Erzeugung von automatenspezifischen Algorithmensammlungen ist die Klasse `SymbolicAlgorithms` auch für die Berechnung der verschiedenen vollsymbolischen Vorgänger-Operatoren zuständig. Hier ist exemplarisch nur die Methode `CPreDirect` aufgeführt, die für einen FSNBA und den gewählten Lookahead das Ergebnis des direkten vollsymbolischen Vorgänger-Operators $\mathrm{CPre}_f^{\mathrm{di}}$ berechnet. Insgesamt existieren fünf dieser Methoden für die Berechnung der entsprechenden Vorgänger-Operatoren:

- `CPreBackward` für $\mathrm{CPre}_f^{\mathrm{bw}}$
- `CPreDirect` für $\mathrm{CPre}_f^{\mathrm{di}}$
- `CPreDelayedOne` für $\mathrm{CPre}_f^{1}$
- `CPreDelayedTwo` für $\mathrm{CPre}_f^{2}$
- `CPreFair` für $\mathrm{CPre}_f$

Weiterhin ist die Berechnung der $k$-Lookahead-Simulationen in `SymbolicAlgorithms` angesiedelt. Auch hier ist exemplarisch nur die Methode `lookaheadRelDirect` aufgeführt, die für einen FSNBA und den gewählten Lookahead die zweistellige boolesche Funktion berechnet, die die direkte $k$-Lookahead-Simulation $\sqsubseteq^{k\text{-}\mathrm{di}}$ auf dem vollsymbolischen NBA beschreibt. Genauso existieren auch die Methoden

- `lookaheadRelBackward` für $\sqsubseteq^{k\text{-}\mathrm{bw}}$,
- `lookaheadRelDirect` für $\sqsubseteq^{k\text{-}\mathrm{di}}$,
- `lookaheadRelDelayed` für $\sqsubseteq^{k\text{-}\mathrm{de}}$ und
- `lookaheadRelFair` für $\sqsubseteq^{k\text{-}\mathrm{f}}$.

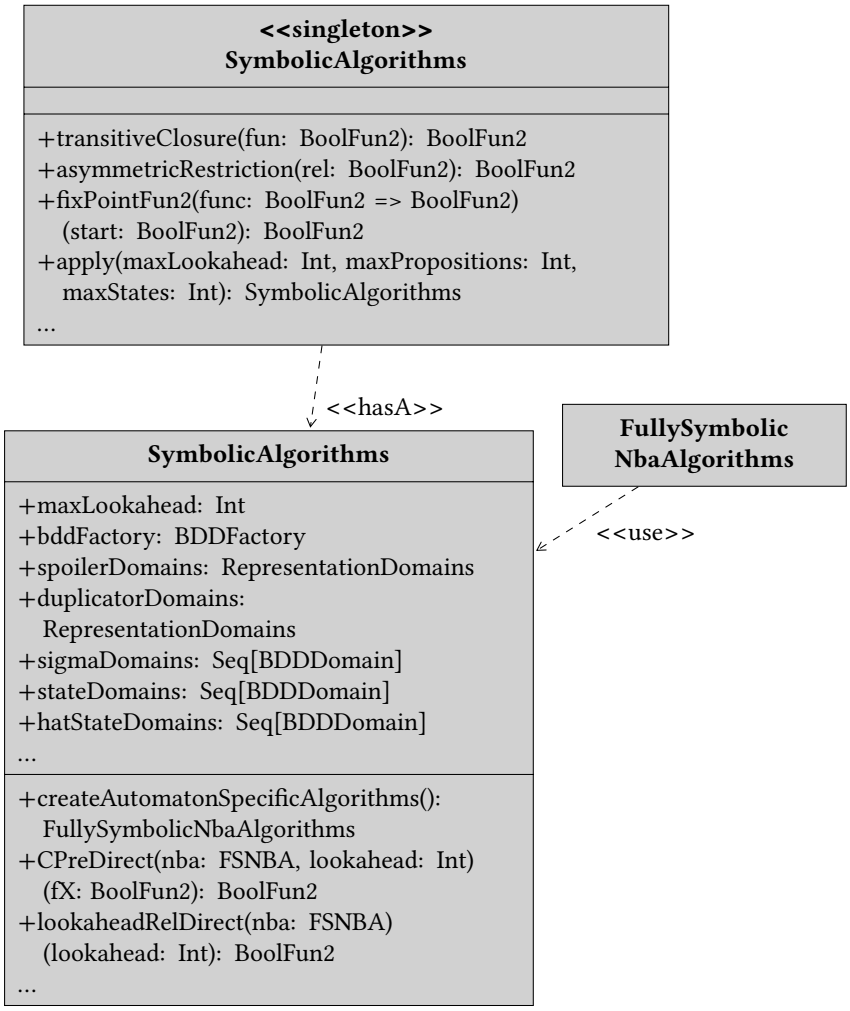



Abbildung 4.7: UML-Klassendiagramm der Algorithmenklassen, die die in dieser Arbeit entworfenen Algorithmen implementieren. Diese Abbildung zeigt nur die allgemeinen Klassen für die Implementierung der vollsymbolischen Algorithmen, der Rest ist in Abbildung 4.8 dargestellt.

### Beschreibung der automatenspezifischen Algorithmen

Wir beginnen mit der Beschreibung der zentralen Algorithmen-Klasse `FullySymbolicNbaAlgorithms`, die alle Algorithmen und Verfahren enthält, die spezifisch für vollsymbolische NBAs sind. Um die Algorithmen zu implementieren, nutzt die Klasse einige der Möglichkeiten, die die Klasse `SymbolicAlgorithms` bietet. Daher wird das erzeugende Algorithmen-Objekt als Attribut gespeichert, da die Korrektheit der Algorithmen nur gegeben ist, wenn sie auf denselben BDD-Strukturen aufbauen. Um sicherzustellen, dass die produzierten BDDs über den richtigen Repräsentationsdomänen konstruiert werden, werden diese zum fortlaufenden Abgleich in `representationDomains` gespeichert. Die Case-Klasse `RepresentationDomains` dient ausschließlich der strukturierten Speicherung dieser Domänen.

Für alle enthaltenen Funktionen gilt, dass Zwischenergebnisse der Berechnungen nach Möglichkeit zur Wiederverwendung gespeichert werden. Dies ist einer der Gründe, warum die automatenspezifischen Algorithmen in eine eigene Klasse ausgelagert wurden. Ein weiterer Grund ist, dass so die Wiederverwendung der vollsymbolischen Vorgänger-Operatoren aus `SymbolicAlgorithms` für die Minimierung anderer vollsymbolischer Automatenmodelle ermöglicht wird.

Ein symbolischer NBA wird durch `representSymbolicNba` in einen vollsymbolischen NBA umgewandelt, welcher durch die verschiedenen anderen Methoden verarbeitet wird. Zum Beispiel werden die toten Zustände durch `livingAutomaton` aus dem FSNBA entfernt. Der vollsymbolische, gestutzte Automate aus Definition 3.19 wird durch die Methode `prunedAutomaton` berechnet, `quotientAutomaton` hingegen berechnet den vollsymbolischen Repräsentantenautomaten aus Definition 3.22. Beide Methoden erwarten den Ausgangsautomaten sowie die zu verwendende Relation als Parameter.

Die Rückübertragung vom vollsymbolischen zum symbolischen NBA bzw. zur expliziten Zustandsrelation auf $Q \times Q$ übernehmen die Methoden `toSymbolicNba` und `symbolicToExplicitRelation`. Damit bleiben nur noch die Methoden `heavyK` und `lightK`, diese

implementieren die Minimierungsverfahren Heavy-$k$ und Light-$k$ aus Unterabschnitt 2.5.4.

Für die Umsetzung dieser Verfahren nutzen sie die Klasse `MinimizationTechniques`, die für jede, der in dieser Arbeit vorgestellten Techniken zur Minimierung von NBAs, ein Funktionsobjekt enthält, das diese Technik auf einen FSNBA anwendet. So beschreibt die Funktion `prune_id_atDiL` zum Beispiel die Berechnung des gestutzten Automaten bezüglich $P(id, \prec^{k\text{-di}})$ und `quotient_tBwL` die Quotientenbildung bezüglich $\preceq^{k\text{-bw}}$. Diese Funktionsobjekte wiederum verwenden für die Durchführung der Technik die bisher beschriebenen Algorithmen-Klassen.

Außerdem bietet die Klasse noch die Methode `chainTechniques` an, die eine beliebige Anzahl an Minimierungstechniken als Parameter erwartet und als Ergebnis eine Minimierungstechnik liefert, die der Verkettung aller übergebenen Techniken entspricht. Ganz ähnlich ist die Methode `fixpoint` gestaltet, jedoch berechnet diese zusätzlich noch den Fixpunkt über den übergebenen Techniken.

Durch die Umsetzung als Closures wird eine funktionale Realisierung von Light-$k$ wie in Listing 4.1 möglich. In Zeile 4 wird das Hilfsobjekt mit den Minimierungstechniken für den gewählten Lookahead erzeugt. Das eigentliche Minimierungsverfahren wird in den Zeilen 7 bis 9 durch Verkettung der Basistechniken definiert. In Zeile 11 wird dann das Verfahren auf den FSNBA angewendet und das Ergebnis zurückgegeben.

Die Implementierung von Heavy-$k$ aus Listing 4.2 zeigt, dass die Basistechniken auch in anderer Kombination verknüpft werden können. So wird in Zeile 7 bis 13 das Verfahren als Fixpunkt über der Verkettung der Basistechniken definiert. Es wäre dadurch auch möglich, die Techniken vollkommen neu zu kombinieren und verschiedene Minimierungsverfahren durch ihre Verschachtelung und Verkettung zu definieren.

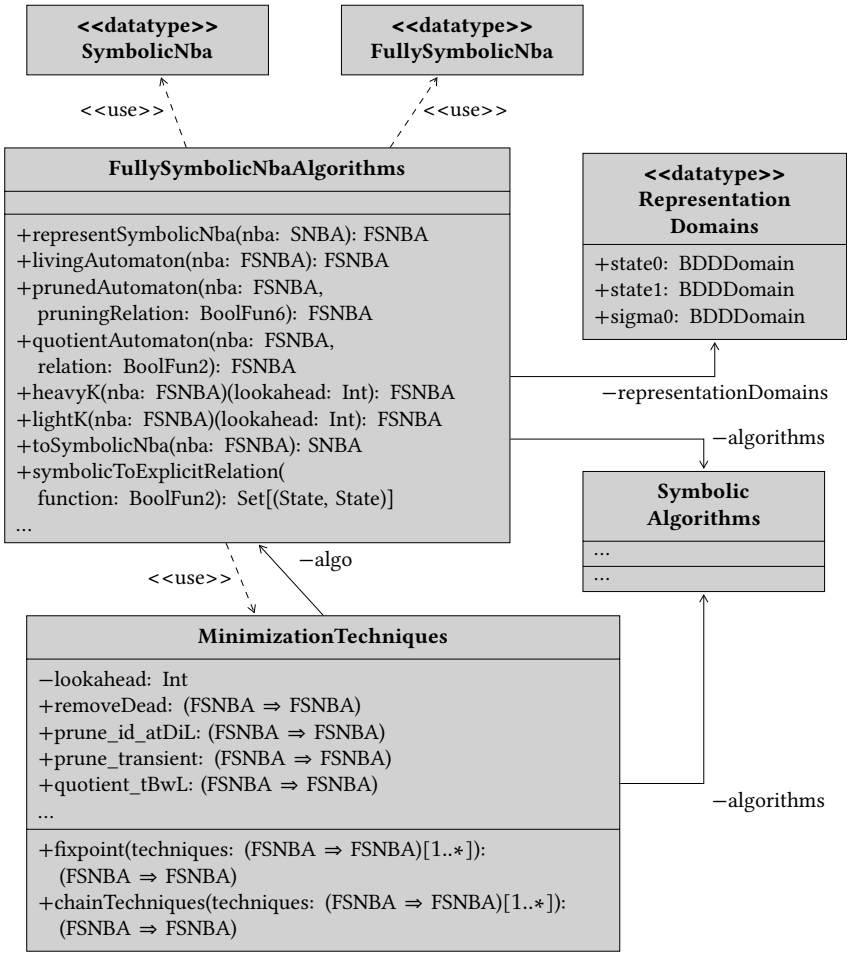



Abbildung 4.8: UML-Klassendiagramm der Algorithmenklassen, die die in dieser Arbeit entworfenen Algorithmen implementieren. Diese Abbildung zeigt nur einen Teil der verwendeten Klassen, der Rest ist in Abbildung 4.7 dargestellt. Die Typen SNBA und FSNBA sind als Alias für `SymbolicNba` und `FullySymbolicNba` zu verstehen.

Listing 4.1: Implementierung des Light-*k* Verfahrens

```
def lightK(nba: FullySymbolicNba)(lookahead: Int):
    FullySymbolicNba = {
  // Erzeugen der Minimierungstechniken
  val techs = techniques(lookahead)

  // Verketten der Techniken
  val minimizer = techs.chainTechniques(
      techs.removeDead,
      techs.quotient_tDeL)

  minimizer(nba) // Ergebnis bestimmen
}
```

Listing 4.2: Implementierung des Heavy-*k* Verfahrens

```
def heavyK(nba: FullySymbolicNba)(lookahead: Int):
    FullySymbolicNba = {
  // Erzeugen der Minimierungstechniken
  val techs = techniques(lookahead)

  // Verketten der Techniken und Fixpunkt
  val minimizer = techs.fixpoint(
      techs.removeDead,
      techs.prune_id_atDiL,
      ... ,
      techs.prune_transient,
      techs.quotient_tDeL,
      techs.quotient_tBwL)

  minimizer(nba) // Ergebnis bestimmen
}
```

### 4.4.3 Verwendungsbeispiel der Implementierung

Im vorherigen Unterabschnitt haben wir beschrieben, wie die verschiedenen Algorithmen als Klassen realisiert wurden. Darauf aufbauend beschreiben wir in diesem Abschnitt, wie eine Minimierung durch das Framework abläuft.

Abbildung 4.9 zeigt den allgemeinen Ablauf der Minimierung von zwei SNBAs, dabei wurden aus Platzgründen abkürzende Typbezeichnungen gewählt. Wir gehen davon aus, dass diese bereits als Objekte `snba1` und `snba2` vom Typ `SymbolicNba` vorliegen. Weiterhin umfasst die Repräsentationsdomäne $V_\Sigma^1$ von `snba1` acht Variablen, während $V_\Sigma^2$ nur zwei Variablen umfasst. Für Anzahl der Zustände gilt wiederum $|Q_1| = 4$ und $|Q_2| = 14$.

Um nun die beiden SNBAs zu minimieren, erzeugt der Nutzer zuerst ein Objekt `algorithms` vom Typ `SymbolicAlgorithms` und übergibt den maximalen Lookahead, für den das Framework initialisiert werden soll. Außerdem wird die maximale Anzahl an Propositionen sowie Zuständen übergeben. Das `algorithms`-Objekt erzeugt ausgehend von diesen Kennwerten die entsprechenden Repräsentationsdomänen, sodass alle Propositionen und Zustände repräsentiert werden können.

Der Nutzer fordert ein Objekt `algo1`, das von da an die weitere Verarbeitung des ersten SNBAs übernimmt.

Zuerst wird der SNBA `snba1` mit `representSymbolicNba` in einen FSNBA `fsnba1` überführt. Der so erzeugte FSNBA wird durch Übergabe an `heavyK` minimiert, dabei kann ein beliebiger Lookahead bis zum zuvor gewählten maximalen Lookahead verwendet werden. Die interne Verarbeitung der BDDs verwendet dann auch nur die entsprechende Menge von Repräsentationsdomänen. Die Verarbeitung innerhalb von `heavyK` ähnelt stark dem in Abbildung 4.10 beschriebenen Ablauf von `lightK`. Das Ergebnis der Minimierung `fsnba1Min` kann dann für andere Zwecke weiterverwendet werden oder durch `toSymbolicNba` in `algo1` wieder in einen SNBA übertragen werden.

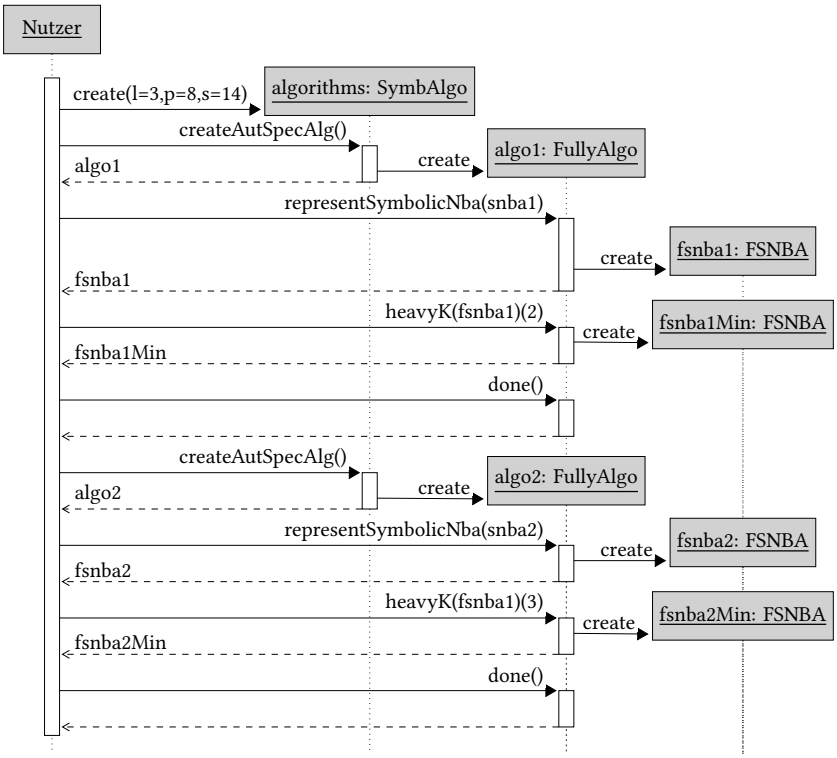



Abbildung 4.9: Allgemeiner Ablauf der Minimierung von zwei SNBAs `snba1` und `snba2`. Diese werden erst in FSNBAs überführt und dann getrennt minimiert.

Sind alle Schritte bezüglich `snba1` durchgeführt worden, so wird `algo1` durch Aufruf von `done` deinitialisiert.

Die Minimierung des zweiten SNBAs erfolgt vollkommen analog. Das Erzeugen von `algorithms` entfällt jedoch, da dieses Objekt bereits erzeugt wurde und das Minimierungsframework noch initialisiert ist.

Nachdem wir den allgemeinen Ablauf der Minimierung beschrieben haben, wollen wir auch exemplarisch den in Abbildung 4.10 dargestellten Ablauf von `lightK` beschreiben. Wir gehen davon aus, dass durch

`fsnba` ein beliebiger FSNBA beschrieben wird. Die Minimierung nach dem Light-$k$ Verfahren wird durch den Aufruf von `lightK` eingeleitet, in diesem Beispiel wurde ein Lookahead von 2 gewählt.

Zuerst wird ein Hilfsobjekt `techs` erzeugt, dieses stellt, wie bereits beschrieben, die verschiedenen Minimierungstechniken zur Verfügung. Durch `chainTechniques` werden diese verkettet und die Funktion `minimizer`, die diese Techniken hintereinander ausführt, wird erzeugt.

Danach wird die Funktion `minimizer` auf den FSNBA angewendet. Für die Berechnung des minimierten Automaten greift `minimizer` wieder auf die Objekte `algo` und `algos` zurück.

Die Minimierungstechnik `removeDead` entfernt mithilfe von `livingAutomaton` die toten Zustände. Ausgehend davon berechnet `quotient_tDeL` den Repräsentantenautomaten des lebendigen Automaten. Dafür wird zuerst die verzögerte 2-Lookahead-Simulation in `algos` bestimmt. Die so berechnete Relation wird dann verwendet, um den Repräsentantenautomaten mit `quotientAutomaton` zu bestimmen.

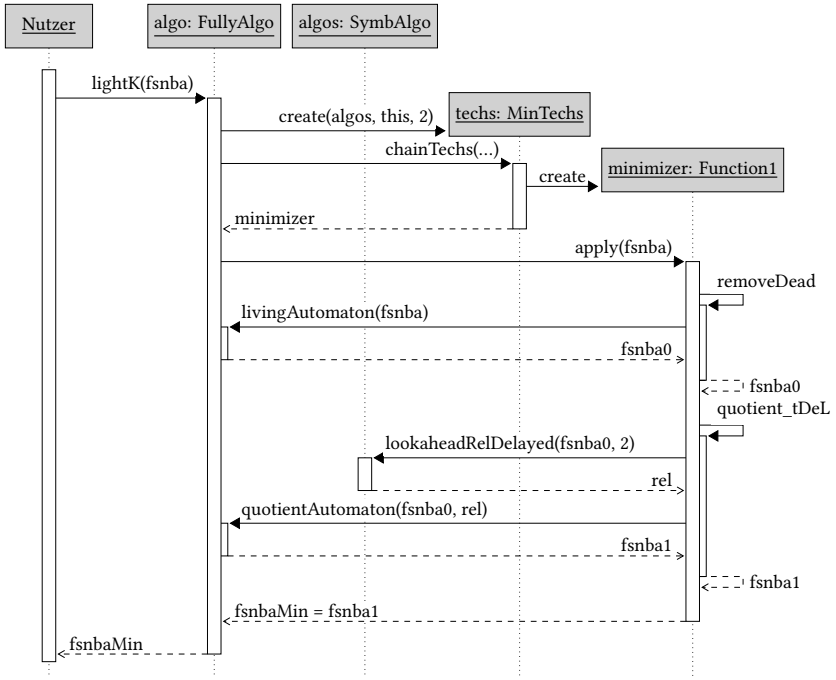



Abbildung 4.10: Ablauf der Methode `lightK`, die das Light-$k$ Verfahren implementiert. Aus Platzgründen wurden abkürzende Schreibweisen verwendet.

# 5 Zusammenfassung und Ausblick

Abschließend fassen wir noch einmal die in dieser Arbeit entwickelten Automatenmodelle und die auf ihnen basierenden Minimierungsverfahren zusammen. Außerdem bieten wir einen Ausblick auf mögliche weiterführende Fragestellungen.

Zuerst wurden die symbolischen nichtdeterministischen Büchi-Automaten als Erweiterung der bekannten expliziten Büchi-Automaten definiert. Diese Form der NBAs lässt sich deutlich prägnanter beschreiben, als es mit expliziten NBAs möglich ist. Dadurch ergibt sich auch ihre besondere Eignung für die Beschreibung von Spezifikationen in der Verifikation von Softwaresystemen und außerdem können mit ihrer Hilfe die $\omega$-regulären Sprachen zur Beschreibung von erwünschtem und unerwünschtem Systemverhalten sehr präzise spezifiziert werden.

Um eine spätere Verarbeitung der Automaten zu erleichtern, wurden die verschiedenen, in [CM13] beschriebenen, Simulationsrelationen, Techniken und Methoden zur Minimierung von expliziten NBAs auf die symbolischen nichtdeterministischen Büchi-Automaten übertragen.

Da die Berechnung der $k$-Lookahead-Simulationen auf SNBAs eine Betrachtung der durch die Transitionen implizierten $\beta$-Transitionen notwendig macht und dies einer nachträglichen Umwandlung in explizite NBAs entspricht, wurden die vollsymbolischen nichtdeterministischen Büchi-Automaten definiert. Nach der Übertragung der Simulationsrelationen und Minimierungstechniken auch auf dieses Automatenmodell ermöglicht dies die effiziente symbolische Berechnung von minimierten Büchi-Automaten, wie sie von den symbolischen Graph-Algorithmen des symbolischen Model-Checkings bekannt ist.

Das während dieser Arbeit entwickelte Minimierungsframework, welches auf FSNBAs basiert, wurde außerdem in der Programmiersprache

Scala implementiert. Die Implementierung nutzt binäre Entscheidungsdiagramme in Gestalt der BDD-Bibliothek JavaBDD zur Repräsentation der quantifizierten aussagenlogischen Formeln, welche die Grundlage des Frameworks bilden. Zudem wurde die Implementierung in die Automaten- und Logikbibliothek RltlConv integriert.

### Ausblick

Für die Zukunft sind verschiedene weitere Schritte und Entwicklungen möglich und wünschenswert. So konnte in dieser Arbeit nicht ausreichend untersucht werden, ob die wiederholten Pruning-Schritte des Heavy-*k* Verfahrens sich auch während der symbolischen Minimierung des Automaten positiv auf die Berechnung auswirken. Da sich gleichartige Strukturen durch BDDs besonders kompakt beschreiben lassen, sollte eine Abwandlung von Heavy-*k* untersucht werden, die eine Transition nur dann stutzt, wenn sich daraus ein strukturell einfacherer BDD mit weniger Knoten ergibt. Eine weitere mögliche Abwandlung ist das Hinzufügen von Transitionen, wenn diese die akzeptierte Sprache nicht verändern, aber gleichzeitig die Menge der BDD-Knoten verringert.

Weitere Verbesserungen der aktuellen Implementierung sollten die Reihenfolge, in der die Operationen auf den BDDs ausgeführt werden, näher betrachten und eine optimale Abfolge der Ausführungen anstreben.

Die in der RltlConv implementierten Umwandlungen von Logiken in Monitore basieren nicht nur auf den expliziten NBAs, sondern auch auf den restlichen, in der Bibliothek enthaltenen, expliziten Automatenmodellen, daher muss bisher nach der symbolischen Minimierung des NBAs dieser wieder in einen expliziten NBA umgewandelt werden, um ihn im weiteren Verlauf der Umwandlung verwenden zu können. Um die Vorteile der symbolischen Berechnung voll ausnutzen zu können, muss die symbolische Repräsentation der Transitionsbedingungen auf weitere Automatenmodelle wie nichtdeterministische Automaten auf endlichen Wörtern, alternierende Büchi-Automaten und alternierende Paritätsautomaten ausgeweitet werden.

Da die Berechnung der $k$-Lookahead-Simulationen nur Teilpfade der Länge $k$ betrachtet, sollte auch eine Implementierung mithilfe der aus dem Bounded-Model-Checking bekannten SAT-Solver noch einmal auf ihre Umsetzbarkeit überprüft werden.

Abschließend kann die durchgehende Verwendung von Automatenmodellen mit symbolischen Transitionsbedingungen in der Softwareverifikation nur als wünschenswert bezeichnet werden, da diese den exponentiellen Blowup in Bezug auf die Anzahl der Transitionen zwischen den Zuständen vermeiden.

# Literatur


[Ada93] Stephen Adams. „Functional Pearls Efficient sets—a balancing act“. In: *Journal of Functional Programming* 3.4 (1993), S. 553–561.

[BRB90] Karl S. Brace, Richard L. Rudell und Randal E. Bryant. „Efficient Implementation of a BDD Package“. In: *Proceedings of the 27th ACM/IEEE Design Automation Conference*. DAC '90. New York, NY, USA: ACM, 1990, S. 40–45.

[Bry86] R.E. Bryant. „Graph-Based Algorithms for Boolean Function Manipulation“. In: *Computers, IEEE Transactions on* C-35.8 (Aug. 1986), S. 677–691.

[Bry92] Randal E. Bryant. „Symbolic Boolean Manipulation with Ordered Binary-decision Diagrams“. In: *ACM Comput. Surv.* 24.3 (Sep. 1992), S. 293–318.

[Bub09] Uwe Bubeck. „Model-based transformations for quantified Boolean formulas“. Dissertation. University of Paderborn, 2009.

[Cla+05] Edmund Clarke u. a. „SATABS: SAT-Based Predicate Abstraction for ANSI-C“. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Hrsg. von Nicolas Halbwachs und LenoreD. Zuck. Bd. 3440. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, S. 570–574.

[Cle11] Lorenzo Clemente. „Büchi Automata Can Have Smaller Quotients“. In: *Automata, Languages and Programming*. Hrsg. von Luca Aceto, Monika Henzinger und Jiří Sgall. Bd. 6756. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, S. 258–270.

[CM13] Lorenzo Clemente und Richard Mayr. „Advanced Automata Minimization“. In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '13. New York, NY, USA: ACM, 2013, S. 63–74.

[DHW92] DavidL. Dill, AlanJ. Hu und Howard Wong-Toi. „Checking for language inclusion using simulation preorders“. In: *Computer Aided Verification*. Hrsg. von KimG. Larsen und Arne Skou. Bd. 575. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1992, S. 255–265.

[EF10] Rüdiger Ehlers und Bernd Finkbeiner. „On the Virtue of Patience: Minimizing Büchi Automata“. In: *Model Checking Software*. Hrsg. von Jaco van de Pol und Michael Weber. Bd. 6349. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, S. 129–145.

[EH00] Kousha Etessami und GerardJ. Holzmann. „Optimizing Büchi Automata“. In: *CONCUR 2000 — Concurrency Theory*. Hrsg. von Catuscia Palamidessi. Bd. 1877. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, S. 153–168.

[Ete02] Kousha Etessami. „A Hierarchy of Polynomial-Time Computable Simulations for Automata“. In: *CONCUR 2002 — Concurrency Theory*. Hrsg. von Luboš Brim u. a. Bd. 2421. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, S. 131–144.

[EWS01] Kousha Etessami, Thomas Wilke und RebeccaA. Schuller. „Fair Simulation Relations, Parity Games, and State Space Reduction for Büchi Automata“. In: *Automata, Languages and Programming*. Hrsg. von Fernando Orejas, PaulG. Spirakis und Jan van Leeuwen. Bd. 2076. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, S. 694–707.

[HKR02] Thomas A. Henzinger, Orna Kupferman und Sriram K. Rajamani. „Fair Simulation“. In: *Information and Computation* 173.1 (2002), S. 64–81.

[Hud+07] Paul Hudak u. a. „A History of Haskell: Being Lazy with Class“. In: *Proceedings of the Third ACM SIGPLAN Conference*

*on History of Programming Languages*. HOPL III. New York, NY, USA: ACM, 2007,

[JR91] Tao Jiang und B. Ravikumar. „Minimal NFA problems are hard“. In: *Automata, Languages and Programming*. Hrsg. von JavierLeach Albert, Burkhard Monien und MarioRodríguez Artalejo. Bd. 510. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1991, S. 629–640.

[LBC03] ShuvenduK. Lahiri, RandalE. Bryant und Byron Cook. „A Symbolic Approach to Predicate Abstraction“. In: *Computer Aided Verification*. Hrsg. von Jr. Hunt WarrenA. und Fabio Somenzi. Bd. 2725. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, S. 141–153.

[McM93] Kenneth L McMillan. *Symbolic model checking*. Springer, 1993.

[Rac09] Meiko Rachimow. „Evaluierung des Einsatzes von Scala bei der Entwicklung für die Android-Plattform“. Diplomarbeit. Berlin: Technische Fachhochschule Berlin, 2009.

[SB00] Fabio Somenzi und Roderick Bloem. „Efficient Büchi Automata from LTL Formulae“. In: *Computer Aided Verification*. Hrsg. von E.Allen Emerson und AravindaPrasad Sistla. Bd. 1855. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, S. 248–263.

[Sch12] Malte Schmitz. „Transformation von regulärer Linearzeit-Temporallogik zu Paritätsautomaten“. Bachelorarbeit. Universität zu Lübeck, 2012.

[Var96] MosheY. Vardi. „An automata-theoretic approach to linear temporal logic“. In: *Logics for Concurrency*. Hrsg. von Faron Moller und Graham Birtwistle. Bd. 1043. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996, S. 238–266.

[Vea13] Margus Veanes. „Applications of Symbolic Finite Automata“. In: *Implementation and Application of Automata*. Hrsg. von Stavros Konstantinidis. Bd. 7982. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, S. 16–23.