technische universität
dortmund

Masterarbeit

Pattern Matching auf FPGAs

Heiko Schwedhelm
März 2016

Gutachter:
Prof. Dr. Jens Teubner
Dr. Louis Woods

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Datenbanken und
Informationssysteme (LS VI)
http://dbis.cs.tu-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

Im Rahmen dieser Masterarbeit wird ein Parsergenerator für das Pattern Matching auf Field-Programmable Gate Arrays (kurz FPGAs) vorgestellt. Die Muster für das Pattern Matching in Datenströmen werden dazu über reguläre Ausdrücke definiert und durch den Parsergenerator als nichtdeterministische endliche Automaten auf Hardwarebausteine von FPGAs abgebildet. Mit der Einbettung von Aktionen können Datenströme auf dem FPGA durch einen generierten Parser gefiltert, komprimiert oder in einen anderen Datenstrom transformiert werden, sodass der entwickelte Parsergenerator vielseitig einsetzbar ist.

1.1 Motivation und Hintergrund

Die Erkennung von Mustern in Datenströmen ist eine zentrale Aufgabe in Anwendungsgebieten wie der Analyse von Netzwerkpaketen zur Abwehr von Gefahren [12, 63] oder in der Biomedizin [46, 54, 56]. Dabei werden die zu suchenden Muster häufig als reguläre Ausdrücke spezifiziert und die Suche erfolgt in Echtzeit oder auf großen Datenmengen, sodass ein effizientes Pattern Matching auf Basis endlicher Automaten benötigt wird. Softwarebasierte Lösungen sind in der Regel auf deterministische Automaten ausgelegt und müssen bei großen oder vielen regulären Ausdrücken mit der hohen Anzahl der benötigten Zustände für die Automaten umgehen können. Bei Hardware-basierten Lösungen dagegen kann die Parallelität der Hardwarebausteine für die Umsetzung nichtdeterministischer Automaten genutzt werden, sodass die Determinisierung der Automaten vermieden werden kann.

Pattern Matching auf Basis regulärer Ausdrücke in Hardware wurde bereits 1982 von Floyd und Ullman [20] für *Programmable Logic Arrays* (kurz *PLAs*) beschrieben. Die Umsetzung nichtdeterministischer endlicher Automaten auf FPGAs basiert auf der Arbeit [58] von Sidhu und Prasanna von 2001, in der Basisblöcke für die grundlegenden Operatoren der Konkatenation, Vereinigung und Wiederholung von regulären Ausdrücken vorgestellt werden. In nachfolgenden Arbeiten wie [5, 40, 59, 72, 73] werden weitere Bausteine für reguläre Operatoren und verschiedene Optimierungen für das Pattern Matching präsentiert.

Mit den Optimierungen kann der Durchsatz eines Pattern Matchers gesteigert, Ressourcen auf einem FPGA eingespart oder die parallele Nutzung mehrerer endlicher Automaten für das Pattern Matching verbessert werden. Neben FPGA-basierten Hardwarerealisierungen für das Pattern Matching auf Basis regulärer Ausdrücke existieren auch andere Ansätze wie GPU-basierte Lösungen [6, 68] oder Plattformen wie der *Micron Automata Processor* [16], ein um Logikelemente erweiterter Speicherbaustein.

1.2 Ziele der Arbeit

Das Ziel dieser Arbeit besteht in der Entwicklung eines Parsergenerators für das Pattern Matching in Datenströmen auf FPGAs. Dazu erzeugt der Parsergenerator aus einer Spezifikation von regulären Ausdrücken eine VHDL-Beschreibung eines Pattern Matchers zur Synthese auf FPGAs. Mit der Einbettung von Aktionen und semantischen Bedingungen in die Mustererkennung können dabei u. a. Ausgabeströme erzeugt werden oder semantische Eigenschaften von Datenströmen überprüft werden. Der zu entwickelnde Parsergenerator orientiert sich an dem in [61] vorgestellten Tool *Snowfall*, welches ebenfalls einen Parsergenerator für das Pattern Matching auf FPGAs bereitstellt.

Für die effiziente Nutzung der Hardwareressourcen auf FPGAs werden zur Implementierung von Pattern Matchern im Gegensatz zu Snowfall nichtdeterministische endliche Automaten verwendet. Mit der inhärenten Parallelität der Hardware sind mehrere aktive Zustände und das gleichzeitige Schalten mehrerer Transitionen von nichtdeterministischen Automaten direkt auf FPGAs übertragbar [62]. Zusätzlich soll die Größe der endlichen Automaten für das Pattern Matching durch den zu entwickelnden Parsergenerator unter Berücksichtigung von Verfahren aus der Automatentheorie reduziert und die Abbildung auf Hardware durch Optimierungen für FPGAs verbessert werden, sodass möglichst kompakte Schaltungen auf FPGAs erreicht werden. Dadurch wird die gleichzeitige Nutzung vieler regulärer Ausdrücke für das Pattern Matching auf einem Datenstrom unterstützt.

1.3 Aufbau der Arbeit

Die Arbeit ist wie folgt strukturiert: In Kapitel 2 werden die Grundlagen für das Pattern Matching auf FPGAs erläutert. Anschließend wird in Kapitel 3 ein Überblick über den entwickelten Parsergenerator gegeben und es werden die zentralen Entwurfsentscheidungen erläutert. In Kapitel 4 werden ausgewählte Aspekte der Implementierung des Parsergenerators detaillierter betrachtet und die gewählten Lösungen für die jeweiligen Herausforderungen vorgestellt. Mit Kapitel 5 wird der Parsergenerator anhand einiger Beispiele bzgl. des Ressourcenbedarfs und des Durchsatzes evaluiert und mit den deterministischen Automaten von Snowfall verglichen. Abschließend folgt in Kapitel 6 die Zusammenfassung der Ergebnisse der Arbeit sowie der Ausblick für die zukünftige Weiterentwicklung.

Kapitel 2

Grundlagen

In diesem Kapitel werden die zur Implementierung eines Parsergenerators für das Pattern Matching auf FPGAs erforderlichen Grundlagen vorgestellt. Darunter fallen reguläre Ausdrücke und endliche Automaten, das Parsen von Datenströmen, die Architektur von FPGAs und die Realisierung endlicher Automaten auf FPGAs.

2.1 Reguläre Ausdrücke und endliche Automaten

In Anwendungen wie den Network Intrusion Detection Systemen (kurz NIDS) *Snort* [12, 55] und *Bro* [63] oder bei biomedizinischen Anwendungen [46, 54, 56, 59] werden reguläre Ausdrücke zur Beschreibung von Mustern in Datenströmen eingesetzt. Die Muster werden zur Durchsuchung von Netzwerkpaketen nach potentiellen Bedrohungen oder für die Suche von Fragmenten in Biosequenzen wie der DNA verwendet. Mit der Definition von regulären Ausdrücken zur Beschreibung von Mengen von Strings durch einfache Operatoren wird dabei von der direkten Implementierung eines Pattern Matchers abstrahiert. Die Umwandlung der regulären Ausdrücke in deterministische oder nichtdeterministische endliche Automaten übernehmen Tools zur Implementierung eines Pattern Matchers in Hard- oder Software.

In den folgenden beiden Abschnitten werden reguläre Ausdrücke und endliche Automaten detaillierter betrachtet. Anschließend werden einige der zentralen algorithmischen Verfahren für reguläre Ausdrücke und endliche Automaten vorgestellt.

2.1.1 Reguläre Ausdrücke

Reguläre Ausdrücke bestehen neben den Zeichen eines Alphabets aus Metazeichen für Operatoren und beschreiben Mengen von Strings. Nach der formalen Definition aus [30] enthalten reguläre Ausdrücke nur die drei Operatoren zur Konkatenation, Vereinigung und Wiederholung von Teilausdrücken. In der Praxis unterstützen viele Tools reguläre

Ausdrücke nach dem PCRE-Format[1]. Dabei handelt es sich einerseits um Erweiterungen zur Vereinfachung der Syntax regulärer Ausdrücke, die als *syntaktischer Zucker* bezeichnet werden. Andererseits enthalten PCRE-Ausdrücke ggf. auch Features wie Backreferenzen, die nicht von der formalen Definition der regulären Sprachen abgedeckt werden.

In der Theorie sind reguläre Ausdrücke wie folgt formal definiert:

2.1.1 Definition ([30]). Ein *Alphabet* Σ ist eine endliche, nicht-leere Menge von Symbolen. Eine *Sprache* über einem Alphabet Σ ist eine Teilmenge von $\Sigma^* = \{\sigma_1 \cdots \sigma_n \,|\, \sigma_i \in \Sigma \forall 1 \leq i \leq n, n \in \mathbb{N}\} \cup \{\epsilon\}$ mit dem leeren Wort ϵ. Ein *regulärer Ausdruck über* Σ ist $\emptyset$, das leere Wort ϵ, ein Zeichen $\sigma \in \Sigma$ oder wird durch Anwendung folgender Regel gebildet: Sind α und β reguläre Ausdrücke, dann auch die Vereinigung $\alpha \,|\, \beta$, die Konkatenation $\alpha \circ \beta$ und die Wiederholung α^*. Für einen regulären Ausdruck α ist die *Sprache* $\mathcal{L}(\alpha)$ definiert als $\mathcal{L}(\emptyset) = \emptyset$, $\mathcal{L}(\epsilon) = \{\epsilon\}$ und $\mathcal{L}(\sigma) = \{\sigma\}$. Zudem ist $\mathcal{L}(\alpha \,|\, \beta) = \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$, $\mathcal{L}(\alpha \circ \beta) = \mathcal{L}(\alpha)\mathcal{L}(\beta) = \{w_1 w_2 \,|\, w_1 \in \mathcal{L}(\alpha), w_2 \in \mathcal{L}(\beta)\}$ und $\mathcal{L}(\alpha^*) = \mathcal{L}(\alpha)^*$.

Die regulären Ausdrücke einer Spezifikation für den implementierten Parsergenerator unterstützen eine Teilmenge der Operatoren des PCRE-Formats. Darüber hinaus werden nicht in der PCRE-Syntax enthaltene Operatoren wie die Negation für komplette reguläre Ausdrücke unterstützt. Die implementierten Operatoren für reguläre Ausdrücke des Parsergenerators sind in Tabelle 2.1 aufgelistet. Dabei wird für das Alphabet Σ ein erweiterter ASCII-Zeichensatz (mit 8-Bit Repräsentation der Symbole) angenommen.

Die Verwendung von regulären Ausdrücken zur Beschreibung von Mengen von Strings wird anhand eines Beispiels veranschaulicht.

2.1.2 Beispiel. Betrachtet werden die folgenden beiden regulären Ausdrücke

$$\alpha_1 = <[\hat{}>\sqcup]^*[a\text{-}z]\{2\}\ (\sqcup[a\text{-}z]^+ = [0\text{-}9]^*)^*> \text{ und}$$
$$\alpha_2 = [a\text{-}zA\text{-}Z][a\text{-}zA\text{-}Z0\text{-}9]^* \ - \ (int \,|\, return)$$

Der Ausdruck α_1 beschreibt öffnende XML-Tags, deren Tagname mit zwei Kleinbuchstaben endet und die eine beliebige Anzahl numerischer Attribute aufweisen. Der reguläre Ausdruck α_2 beschreibt Identifier einer fiktiven Programmiersprache, die nur den Datentyp `int` unterstützt. Die Bezeichner beginnen mit einem Buchstaben, gefolgt von einer beliebigen Anzahl an Buchstaben oder Ziffern und dürfen nicht den beiden Schlüsselwörtern `int` oder `return` entsprechen.

Mit regulären Ausdrücken können Anwender von einer konkreten Implementierung abstrahiert Mengen von Strings für das Pattern Matching beschreiben. Allerdings lassen sich reguläre Ausdrücke nicht direkt in Hard- oder Software realisieren, sodass sie zunächst in endliche Automaten umgewandelt und in dieser Form für das Pattern Matching verwendet werden. Im folgenden Abschnitt werden daher endliche Automaten genauer betrachtet.

[1] Perl-Compatible Regular Expressions, `http://www.pcre.org`

Tabelle 2.1: Ausgewählte Operatoren für reguläre Ausdrücke (PCRE-Syntax [26], Ragel [65])

Feature	Beschreibung
a	Zeichen, das eine einzelne Instanz von sich selbst matcht
.	Wildcard, welche alle Zeichen des Alphabets matcht
$[abc]$	Zeichenklasse, die alle Zeichen innerhalb der Klammern matcht
$[a\text{-}z]$	Zeichenklasse, die alle Zeichen des Bereichs matcht
$[\hat{}abc]$	Invertierte Zeichenklasse, die alle nicht angegebenen Zeichen matcht
α^*	Matcht null- oder mehrmalige Wiederholung von α
α^+	Matcht ein- oder mehrmalige Wiederholung von α
$\alpha?$	Matcht null- oder einmaliges Vorkommen von α
$\alpha\{n\}$	Matcht n-malige Wiederholung von α
$\alpha\{n,m\}$	Matcht zwischen n- und m-malige Wiederholung von α
$\alpha\{n,\}$	Matcht mindestens n-malige Wiederholung von α
$\alpha\beta$	Matcht Konkatenation von α gefolgt von β
$\alpha \mid \beta$	Matcht α oder β
$\alpha - \beta$	Matcht α, wenn gleichzeitig β nicht matcht
α -- β	Matcht α, wenn gleichzeitig kein Präfix von β matcht
$!\alpha$	Matcht alles außer α
(α)	Gruppierung von α (z. B. für die Anwendung von Operatoren)

2.1.2 Endliche Automaten

Die regulären Ausdrücke werden für das Pattern Matching auf Datenströmen in endliche Automaten zur Realisierung in Hard- oder Software umgewandelt. Dabei stehen mit deterministischen und nichtdeterministischen endlichen Automaten zwei Formen der Automaten zur Verfügung. Beide Varianten werden im Folgenden detaillierter betrachtet.

Formal sind deterministische und nichtdeterministische endliche Automaten wie folgt definiert:

2.1.3 Definition ([30]). Ein *nichtdeterministischer endlicher Automat* (kurz *NFA*) $\mathcal{A}$ ist definiert als ein Tupel $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ mit einer endliche Menge Q von Zuständen, einem Alphabet Σ, einer Transitionsfunktion $\delta : Q \times \Sigma \to 2^Q$, einem Startzustand $s \in Q$ und einer Menge $F \subseteq Q$ akzeptierender Zustände. Dabei bezeichne 2^Q die Potenzmenge der Menge Q, d. h. $2^Q = \{X \mid X \subseteq Q\}$.

Ein *deterministischer endlicher Automat* (kurz *DFA*) $\mathcal{A}$ ist ein NFA $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ mit $|\delta(q, \sigma)| = 1$ für alle Zustände $q \in Q$ und alle Symbole $\sigma \in \Sigma$.

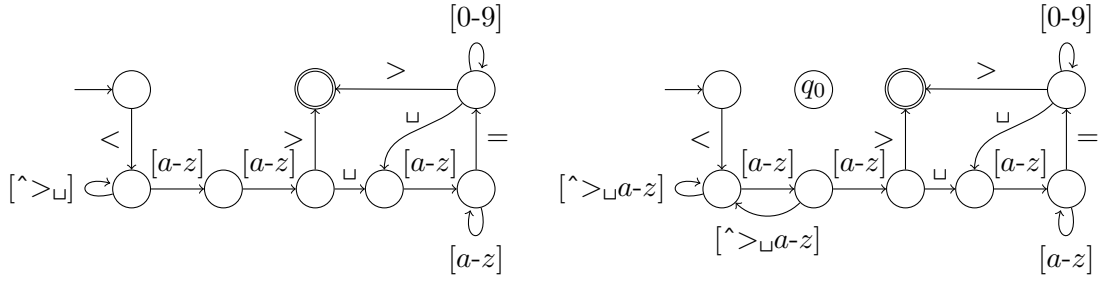
Der Unterschied zwischen deterministischen und nichtdeterministischen endlichen Automaten liegt somit allein in der Transitionsfunktion. In einem DFA ist zu jedem Zeitpunkt

der Mustererkennung genau ein Zustand aktiv. Nachdem dies zu Beginn der Startzustand s ist, ist nach dem Lesen eines Zeichens σ_1 der Folgezustand $\delta(s, \sigma_1)$ aktiv und nach dem folgenden Zeichen σ_2 der Zustand $\delta(\delta(s, \sigma_1), \sigma_2)$. Diese Eigenschaft gilt während der gesamten Verarbeitung eines Datenstroms. Dadurch sind deterministische endliche Automaten besonders für die Realisierung in Software geeignet, da jeweils nur der aktive Zustand und das nächste Zeichen des Datenstroms zur Bestimmung des Folgezustandes betrachtet werden müssen [62].

In einem NFA dagegen können zu jedem Zeitpunkt mehrere Zustände aktiv sein. Nach dem Beginn der Verarbeitung beim Startzustand s kann ein NFA durch das Verarbeiten eines Zeichens σ_1 in mehrere Folgezustände $q_1, \ldots, q_n$ übergehen. Für das folgende Zeichen σ_2 des Datenstroms müssen dann für die aktiven Zustände $q_1, \ldots, q_n$ die ausgehenden Transitionen für σ_2 ausgewertet werden. Somit sind NFAs zur Implementierung in Software weniger geeignet, da stets mit Mengen von Zuständen operiert werden muss, die iterativ zu verarbeiten sind. Dagegen ist die Realisierung von NFAs auf Hardware durch die inhärente Parallelität der Hardwarebausteine direkt übertragbar, da mehrere Zustände gleichzeitig aktiv sein können und die Transitionen parallel berechenbar sind [62].

Die beiden Varianten endlicher Automaten werden anhand eines Beispiels betrachtet.

2.1.4 Beispiel. Für den regulären Ausdruck α_1 aus Beispiel 2.1.2 ist in Abbildung 2.1(a) ein nichtdeterministischer Automat angegeben. In Abbildung 2.1(b) befindet sich ein DFA für α_1 Ausdruck, wobei alle nicht eingezeichneten Transitionen in den Fehlerzustand q_0 führen. Somit benötigt der DFA einen Zustand (nämlich q_0) und zehn Transitionen mehr



(a) NFA für α_1

(b) DFA für α_1

Abbildung 2.1: Beispiel: Endliche Automaten für $\alpha_1 = <[\hat{}>␣]^*[a\text{-}z]\{2\}(␣[a\text{-}z]^+ = [0\text{-}9]^*)^*>$

als der NFA. Darunter fällt die zusätzliche Transition, um die beiden Kleinbuchstaben am Ende des Tagnamen korrekt erkennen zu können - diese ist im NFA nicht erforderlich.

Im weiteren Verlauf der Arbeit werden verschiedene formale Sprachen für endliche Automaten verwendet, die nun definiert werden.

2.1.5 Definition ([30, 31]). Sei $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ ein NFA. Die *erweiterte Transitionsfunktion* $\delta^* : Q \times \Sigma^* \to 2^Q$ von $\mathcal{A}$ ist induktiv definiert als

$$\delta^*(q, \epsilon) = \{q\} \text{ und}$$
$$\delta^*(q, w\sigma) = \delta(\delta^*(q, w), \sigma)$$

wobei $w \in \Sigma^*$ ein Wort, $\sigma \in \Sigma$ ein Zeichen und $\delta(S, \sigma) = \bigcup_{q \in S} \delta(q, \sigma)$ für $S \subseteq Q$ ist. Die *Menge der von $\mathcal{A}$ akzeptierten Wörter* ist definiert als die Sprache $\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \delta^*(s, w) \cap F \neq \emptyset\}$. Für einen Zustand $q \in Q$ von $\mathcal{A}$ definiert $\mathcal{L}_L(\mathcal{A}, q) = \{w \in \Sigma^* \mid q \in \delta^*(s, w)\}$ die *linksseitige Sprache* und $\mathcal{L}_R(\mathcal{A}, q) = \{w \in \Sigma^* \mid \delta^*(q, w) \cap F \neq \emptyset\}$ die *rechtsseitige Sprache*[2] von q.

Die Sprache $\mathcal{L}(\mathcal{A})$ eines Automaten $\mathcal{A}$ enthält alle Wörter, die der Automat akzeptiert. Für einen Zustand q enthält die Sprache $\mathcal{L}_L(\mathcal{A}, q)$ die Wörter, die der Automat akzeptiert, wenn q alleiniger akzeptierender Zustand ist, d. h. $\mathcal{L}_L(\mathcal{A}, q) = \mathcal{L}((Q, \Sigma, \delta, s, \{q\}))$. Dagegen enthält die Sprache $\mathcal{L}_R(\mathcal{A}, q)$ die Menge der Wörter, die $\mathcal{A}$ akzeptiert, wenn q der Startzustand von $\mathcal{A}$ ist, d. h. $\mathcal{L}_R(\mathcal{A}, q) = \mathcal{L}((Q, \Sigma, \delta, q, F))$.

Deterministische und nichtdeterministische endliche Automaten weisen die gleiche Ausdrucksstärke auf, d. h. für jeden NFA $\mathcal{A}$ gibt es einen DFA $\mathcal{B}$ mit $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$ und umgekehrt [30]. Allerdings können sich NFA und DFA in der Anzahl der benötigten Zustände unterscheiden. Die Umwandlung von nichtdeterministischen endlichen Automaten in DFAs ist mit der Teilmengenkonstruktion [1, 30] möglich. Dabei kann durch die *Zustandsexplosion* die Anzahl der Zustände exponentiell steigen. Beispielsweise besitzt der reguläre Ausdruck $(0 \mid 1)^*1(0 \mid 1)\{n-1\}$ für jedes $n \in \mathbb{N}$ einen NFA mit $n+1$ Zuständen, wogegen jeder DFA für diesen Ausdruck mindestens 2^n Zustände benötigt [30]. Auch in der Praxis treten Unterschiede bei der Anzahl der Zustände von DFAs und NFAs auf. In Abbildung 2.2 ist beispielhaft die Anzahl der Zustände eines minimalen DFAs und eines (nicht minimierten) NFAs für den Snort-Regelsatz Attack-Response angegeben. Der Unterschied schlägt sich auch beim Ressourcenbedarf auf FPGAs in Form von Lookup-Tabellen und Registern wieder. Dieser ist für den Snort-Regelsatz beim Pattern Matching mit UDP-Kommunikation ebenfalls in Abbildung 2.2 aufgeführt (weitere Details folgen in Kapitel 5).

Zur Implementierung in Hard- oder Software werden reguläre Ausdrücke in endliche Automaten umgewandelt. Die Umwandlung in NFAs kann u. a. mit den Verfahren von Thompson [64] oder McNaughton und Yamada [43] sowie Glushkov [22] erfolgen. Mit dem Konstruktionsverfahren von Thompson wird ein regulärer Ausdruck α in einen NFA $\mathcal{A}$ mit Epsilon-Transitionen mit $\mathcal{L}(\alpha) = \mathcal{L}(\mathcal{A})$ umgewandelt. Dabei ist ein *NFA mit Epsilon-Transitionen* (kurz *ϵ-NFA*) ein NFA mit einer Transitionsfunktion $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$,

[2] In der englischsprachigen Literatur wird die linksseitige Sprache aus als *Pre-Language* und die rechtsseitige Sprache als *Post-Language* bezeichnet [42].

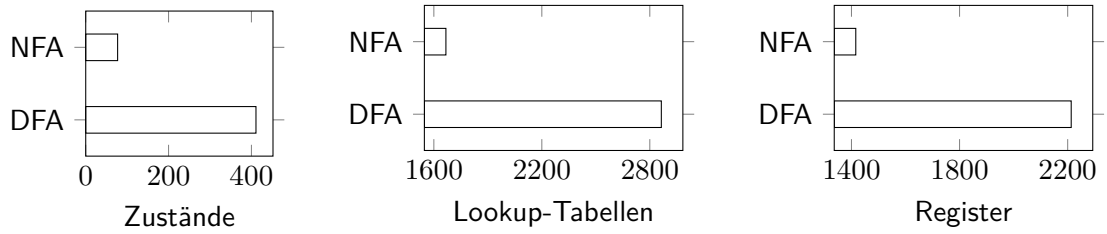
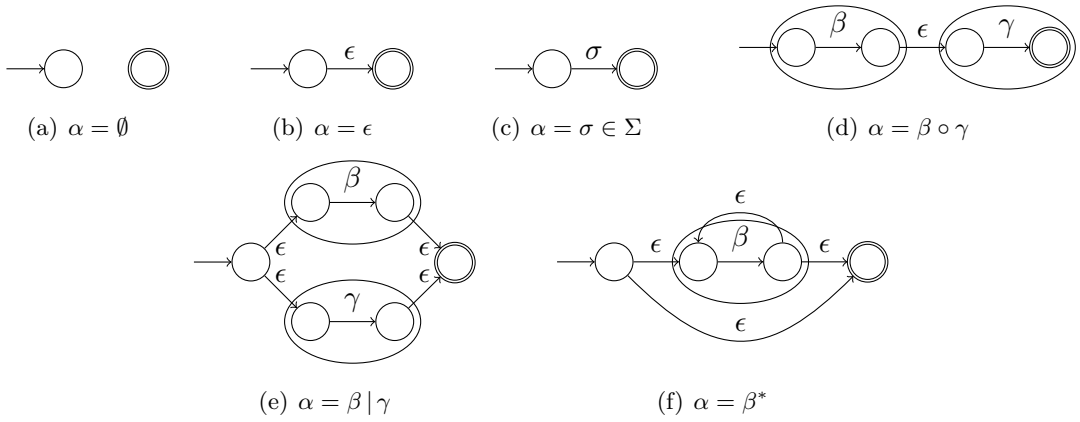


Abbildung 2.2: Beispiel: Vergleich von NFA und DFA für den Snort-Regelsatz Attack-Response

welche auch Zustandsübergänge ohne das Lesen von Zeichen ermöglicht. Die Regeln zur Konstruktion des ϵ-NFAs $\mathcal{A}$ für α sind in Abbildung 2.3 grafisch dargestellt. Das Verfahren ist auf die in der Theorie verwendeten regulären Ausdrücke aus Definition 2.1.1 beschränkt.



(a) $\alpha = \emptyset$ (b) $\alpha = \epsilon$ (c) $\alpha = \sigma \in \Sigma$ (d) $\alpha = \beta \circ \gamma$

(e) $\alpha = \beta \,|\, \gamma$ (f) $\alpha = \beta^*$

Abbildung 2.3: Umwandlung des regulären Ausdrucks α in einen ϵ-NFA nach Thompson [1, 64]

Im nächsten Abschnitt werden einige Eigenschaften und weitere algorithmische Verfahren für reguläre Ausdrücke und endliche Automaten vorgestellt, die zur Implementierung eines Parsergenerators für das Pattern Matching auf FPGAs eingesetzt werden können.

2.1.3 Algorithmische Verfahren für endliche Automaten

Das in Abschnitt 2.1.2 skizzierte Verfahren zur Umwandlung von regulären Ausdrücken in endliche Automaten deckt nur die in Definition 2.1.1 vorkommenden Operatoren für reguläre Ausdrücke ab. Für erweiterte Operatoren wie die Komplementierung oder den Schnitt fließen die Abschlusseigenschaften regulärer Sprachen mit in die Umwandlung ein. Außerdem sind durch das Verfahren von Thompson gewonnene Automaten bzgl. der Anzahl der benötigten Zustände und Transitionen im Allgemeinen nicht minimal. Somit werden Verfahren zur Reduzierung der Größe von Automaten betrachtet, um z. B. den Ressourcenbedarf für Automaten auf FPGAs möglichst gering zu halten.

Komplement und Schnitt Die Komplementierung eines deterministischen Automaten ist durch das Vertauschen von akzeptierenden und nicht akzeptierenden Zuständen algorithmisch sehr einfach möglich [30]. Bei der Komplementierung von NFAs ist das Vertauschen akzeptierender und nicht akzeptierender Zustände dagegen nicht ausreichend [28]. Daher wird ein NFA für die Komplementierung erst in einen deterministischen Automaten umgewandelt, welcher dann komplementiert wird. Durch die Determinisierung des NFAs lässt sich eine mögliche exponentielle Automatengröße nicht vermeiden - diese Eigenschaft gilt für alle Verfahren zur Komplementierung von NFAs. Beispielsweise besitzt der reguläre Ausdruck $\alpha = (0 \mid 1)^*0(0 \mid 1)\{n\}1(0 \mid 1)^*$ für jedes $n \in \mathbb{N}$ einen NFA $\mathcal{A}$ mit $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\alpha)$ und $n+3$ Zuständen; jeder NFA $\mathcal{B}$ mit $\mathcal{L}(\mathcal{B}) = \Sigma^* - \mathcal{L}(\alpha)$ benötigt dagegen mindestens 2^{n-2} Zustände [27].

Für den Schnitt von zwei regulären Ausdrücken α_1 und α_2 kann der Produktautomat verwendet werden. Dieser kann sowohl für NFAs als auch für DFAs gebildet werden [30]. Dazu bezeichne $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ einen NFA für α_1 und $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, s_1, F_2)$ einen NFA für α_2. Der Produktautomat für den Schnitt ist $\mathcal{A}_{prod} = (Q_1 \times Q_2, \Sigma, \delta, (s_1, s_2), F_1 \times F_2)$ mit $\delta((q_1, q_2), \sigma) = \delta_1(q_1, \sigma) \times \delta_2(q_2, \sigma)$ für alle $q_1 \in Q_1, q_2 \in Q_2$ und $\sigma \in \Sigma$.

Für reguläre Ausdrücke mit den Differenzoperatoren aus Tabelle 2.1 können die Techniken für Komplement und Schnitt zu einem Umwandlungsverfahren kombiniert werden.

Reduzierung der Automatengröße Die Reduzierung der Größe von endlichen Automaten für das Pattern Matching trägt zur Einsparung von Ressourcen auf dem FPGA bei und könnte die Effizienz der Implementierung steigern. Während die Minimierung[3] eines DFA mit n Zuständen nach dem Verfahren von Hopcroft [29] in $\mathcal{O}(|\Sigma| \times n \log n)$ Zeit möglich ist, ist die Minimierung von NFAs ein algorithmisch schwierigeres Problem und erfordert mehr Laufzeit. In [35] wird gezeigt, dass die Minimierung von nichtdeterministischen endlichen Automaten im Allgemeinen PSPACE-vollständig ist und damit im schlechtesten Fall exponentielle Laufzeit benötigt. Selbst für eingeschränkte Erweiterungen von DFAs in Richtung Nichtdeterminismus (wie z. B. mehrere Startzustände oder eine beschränkte Anzahl an Folgezuständen auf jedem Pfad für alle akzeptierten Wörter) ist die Minimierung im Allgemeinen NP-vollständig [41] und damit nicht effizient berechenbar (sofern P $\neq$ NP gilt). Somit weisen die Verfahren zur Minimierung von NFAs u. a. von Kameda und Weiner [36] oder Melnikov [44, 45] hohe Laufzeiten auf und sind in der Praxis in vielen Tools nicht enthalten [67]. Neben der exakten Berechnung des Minimalautomaten für einen NFA ist im Allgemeinen auch eine Approximation des Minimalautomaten mit einem konstanten Faktor nicht in polynomieller Laufzeit möglich: Für einen NFA mit n Zuständen ist die Berechnung einer $o(n)$-Approximation des minimalen Automaten PSPACE-vollständig, wenn für die Alphabetsgröße $|\Sigma| \geq 2$ gilt [23, 24].

[3] Ein *Minimalautomat* für eine reguläre Sprache L ist ein NFA (DFA) $\mathcal{A} = (Q_1, \Sigma, \delta_1, s_1, F_1)$ mit $\mathcal{L}(\mathcal{A}) = L$, sodass für alle NFAs (DFAs) $\mathcal{B} = (Q_2, \Sigma, \delta_2, s_2, F_2)$ mit $\mathcal{L}(\mathcal{B}) = L$ gilt, dass $|Q_1| \leq |Q_2|$ ist.

Neben der Minimierung von NFAs können Heuristiken [66] und Verfahren zur Reduzierung der Größe von NFAs [7, 33] die Anzahl von Zuständen verringern und besitzen bessere Laufzeiten, liefern im Allgemeinen aber keinen minimalen NFA. Mit den geringeren Laufzeiten sind diese Verfahren in der Praxis häufig auch für größere Automaten einsetzbar. Im Folgenden wird mit der *Äquivalenz-basierten Reduzierung der Automatengröße* [33] ein solches Verfahren genauer vorgestellt. Wie bei der Minimierung von DFAs werden auch bei der Äquivalenz-basierten Reduzierung die Zustände des Automaten zusammengefasst, die nicht unterscheidbar sind. Zwei Zustände p und q eines Automaten $\mathcal{A}$ sind genau dann *ununterscheidbar*, wenn $\mathcal{L}_R(\mathcal{A}, p) = \mathcal{L}_R(\mathcal{A}, q)$ gilt. In DFAs beginnt die Bestimmung solcher Zustände mit der Unterscheidung von akzeptierenden und nicht akzeptierenden Zuständen. Induktiv werden zwei Zustände unterscheidbar, wenn sie eine Transition mit gleichem Symbol zu unterscheidbaren Zuständen besitzen. In NFAs dagegen müssen Mengen aktiver Zustände betrachtet werden, sodass nicht direkt klar ist, wie entschieden werden kann, ob zwei Zustände unterscheidbar sind. Die Bestimmung unterscheidbarer Zustände in NFAs ist nur mit erheblichen Aufwand möglich. Bei der Äquivalenz-basierten Reduzierung wird daher eine Teilmenge der ununterscheidbaren Zustände berechnet, welche den Äquivalenzklassen einer zu ermittelnden Relation $\equiv_R$ entsprechen. Dadurch werden niemals zwei Zustände zusammengefasst, die unterscheidbar sind. Allerdings kann es weiterhin Zustände geben, die zusammengefasst werden könnten, deren Bestimmung aber sehr aufwändig wäre. Zwei Zustände, die von der Äquivalenz-basierten Reduzierung als ununterscheidbar angesehen werden, werden als *äquivalent* bezeichnet. Damit gilt: Äquivalente Zustände sind ununterscheidbar, aber nicht äquivalente Zustände müssen nicht notwendigerweise unterscheidbar sein [33, 34].

Die Ununterscheidbarkeit von zwei Zuständen kann symmetrisch zur Definition über die rechtsseitige Sprache $\mathcal{L}_R(.)$ auch über die linksseitige Sprache $\mathcal{L}_L(.)$ definiert werden. Dazu wird in dem Verfahren analog eine Teilmenge von Zuständen berechnet, die bzgl. der linksseitigen Sprache ununterscheidbar ist. Diese berechnete Teilmenge entspricht den Äquivalenzklassen einer zu $\equiv_R$ symmetrischen Relation $\equiv_L$. Die Kombination der beiden Varianten zur Ununterscheidbarkeit kann bei der Reduzierung der Automatengröße zu verbesserten Ergebnissen führen [33, 34] - dies zeigen auch Experimente mit zufällig erzeugten regulären Ausdrücken [38].

Die beiden Äquivalenzrelationen $\equiv_L$ und $\equiv_R$ sind formal wie folgt definiert:

2.1.6 Definition ([31, 32]). Sei $\mathcal{A} = (Q, \Sigma, \delta, S, F)$[4] ein NFA und $\mathcal{A}^r = (Q, \Sigma, \delta^r, F, S)$ der reverse Automat mit $q \in \delta^r(p, \sigma)$ genau dann, wenn $p \in \delta(q, \sigma)$ gilt. Die Relation

[4] In dieser Definition werden NFAs mit mehreren Startzuständen verwendet, damit der reverse Automat $\mathcal{A}^r$ mit $\mathcal{L}(\mathcal{A}^r) = \{w^r \,|w \in \mathcal{L}(\mathcal{A})\}$ mit $\epsilon^r = \epsilon$ und $(\sigma_1 \cdots \sigma_n)^r = \sigma_n \cdots \sigma_1 \forall \sigma_1 \cdots \sigma_n \in \Sigma^* - \{\epsilon\}$ einfacher zu definieren ist.

$\equiv_R$ ist definiert als die gröbste Äquivalenzrelation[5] über Q, für die folgende Bedingungen gelten:

$$\equiv_R \cap (F \times (Q - F)) = \emptyset \tag{B_1}$$

$$\forall p, q \in Q \forall \sigma \in \Sigma : (p \equiv_R q \Rightarrow \forall q' \in \delta(q, \sigma) \exists p' \in \delta(p, \sigma) : q' \equiv_R p') \tag{B_2}$$

Die Äquivalenzrelation $\equiv_L$ entspricht der Relation $\equiv_R$ auf dem Automaten $\mathcal{A}^r$.

Wenn zwei Zustände $p, q \in Q$ in Relation $\equiv_R$ zueinander stehen ($p \equiv_R q$), dann ist deren rechtsseitige Sprache gleich, d. h. es gilt $\mathcal{L}_R(\mathcal{A}, p) = \mathcal{L}_R(\mathcal{A}, q)$ gilt. Analog gilt dies für die Relation $\equiv_L$ und die linksseitige Sprache der Zustände. Die Berechnung der Relationen $\equiv_R$ und $\equiv_L$ ist mit Algorithmus 2.1 möglich. Das Verfahren besitzt eine polynomielle Laufzeit, sodass es in der Praxis auch für größere Automaten eingesetzt werden kann. Eine effizientere Möglichkeit zur Berechnung der Äquivalenzrelationen für einen NFA wird

Eingabe: NFA $\mathcal{A} = (Q, \Sigma, \delta, S, F)$
Ausgabe: Äquivalenzrelation $\equiv_R$ für $\mathcal{A}$
$Q^\emptyset \leftarrow Q \cup \{\emptyset\}$; $\delta^\emptyset \leftarrow \delta \cup \{(p, \sigma, \emptyset) \mid \delta(p, \sigma) = \emptyset\} \cup \{(\emptyset, \sigma, \emptyset) \mid \sigma \in \Sigma\}$
$\not\equiv_R \leftarrow \emptyset$
for all $(p, q) \in (Q^\emptyset - F) \times F$ **do**
  $\not\equiv_R \leftarrow \not\equiv_R \cup \{(p, q), (q, p)\}$
end for
while $\exists p, q \in Q^\emptyset \exists \sigma \in \Sigma \exists r \in \delta^\emptyset(p, \sigma) \forall s \in \delta^\emptyset(q, \sigma) : r \not\equiv_R s$ **do**
  wähle ein solches Paar (p, q); $\not\equiv_R \leftarrow \not\equiv_R \cup \{(p, q), (q, p)\}$
end while
return $\equiv_R \leftarrow ((Q^\emptyset \times Q^\emptyset) - \not\equiv_R) - \{(\emptyset, \emptyset)\}$

Algorithmus 2.1: Berechnung der Äquivalenzrelation $\equiv_R$ [33]

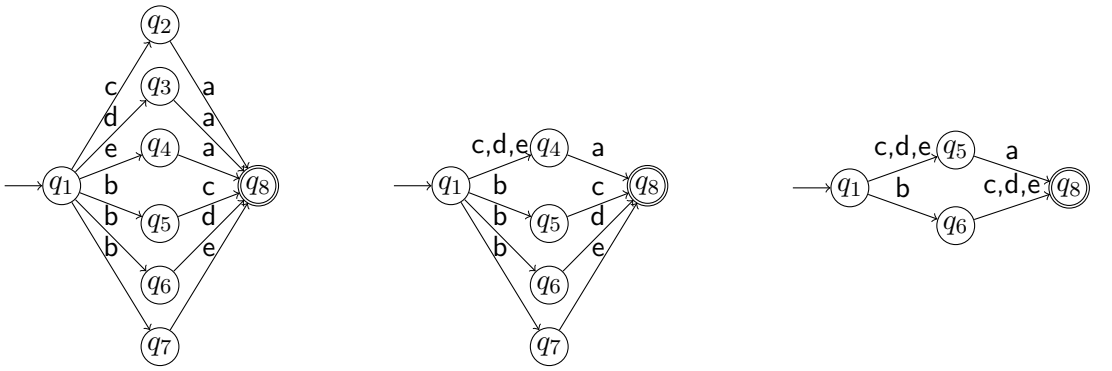
in [31] vorgestellt (siehe Anhang B) und benötigt für einen NFA mit n Zuständen und m Transitionen $\mathcal{O}(|\Sigma| \times m \log n)$ Zeit.

Die Berechnung der Äquivalenzrelationen $\equiv_L$ und $\equiv_R$ für einen NFA $\mathcal{A}$ liefert durch die Äquivalenzklassen zwei Partitionen Π_L und Π_R der Zustandsmenge Q. Das Verfahren fasst die Zustände dann anhand der Äquivalenzklassen zur Reduzierung von $\mathcal{A}$ zusammen [31, 32]. Für ein bzgl. $\equiv_L$ und $\equiv_R$ optimales Zusammenfassen müssen im Allgemeinen beide Partitionen miteinander kombiniert werden, wie das folgende Beispiel zeigt.

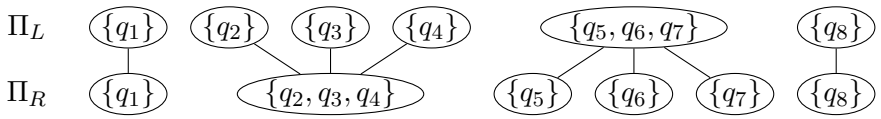
2.1.7 Beispiel. Für den NFA aus Abbildung 2.4(a) ist der nach $\equiv_R$ reduzierte NFA in Abbildung 2.4(b) angegeben. Dabei können lediglich drei Zustände des NFA zu einem

[5] Eine Äquivalenzrelation A über einer Menge U verfeinert eine Relation $B \subseteq U \times U$, wenn für alle $x, y \in U$ gilt: $(x, y) \in A \Rightarrow (y, x) \in B$. In diesem Fall ist B gröber als A.

Zustand zusammengefasst werden. Ähnlich verhält es sich, wenn nur die Relation $\equiv_L$ betrachtet wird. Der nach $\equiv_L$ und $\equiv_R$ reduzierte NFA ist in Abbildung 2.4(c) angegeben. Mit der Kombination beider Relationen können weitere drei Zustände des NFA zusammengefasst werden. Somit wird anhand des Beispiels ersichtlich, dass für eine optimale Reduzierung der Zustandsanzahl die beiden Partitionen Π_L und Π_R zu kombinieren sind.



(a) NFA $\mathcal{A}$ (b) Reduzierung nach $\equiv_R$ (c) Reduzierung nach $\equiv_L$ und $\equiv_R$



(d) Bipartiter Graph G_B für $\mathcal{A}$

Abbildung 2.4: Beispiel: Äquivalenz-basierte Reduzierung eines NFAs [32]

Eine optimale Lösung zum Zusammenfassen der Zustände unter Berücksichtigung von Π_L und Π_R kann mit einem Algorithmus für das Set-Covering-Problem[6] mit der Eingabe $(Q, \Pi_L \cup \Pi_R)$ berechnet werden [32]. Da das Set-Covering-Problem im Allgemeinen jedoch NP-vollständig ist [37], existiert kein effizienter Algorithmus für das Problem. Jeder Zustand q von $\mathcal{A}$ ist in genau zwei Mengen aus $\Pi_L \cup \Pi_R$ enthalten, sodass sich die Eingabe auch als bipartiter Graph[7] $G_B = (V, E)$ mit $V = L \cup R$ darstellen lässt. Die Blöcke X_i aus $\Pi_L \cup \Pi_R$ bilden die Knoten des Graphen, wobei alle X_i aus Π_L zu L zugeordnet werden und die X_i aus Π_R zu R. Die Kantenmenge E wird durch die Zustände des Automaten bestimmt, indem für jeden Zustand $q \in Q$ eine Kante zwischen den Mengen X_i und X_j eingefügt wird, die q enthalten (siehe Abbildung 2.4(d) für den Graphen G_B des NFAs aus Beispiel 2.1.7). Die gesuchte minimale Lösung für das Set-Covering-Problem

[6]Set-Covering-Problem [37]: Gibt es zu einer Menge U, n Teilmengen $S_1, \ldots, S_n$ von U und einer Zahl $k \leq n$ eine Auswahl von höchstens k Teilmengen $S_{i_1}, \ldots, S_{i_k}$, deren Vereinigung U entspricht?

[7]Ein ungerichteter Graph $G = (V, E)$ ohne Schleifen heißt *bipartit*, wenn sich seine Knotenmenge V in zwei disjunkte Teilmengen V_1 und V_2 unterteilen lässt, sodass zwischen den Knoten innerhalb der Teilmengen keine Kanten verlaufen [14].

korrespondiert zu einem minimalen Vertex-Cover[8] für G_B. Für bipartite Graphen kann eine minimale Lösung des Vertex-Covering-Problems nach dem Satz von König [4] über ein maximales Matching effizient berechnet werden. Ein Matching M ist eine Teilmenge der Kanten E, sodass für jeden Knoten $v \in V$ des Graphen höchstens eine der Kanten aus M inzident zu v ist. Ein maximales Matching ist ein Matching M mit maximaler Kardinalität, d. h. für alle Matchings $M^{'}$ ist $|M| \geq |M^{'}|$ [14]. Maximale Matchings lassen sich nach dem Algorithmus von Hopcroft und Karp in $\mathcal{O}(|E|\sqrt{|V|})$ Zeit für den Graphen G_B berechnen. In diesem Fall ist $|E| = |Q|$ und $|V| \leq 2|Q|$, sodass die Berechnung des maximalen Matchings in $\mathcal{O}(n^{\frac{3}{2}})$ Zeit für einen NFA mit n Zuständen möglich ist. Ein minimales Vertex-Cover U kann aus dem maximalen Matching ebenfalls in polynomieller Zeit berechnet werden [32].

Mit Hilfe von U können die Zustände von $\mathcal{A}$ zusammengefasst werden: Alle Zustände der zu einem Knoten $v_i \in U$ gehörenden Zustandsmenge $X_i \subseteq Q$ werden zu einem Zustand aus X_i verschmolzen. Falls ein Zustand in zwei Mengen von U enthalten, ist er vorher aus einer der beiden Mengen zu entfernen. Das Ergebnis des Verfahrens ist der *Äquivalenz-reduzierte* NFA. Bezüglich der Güte dieses Verfahren zur Reduzierung der Automatengröße ist anzumerken, dass der Äquivalenz-reduzierte NFA lediglich bzgl. der Äquivalenzklassen aus Π_L und Π_R minimal ist. Bislang offen ist u. a. die Frage, ob durch Zusammenfassen von einer Teilmenge äquivalenter Zustände aus $\Pi_L \cup \Pi_R$ und anschließender Nutzung der vollständigen Äquivalenzklassen $\Pi_L^{'}$ und $\Pi_R^{'}$ des entstandenen NFA bessere Ergebnisse erzielbar sind. Auch eine wiederholte Anwendung des zuvor beschriebenen Verfahren liefert noch keine Antwort auf diese Frage [32].

Bei der Minimierung von endlichen Automaten ist zu beachten, dass eine minimale Anzahl an Zuständen nicht immer mit einer minimalen Anzahl an Transitionen korrespondieren muss: Bei DFAs besitzt der Minimalautomat (mit minimaler Anzahl an Zuständen) zugleich auch die minimale Anzahl an Transitionen. Bei nichtdeterministischen Automaten dagegen kann ein Minimalautomat mehr Transitionen besitzen als ein NFA mit einer (minimal) größeren Zustandsmenge. So gibt es reguläre Sprachen, bei denen mit einer um eins vergrößerten Zustandsmenge die Anzahl der Transitionen in einem NFA von m auf $\sqrt{m}$ gesenkt werden kann [17, 57].

Mit der Vorstellung ausgewählter algorithmischer Verfahren ist die Betrachtung von regulären Ausdrücken und endlichen Automaten zunächst abgeschlossen. Im weiteren Verlauf der Arbeit wird auf diese Grundlagen immer wieder zurückgegriffen. Im nächsten Abschnitt wird das Parsing erläutert, mit dem u. a. eine Spezifikation mit regulären Ausdrücken durch den Parsergenerator eingelesen werden kann. Parsing kann aber auch in Hardware wie FPGAs für das Pattern Matching mit kontextfreien Sprachen in Datenströmen genutzt werden.

[8] Vertex-Cover-Problem [37]: Gibt es für einen Graphen G und eine Zahl k eine Teilmenge U mit maximal k Knoten, sodass jede Kante von G zu mindestens einem Knoten aus U inzident ist?

2.2 Parsing

Mit *Parsing* wird die Aufteilung eines Datenstroms in zusammenhängende Teile und der parallele Aufbau einer grammatikalischen Struktur bezeichnet [1]. Parsing kommt u. a. bei der Implementierung von Compilern zum Einsatz, wird aber auch zum Einlesen einer Spezifikation des realisierten Parsergenerators benötigt.

Das Parsen von Datenströmen erfolgt typischerweise in zwei Schritten [1]: Im ersten Schritt werden in der *lexikalischen Analyse* die Eingabesymbole des Zeichenstroms zu größeren Einheiten (wie z. B. Identifiern) zusammengefasst, die als *Token* bezeichnet werden. Das Ergebnis der lexikalischen Analyse ist dann ein Tokenstrom, welcher im zweiten Schritt, der *Syntaxanalyse*, durch Überprüfung der Wohlgeformtheit gegenüber einer kontextfreien Grammatik analysiert wird. Für wohlgeformte Tokenströme erzeugt die Syntaxanalyse einen *Parsebaum* zur Repräsentation der Struktur des Stroms bzgl. der Grammatik, der für die weitere Verarbeitung wie die Codegenerierung genutzt werden kann. Der Ablauf beim Parsen von Datenströmen ist in Abbildung 2.5 grafisch dargestellt.

Abbildung 2.5: Ablauf beim Parsen von Datenströmen (nach [1])

Die lexikalische Analyse auf dem Zeichenstrom erfolgt mit Hilfe von regulären Sprachen. Die einzelnen Eingabezeichen werden anhand von regulären Ausdrücken zu Token gruppiert und bilden fortan eine Einheit. Ein Token besitzt einen Typ in Form eines abstrakten Bezeichners (wie z. B. der Bezeichner *ID* für Identifier) und einem Zeiger auf einen Tabelleneintrag mit weiteren Details. Der Typ des Tokens wird zur Repräsentation der lexikalischen Einheit in der Syntaxanalyse verwendet. Im Tabelleneintrag des Tokens werden Informationen wie der zum Token gehörende String im Zeichenstrom abgespeichert (wie z. B. der Name $x25$ eines Identifiers). Die in der lexikalischen Analyse eingesetzten Tools werden als *Lexer* bezeichnet und erzeugen als Ausgabe einen Tokenstrom [1].

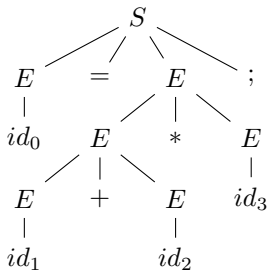
Mit der Syntaxanalyse wird die Wohlgeformtheit des Tokenstroms bzgl. einer kontextfreien Grammatik überprüft. Dabei ist ein Tokenstrom wohlgeformt, wenn die Bezeichner der Token zusammen ein Wort aus der Sprache der Grammatik bilden. Zur Syntaxanalyse werden kontextfreie Grammatiken eingesetzt, um die Struktur des Tokenstroms erfassen zu können - die Ausdrucksstärke regulärer Ausdrücke reicht dafür nicht aus. Für wohlgeformte Tokenströme erzeugen die als *Parser* bezeichneten Tools der Syntaxanalyse einen Parsebaum zur Wiedergabe der Struktur des Tokenstroms [1].

Das folgende Beispiel veranschaulicht Parsebäume als Datenstruktur zur Abbildung der Struktur eines Tokenstroms bzgl. einer Grammatik.

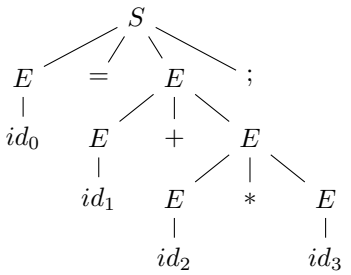
2.2.1 Beispiel. Betrachtet wird die mehrdeutige Grammatik G für Zuweisungen arithmetischer Ausdrücke mit Funktionsaufrufen und dem Startsymbol S aus Abbildung 2.6(a). Der Zeichenstrom w der Zuweisung $x = a + b * c;$ besitzt bzgl. der Grammatik G zwei Parsebäume, die sich in der Assoziativität der arithmetischen Operatoren $+$ und $*$ unterscheiden. Die Parsebäume für w sind in den Abbildungen 2.6(b) (für linksassoziative Operatoren) und 2.6(c) (für rechtsassoziative Operatoren) dargestellt. Dabei stehen die Token id_0, id_1, id_2 und id_3 für die Variablen x, a, b und c; die Bezeichner der anderen Token entsprechen dem jeweiligen, zugeordneten Zeichen. Im ersten Fall wird die Addition vor der Multiplikation ausgewertet, im zweiten Fall ist es genau umgekehrt herum.

$S \to E = E;$
$S \to E;$
$E \to E + E$
$E \to E * E$
$E \to E\,(E)$
$E \to id$

(a) Grammatik G



(b) Parsebaum für w



(c) Parsebaum für w

Abbildung 2.6: Beispiel: Kontextfreie Grammatik mit Parsebäumen (nach [52])

Für die Syntaxanalyse existieren verschiedene Arten von Parsern, die sich anhand der verwendeten Verfahren und der Art zur Erzeugung des Parsebaums unterscheiden. Nach [1] wird zwischen *universellen Parsern*, *Top-Down Parsern* und *Bottom-Up Parsern* unterschieden. Universelle Parser nutzen Verfahren wie den Algorithmus von Cocke-Younger-Kasami (bekannt als CYK-Algorithmus) oder Earley's Algorithmus. Diese Parser können Tokenströme beliebiger kontextfreier Grammatiken parsen, sind aber nicht sehr effizient (CYK- und Earley-Algorithmus benötigen $\mathcal{O}(n^3)$ Zeit für einen Tokenstrom mit n Token[9]), sodass sie in der Praxis nicht sehr häufig verwendet werden. Bei Top-Down und Bottom-Up Parsern ist das Vorgehen bei der Konstruktion des Parsebaums fest vorgegeben. Top-Down Parser erzeugen den Parsebaum beginnend bei der Wurzel und arbeiten sich zu den Blättern vor. Bottom-Up Parser dagegen erzeugen den Parsebaum beginnend bei den Blättern und landen am Ende bei der Wurzel. Sowohl Top-Down Parser als auch Bottom-Up Parser können dabei nicht für alle kontextfreien Grammatiken implementiert werden, sondern sind auf bestimmte Teilmengen kontextfreier Grammatiken beschränkt.

Die Techniken zum Top-Down Parsing und Bottom-Up Parsing werden in den folgenden beiden Abschnitten detaillierter vorgestellt. Anschließend wird der Parsergenerator ANTLR v4 präsentiert, der zum Erzeugen eines Parsers für die Verarbeitung einer Spezifikation innerhalb des implementierten Parsergenerators eingesetzt wird.

[9] Der Earley-Algorithmus benötigt bei nicht-mehrdeutigen Grammatiken für n Token $\mathcal{O}(n^2)$ Zeit [1].

2.2.1 Top-Down Parsing

Beim Top-Down Parsing wird der Parsebaum bei der Suche nach einer Linksableitung für einen Tokenstrom top-down von der Wurzel zu den Blättern erzeugt. Die Schwierigkeit besteht darin, in jedem Schritt des Vorgangs für das aktuelle Nichtterminal der Grammatik die passende Regel zu finden, sodass am Ende der komplette Tokenstrom vom Startsymbol der Grammatik abgeleitet wird [1]. Bei Top-Down Parsern existieren zwei typische Varianten: *Backtracking-Parser* und *vorhersagende Parser*.

In einfachen Top-Down Parsern wird das Prinzip des rekursiven Abstiegs (engl. *recursive descent*) mit Backtracking kombiniert, um den Parsebaum für einen Tokenstrom zu erzeugen. Dabei werden für das jeweils aktuelle Nichtterminal der Grammatik alle Regeln der Reihe nach über rekursive Methodenaufrufe ausprobiert. Bei einem Parsefehler wird der Zeiger im Tokenstrom zurückgesetzt und die nächste Regel des Nichtterminals ausprobiert. Falls alle verfügbaren Regeln des Nichtterminals erfolglos ausprobiert wurden, wird ein Fehler an die aufrufende Methode zurückgegeben. Die Syntaxanalyse für einen Tokenstrom scheitert, wenn alle Regeln des Startsymbols der Grammatik erfolglos ausprobiert werden. Durch die Verwendung von Backtracking sind Top-Down Parser, die nach dem Prinzip des rekursiven Abstiegs funktionieren, mit exponentieller Laufzeit sehr ineffizient, dafür aber sehr einfach zu implementieren [1].

Für eine Teilmenge der kontextfreien Grammatiken können mit LL-Parsern deterministische Top-Down-Parser implementiert werden. Die Auswahl der nächsten Regel des aktuellen Nichtterminals erfolgt unter Berücksichtigung der folgenden k Token des Tokenstroms als *Lookahead*. Mit dem Lookahead muss die nächste Regel der Grammatik dabei eindeutig bestimmt werden können. Ein solcher Parser wird als $LL(k)$*-Parser* bezeichnet, die kontextfreie Grammatik als $LL(k)$*-Grammatik*. Durch die Verwendung des Lookaheads wird die Auswahl der anzuwendenden Regel für ein Nichtterminal deterministisch, sodass die Laufzeit des Parsers linear bleibt [1]. In der Praxis ist die Größe des Lookaheads häufig auf eins oder zwei beschränkt.

Es gibt aber auch Fälle, in denen ein fest definierter maximaler Lookahead nicht sinnvoll ist und die Größe des benötigten Lookaheads dynamisch zur Parsezeit bestimmt wird. In diesem Fall handelt es sich um $LL(*)$*-Parser*. Zur Realisierung des dynamischen Lookaheads verwenden $LL(*)$-Parser sogenannte *Lookahead-DFAs* auf Basis endlicher Automaten sowie Backtracking [51]. Für jedes Nichtterminal der Grammatik wird ein Lookahead-DFA durch statische Analyse der Grammatik zur Unterscheidung der Regeln des Nichtterminals erzeugt. Wenn die Lookahead-Sprache eines Nichtterminals kontextfrei ist und damit das Erzeugen eines Lookahead-DFAs nicht möglich ist, wird Backtracking eingesetzt. Die Lookahead-DFAs werden nur zur Bestimmung der nächsten anzuwendenden Regel für das aktuelle Nichtterminal eingesetzt, nicht zum Parsen des Tokenstroms selbst.

Alternativ zu Top-Down Parsern können Bottom-Up Parser verwendet werden.

2.2.2 Bottom-Up Parsing

Bottom-Up Parser erzeugen den Parsebaum für einen Tokenstrom bottom-up von den Blättern zur Wurzel und bestimmen dabei eine Rechtsableitung für den Tokenstrom. Der Tokenstrom wird dazu schrittweise auf das Startsymbol der Grammatik reduziert, indem in jedem Reduktionsschritt mehrere Token, die zusammen den Rumpf einer Regel der Grammatik bilden, auf den Kopf der Regel reduziert werden. Die Herausforderung besteht darin, zu entscheiden, wann zu reduzieren ist und welche Regel der Grammatik dazu verwendet wird. Diese Parser werden als *Shift-Reduce-Parser* bezeichnet [1].

Shift-Reduce-Parser verwenden als Datenstruktur einen Stack für Symbole der Grammatik, auf den die Token des Tokenstroms geshiftet werden. Wenn oben auf dem Stack der Rumpf einer Grammatikregel (in umgekehrter Reihenfolge) liegt, können diese Token auf dem Kopf der Regel reduziert werden - dieser ersetzt den Rumpf auf dem Stack. Somit ist die Wahl zwischen den Aktionen *Shift* und *Reduce* eine der Herausforderungen dieser Parser. Ähnlich wie bei den LL-Parsern existieren auch deterministische Bottom-Up Parser, die einen Lookahead nutzen. Sie werden als $LR(k)$-*Parser* bezeichnet, die jeweiligen Grammatiken als $LR(k)$-*Grammatiken*. Anhand des Lookaheads und des Stackinhaltes wird entschieden, ob geshiftet oder mit welcher Regel der Grammatik reduziert wird [1].

Zur Unterstützung der Implementierung von Lexern und Parsern existieren Softwarewerkzeuge, die Lexer oder Parser generieren und damit die lexikalische Analyse oder die Syntaxanalyse vereinfachen. Scannergeneratoren wie *Lex* erzeugen vollständige Lexer aus der Beschreibung von Token durch reguläre Ausdrücke. Parsergeneratoren generieren aus einer grammatikalischen Beschreibung einer Sprache einen Parser für diese Sprache. Beispiele für Parsergeneratoren sind *Yacc*, *GNU Bison*[10] oder *ANTLR*. Die generierten Lexer oder Parser können dann als Teil größerer Softwaretools wie z. B. Compiler verwendet werden. Mit einem von ANTLR v4 generierten Parser erfolgt das Einlesen der Spezifikation des implementierten Parsergenerators für das Pattern Matching auf FPGAs, sodass ANTLR im nächsten Abschnitt näher vorgestellt wird.

2.2.3 ANTLR v4

ANTLR v4[11] (im Folgenden als ANTLR bezeichnet) ist ein in Java geschriebener Parsergenerator, der Top-Down Parser für Sprachen wie Java oder C# generiert. Bei den von ANTLR generierten Parsern handelt es sich um *Adaptive* $LL(*)$-*Parser* (kurz $ALL(*)$-*Parser*) [52], die einen beliebigen Lookahead für die Auswahl der passenden Grammatikregel nutzen. $ALL(*)$-Parser können für alle kontextfreien Grammatiken generiert werden, sofern diese keine Linksrekursion enthalten. In ANTLR ist zudem ein Algorithmus zur Elimination von direkter Linksrekursion integriert. Als Ausgabe berechnet ein von

[10] Lex und Yacc: `http://dinosaur.compilertools.net`; Bison: `http://www.gnu.org/software/bison/`
[11] **AN**other **T**ool for **L**anguage **R**ecognition (Version 4), `http://www.antlr.org/`

ANTLR generierter Parser einen Parsebaum, der über Listener oder nach dem Visitor-Entwurfsmuster [21] weiterverarbeitet werden kann.

Wie bei $LL(*)$-Parsern werden auch bei $ALL(*)$-Parsern Lookahead-DFAs zur Umsetzung beliebiger Lookaheadgrößen eingesetzt. Allerdings werden die Lookahead-Automaten für $ALL(*)$-Parser nicht durch eine statische Analyse der Grammatik berechnet, sondern dynamisch zur Parsezeit individuell für einen Tokenstrom erzeugt. Somit entfällt die statisch unentscheidbare Analyse der Grammatik, die bei $LL(*)$-Parsern zur Verwendung von Backtracking führt [52]. Die von ANTLR generierten Parser konstruieren die Lookahead-DFAs für einen Tokenstrom inkrementell und setzen Caching zur Steigerung der Effizienz ein. Da zur Parsezeit immer nur endliche Teilmengen einer kontextfreien Sprache betrachtet werden müssen, reicht die Mächtigkeit regulärer Sprachen für die Auswahl der nächsten Regel der Grammatik stets aus.

Die inkrementelle Berechnung der Lookahead-DFAs für einen $ALL(*)$-Parser wird anhand eines einfachen Beispiels verdeutlicht.

2.2.2 Beispiel ([52]). Betrachtet wird die Grammatik G aus Beispiel 2.2.1. Für das Nichtterminal S wird der Lookahead-DFA für den Zeichenstrom $x = y; f(x);$ wie folgt gebildet: Zu Beginn besteht der Lookahead-DFA nur aus dem Startzustand (siehe Abbildung 2.7(a)). Für den Präfix $x = y;$ der Eingabe werden nur die beiden Token für x und $=$ benötigt, um eindeutig die erste Regel für S in G zur Ableitung auszuwählen (Abbildung 2.7(b)). Für das Parsen von $f(x);$ wird der Lookahead-DFA erweitert (Abbildung 2.7(c)) und schließlich die zweite Regel für S gewählt.

(a) Beginn (b) Nach Parsen von $x = y;$ (c) Nach Parsen von $x = y; f(x);$

Abbildung 2.7: Beispiel: Inkrementelle Berechnung eines Lookahead-DFAs [52]

In einigen Fällen muss neben dem Lookahead des Tokenstroms auch der Stack des $ALL(*)$-Parsers zur Bestimmung der nächsten Regel für das aktuelle Nichtterminal der Grammatik berücksichtigt werden. Damit nicht für jeden möglichen Stackinhalt ein Lookahead-DFA berechnet werden muss, wird ein optimierter LL-Modus verwendet, bei dem zunächst auf den Stack für die Auswahl der nächsten Regel verzichtet wird - dabei handelt es sich um faktisch um einen *Strong LL-Parser* (kurz *SLL-Parser*). Falls Konflikte bei der Regelauswahl auftreten, erfolgt ein zweiter Durchlauf mit Berücksichtigung des Stackinhaltes. Die von ANTLR generierten Parser verarbeiten einen Tokenstrom zusätzlich in zwei Phasen. In der ersten Phase werden alle Lookahead-DFAs ohne Betrachtung des Stacks gebildet. Falls dort Syntaxfehler auftauchen, kann dies an der Schwäche des

SLL-Parsing liegen oder es handelt sich tatsächlich um Syntaxfehler. Daher wird in der ggf. benötigten zweiten Phase der optimierte LL-Modus für das Parsen des Tokenstroms verwendet. So kann die Effizienz des Parsevorgangs insgesamt gesteigert werden [52].

Die Worst-Case-Laufzeit der von ANTLR generierten Parser für Tokenströme mit n Token liegt bei $\mathcal{O}(n^4)$, wobei in der Praxis häufig Linearzeit erreicht wird. Durch die dynamische Analyse einer Grammatik können jedoch Mehrdeutigkeiten in der Grammatik nur zur Parsezeit und nur bei Eingabe mehrdeutiger Tokenströme erkannt werden [52].

Die Darstellung des Parsing als ein Bestandteil des implementierten Parsergenerators ist damit abgeschlossen. Im nächsten Abschnitt wird der Fokus auf FPGAs als Zielplattform der Arbeit sowie die Realisierung von Pattern Matching auf FPGAs gelegt.

2.3 FPGAs und Pattern Matching

Field-Programmable Gate Arrays (kurz *FPGAs*) sind Hardwareschaltungen, die nach der Herstellung programmiert und damit für verschiedenste Aufgaben eingesetzt werden können. FPGAs bestehen vor allem aus kombinatorischen Logik-Bausteinen, Speicherelementen und Kommunikationsleitungen. Mit der kombinatorischen Logik können Berechnungen durchgeführt werden, deren Ergebnisse in den Speicherelementen zwischengespeichert werden. Die Verbindungsleitungen erlauben die Kommunikation zwischen verschiedenen Hardwarebausteinen. Nach der Programmierung sind FPGAs für Aufgaben wie das Pattern Matching einsetzbar. Die Reprogrammierbarkeit wird durch eine feste Architektur erreicht, welche nicht auf eine spezielle Schaltung ausgerichtet ist. Damit können FPGAs u. a. über die Zeit variierende Anforderungen beim Pattern Matching abdecken [49, 62].

Im Folgenden wird zunächst der Aufbau und die Programmierung von FPGAs beschrieben. Anschließend folgt die Darstellung der Realisierung endlicher Automaten auf FPGAs, mit denen Pattern Matching in Hardware umgesetzt werden kann.

2.3.1 Aufbau von FPGAs

Die Architektur von FPGAs wird in diesem Abschnitt hierarchisch vorgestellt, beginnend mit den Basiselementen bis hin zum Layout eines kompletten Chips auf dem FPGA. Dabei werden die zentralen Bestandteile von FPGAs erläutert, mit denen Hardwareschaltungen für Aufgaben wie das Pattern Matching realisiert werden können.

Ein wesentlicher Bestandteil von kombinatorischen Schaltungen sind Logikgatter, die beispielsweise für arithmetische Funktionen oder als Strukturen für bedingte Anweisungen genutzt werden können [25]. Zur Gewährleistung der Reprogrammierbarkeit werden Logikgatter auf FPGAs durch programmierbare *Lookup-Tabellen* (kurz *LUTs*) simuliert. Eine Lookup-Tabelle mit n Eingängen kann eine beliebige n-stellige boolesche Funktion realisieren, indem die Eingänge den Index einer Zelle der LUT bilden, die den jeweiligen Funktionswert enthält. Mit einem einfachen Lookup wird der Funktionswert aus der

Zelle der Lookup-Tabelle ausgelesen [62]. Für die Simulation von Und-Gattern bzw. Oder-Gattern mit zwei Eingängen durch eine LUT werden beispielsweise vier Zellen benötigt, in denen die Ergebnisse der booleschen Verknüpfung abgelegt werden (siehe Abbildung 2.8).

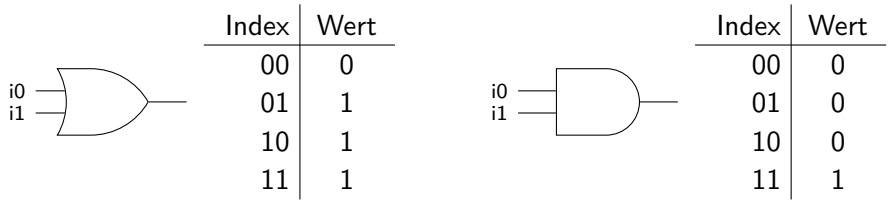


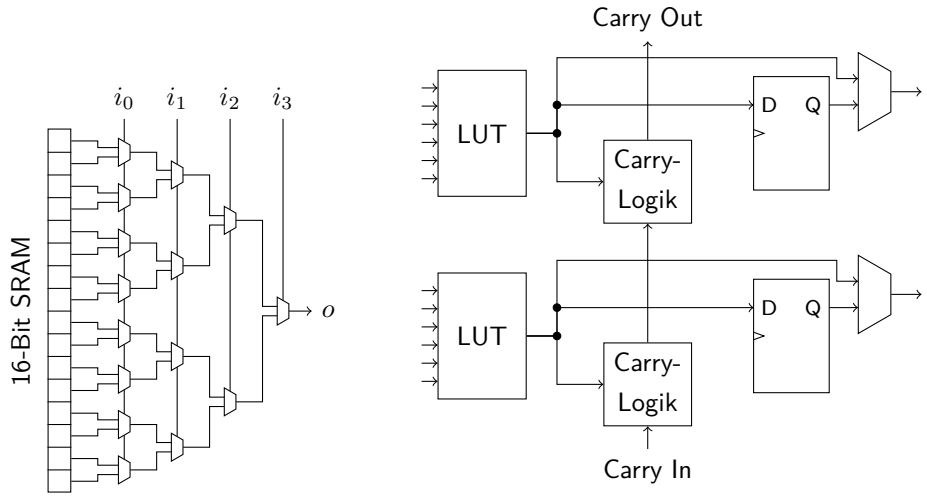
Abbildung 2.8: Beispiel: Simulation von kombinatorischer Logik durch LUTs (nach [62])

Eine Lookup-Tabelle mit n Eingängen besteht intern aus einem SRAM-Baustein mit 2^n Bit zur Abspeicherung der Inhalte der Tabellenzellen. Hinzu kommt ein $2^n : 1$-Multiplexer für das Auslesen eines Bits der Tabellenkonfiguration. Der Multiplexer wird typischerweise als ein Baum von $2 : 1$-Multiplexern realisiert (siehe Abbildung 2.9(a)). Durch die Verwendung von SRAM für die Realisierung von LUTs ist die Reprogrammierung mit einem Update der Inhalte des SRAM-Bausteins direkt möglich. Da der SRAM-Baustein als Shiftregister mit einer Tiefe von 2^n Bit und einer Breite von einem Bit organisiert ist, benötigt das Konfigurieren der Tabelle 2^n Taktzyklen. Das Auslesen einer Zelle ist dagegen in einem Taktzyklus möglich. Aktuelle FPGAs verfügen über Lookup-Tabellen mit vier oder sechs Eingängen [62].

Neben der Möglichkeit, Lookup-Tabellen zur Simulation von Logikgattern zu verwenden, können LUTs auch als verteilter RAM (z. B. für kleine FIFOs) oder als Shiftregister genutzt werden [25, 62]. Als verteilter RAM bietet eine LUT einen $2^n \times 1$ LUT-RAM Block, der mit weiteren Lookup-Tabellen zu größeren und breiteren RAM-Blocken zusammengefasst werden kann. Mit Hilfe dieser verteilten RAM-Blöcke können beispielsweise FIFOs realisiert werden. Alternativ können LUTs auch als Shiftregister genutzt werden. Die Logik für die Shiftregister-Funktionalität ist bereits durch das Verfahren zur Programmierung des SRAM-Bausteins vorhanden, sodass dafür keine weiteren Hardwareressourcen benötigt werden.

Lookup-Tabellen werden auf FPGAs mit einigen anderen Hardwarebausteinen zu *elementaren Logikeinheiten* zusammengefasst, die von Xilinx[12] als *Slices* bezeichnet werden. Eine elementare Logikeinheit besteht neben einer geringen, festen Anzahl von LUTs (typischerweise zwei bis acht) aus einer proportionalen Anzahl von 1-Bit Registern, arithmetischer Logik und einigen Multiplexern [62]. Eine Logikeinheit mit zwei LUTs ist in Abbildung 2.9(b) dargestellt. Jeder Lookup-Tabelle der Logikeinheit ist ein 1-Bit Register

[12] Xilinx (`http://www.xilinx.com/`) ist neben Altera einer der Hersteller von FPGAs mit großen Marktanteil.



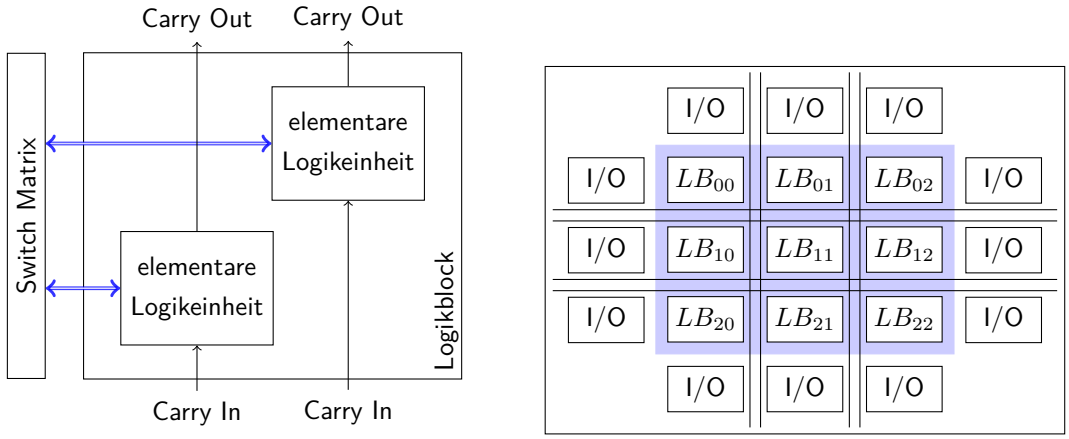
(a) Interne Struktur einer LUT (b) Elementare Logikeinheit mit Carry-Logik

Abbildung 2.9: Interne Struktur von LUTs und elementare Logikeinheit auf einem FPGA [62]

in Form eines Flip-Flops zugeordnet, welches das Lookup-Ergebnis der LUT zwischenspeichert, um Signallaufzeiten zu entkoppeln und höhere Taktraten zu ermöglichen. Bei Bedarf (wie z. B. zur Realisierung boolescher Funktionen mit vielen Eingängen) kann das Flip-Flop für die Zwischenspeicherung mit einem Multiplexer übersprungen werden, indem dieser das Lookup-Ergebnis direkt weiterleitet. Die Konfiguration des Multiplexer erfolgt über einen SRAM-Baustein, sodass auch hier die Programmierbarkeit gegeben ist. Mit der arithmetischen Logik können mehrere benachbarte Lookup-Tabellen zur Realisierung von arithmetischen Blöcken wie Addierer oder Multiplizierer kombiniert werden. Die arithmetische Logik wird dabei als Carry-Logik u. a. zur Weitergabe von Überträgen verwendet und ist auf dem FPGA als Menge vertikaler Kommunikationsleitungen zwischen den einzelnen LUTs der Logikeinheit realisiert [62].

Neben der Kommunikation benachbarter Lookup-Tabellen mit Hilfe der Carry-Logik können auch beliebige Logikeinheiten auf einem FPGA miteinander kommunizieren. Dazu besitzt das FPGA ein leistungsfähiges Kommunikationssystem, das als *Interconnect* bezeichnet wird [62]. Für die Anbindung an den Interconnect werden mehrere elementare Logikeinheiten zu einem *Logikblock* gruppiert, der von Xilinx als *Configurable Logic Block* (kurz *CLB*) bezeichnet wird. Die Logikeinheiten innerhalb des Logikblocks werden über eine *Switch Matrix* an den Interconnect angebunden (siehe Abbildung 2.10(a)). Die einzelnen Logikblöcke sind auf dem FPGA als zweidimensionales Array angeordnet, wobei zwischen den Zeilen und Spalten des Arrays die Kommunikationsleitungen des Interconnect liegen (siehe Abbildung 2.10(b)). Über diese Kommunikationsleitungen können beliebige Logikblöcke miteinander kommunizieren. Zur Vorbereitung der Kommunikation für eine konkrete Anwendung werden die Schnittpunkte der Verbindungsleitungen über SRAM-

Zellen konfiguriert und aktivieren bzw. deaktivieren so bestimmte Kommunikationswege.



(a) Logikblock mit Switch Matrix

(b) Arraystruktur eines FPGAs

Abbildung 2.10: Switch Matrix-Anbindung und Arraystruktur auf FPGAs [62]

Für die Kommunikation von Schaltungen auf einem FPGA mit der Außenwelt existieren I/O-Blöcke, die einerseits mit der Peripherie des FPGA und andererseits mit dem Interconnect verbunden sind. Dadurch kann beispielsweise über USB oder über das Netzwerk mit anderen Geräten kommuniziert werden. Die I/O-Blöcke sind rund um das zweidimensionale Array der Logikblöcke angeordnet (siehe Abbildung 2.10(b)) [62].

Neben den vorgestellten grundlegenden Bausteinen existieren auf FPGAs auch einige bereits vorgefertigte, komplexere Bausteine. Darunter fallen Block-RAM Elemente für die Speicherung größerer Datenmengen, digitale Signalprozessoren für spezielle mathematische Operationen oder einzelne, dedizierte Prozessoren [62].

Die Vorbereitung eines FPGAs zum Einsatz für Aufgaben wie dem Pattern Matching erfolgt durch die Programmierung, welche im folgenden Abschnitt betrachtet wird.

2.3.2 Programmierung von FPGAs

Für die Programmierung eines FPGAs wird die Beschreibung einer Schaltung mit einer Hardware-Beschreibungssprache (engl. *Hardware Description Language*, kurz *HDL*) wie VHDL[13] oder Verilog benötigt. Anschließend kann mit Hilfe eines Tools für die Synthese (wie z. B. Xilinx ISE) aus der Beschreibung der Schaltung ein Bitstream generiert werden, der auf ein FPGA geladen werden kann. Durch den Upload des Bitstreams wird das FPGA programmiert und kann anschließend verwendet werden. Der Prozess der Programmierung eines FPGAs von Xilinx ist in Abbildung 2.11 grafisch dargestellt. Im Folgenden werden die einzelnen Phasen des Prozesses kurz erläutert [62]: Die Programmierung eines FPGA

[13] Very High Speed Integrated Circuit Hardware Description Language

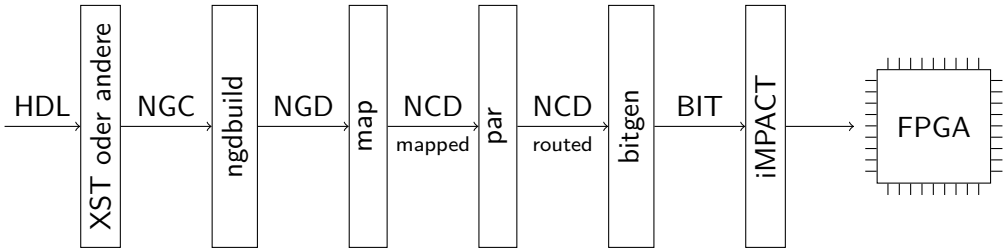


Abbildung 2.11: Syntheseprozess für FPGAs von Xilinx [62]

startet mit der Beschreibung einer Schaltung über eine HDL, welche durch den Xilinx Synthesizer (XST) in eine Menge von Netzlisten auf Gatterebene (engl. *Native Generic Circuit*, kurz *NGC*) umgewandelt wird. Mit dem Tool *ngdbuild* werden alle Netzlisten zusammen mit den Constraints des Designs in eine einzelne Netzliste (engl. *Native Generic Database*, kurz *NGD*) übersetzt. Das *map*-Werkzeug bildet die NGD auf Ressourcen wie Logikblöcke, I/O-Blöcke oder BRAM-Elemente ab und generiert eine *Native Circuit Description* (kurz *NCD*) mit dem physikalischen Mapping. Anschließend folgt mit dem Tool *par* der Place & Route Prozess zum Platzieren der benötigten Hardwarebausteine auf konkrete Schaltungselemente des FPGAs. Durch das Tool *bitgen* wird ein Bitstream für die Programmierung des FPGAs generiert, der mit dem Tool *iMPACT* auf das FPGA geladen werden kann.

Zur Programmierung von FPGAs ist anzumerken, dass der gesamte Prozess beginnend mit der Schaltungsbeschreibung in einer HDL bis hin zu einem Bitstream, der auf ein FPGA geladen werden kann, einige Zeit (ggf. mehrere Stunden) in Anspruch nehmen kann. Somit spielt die Laufzeit des implementierten Parsergenerators durch die nachfolgende Synthese auf dem FPGA bzgl. der Gesamtlaufzeit nur eine untergeordnete Rolle, sofern die Laufzeit des Parsergenerators in einem akzeptablen Verhältnis zur Eingabegröße steht.

Nach der Betrachtung der Struktur von FPGAs werden existierende Realisierungen von endlichen Automaten für das Pattern Matching auf FPGAs vorgestellt.

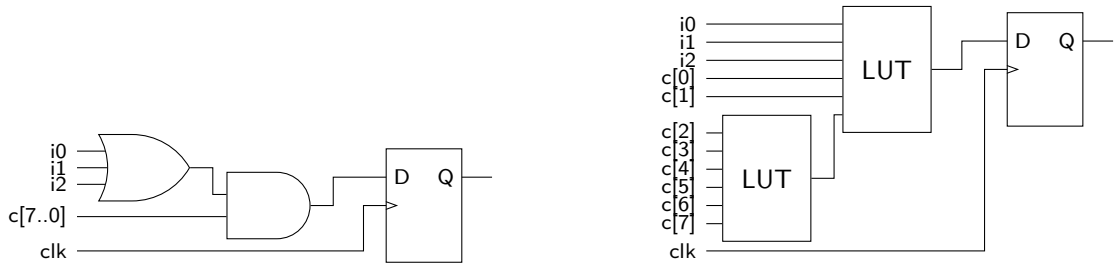
2.3.3 Realisierung endlicher Automaten auf FPGAs

Endliche Automaten können als deterministische oder nichtdeterministische Automaten auf FPGAs in Hardware realisiert werden. Durch die inhärente Parallelität der einzelnen Hardwarebausteine auf einem FPGA können mehrere Zustände problemlos gleichzeitig aktiv sein und mehrere Transitionen eines NFA zeitgleich schalten, sodass die Nachteile von Software-Implementierungen nichtdeterministischer Automaten auf FPGAs entfallen. Im Folgenden werden nur Lösungen zur Realisierung nichtdeterministischer endlicher Automaten auf FPGAs betrachtet, welche für den Parsergenerator verwendet werden können.

Pattern Matching auf FPGAs auf Basis nichtdeterministischer endlicher Automaten ist in zahlreichen Arbeiten und Artikeln der letzten Jahre beschrieben worden. Dabei

steht neben der Realisierung von NFAs auf FPGAs die Optimierung im Fokus. Die Übertragung von NFAs auf FPGAs erfolgt durch Abbildung der Zustände auf Register und der Transitionen auf kombinatorische Logik zum Matchen von Zeichen. Dabei wird jedem Zustand eines NFA ein Register zugeordnet, welches genau dann aktiv ist, wenn der korrespondierende Zustand aktiv ist. Dieses Prinzip wird als *One-Hot Encoding* bezeichnet. Signalleitungen verbinden die Transitionslogik mit den Zustandsregistern entsprechend der Transitionen des NFA [58, 59, 72, 73]. Die Lösungen der Arbeiten unterscheiden sich darin, ob die Transitionslogik vor oder nach dem Register für den korrespondierenden Zustand angeordnet ist. Während in [58] das Zustandsregister vor der Matching-Logik angeordnet ist (und der NFA damit mit einem Register beginnt), ist die Reihenfolge in [72, 73] genau umgekehrt. Dadurch lässt sich der endliche Automat besser auf die Struktur der elementaren Logikeinheiten des FPGA mit Lookup-Tabellen als Datenquelle für Register abbilden, sodass diese Lösung bevorzugt wird. Ein weiterer Unterschied der Lösungen liegt beim Matching der Zeichen des Eingabestroms innerhalb der Automaten. Das Matchen der Zeichen kann vollständig lokal in der kombinatorischen Logik der Transitionen erfolgen, sodass die Eingabezeichen mit Signalleitungen zu jeder Transitionslogik transportiert werden müssen. Alternativ kann das Matchen der Zeichen auch zentral (z. B. über einen BRAM-Block) erfolgen, sodass an die kombinatorische Logik für die Transitionen lediglich ein 1-Bit Signal angebunden ist. Dieses Signal gibt an, ob ein für die Transition matchendes Zeichen verarbeitet wird. Beide Varianten können auch kombiniert werden, um Ressourcen einzusparen und somit komplexere Logik auf einem FPGA realisieren zu können [60, 73].

Für den implementierten Parsergenerator werden NFAs betrachtet, die auf FPGAs mit lokalen Matching der Zeichen an der Transitionslogik realisiert werden. Die Abbildung der endlichen Automaten auf Hardware erfolgt über einen Basisbaustein, welcher für alle Automaten verwendet wird [73]. Die Verbindung der Ein- und Ausgänge der einzelnen Basisbausteine wird durch die Transitionen des Automaten bestimmt. Der Basisbaustein ist in Abbildung 2.12(a) als Schaltkreis und in Abbildung 2.12(b) als FPGA-Umsetzung skizziert. Das aktuelle Eingabezeichen des zu matchenden Datenstroms bildet als Bün-



(a) Schaltung

(b) FPGA-Umsetzung

Abbildung 2.12: Basisbaustein für endliche Automaten auf FPGAs (nach [73])

del von acht 1-Bit Leitungen den Eingang $c[7..0]$ der kombinatorischen Transitionslogik.

Zusätzlich sind in dem dargestellten Fall drei Eingangsleitungen $i0, i1$ und $i2$ von Vorgängerzuständen vorhanden. Diese werden zunächst disjunktiv verknüpft und das Ergebnis bildet einen Eingang zur Konjunktion mit den gesuchten Zeichen. Nur wenn mindestens ein vorheriger Zustand aktiv ist und ein zur Transition matchendes Eingabezeichen verarbeitet wird, wird das Register des Zustandes als gesetzt markiert und der Zustand ist aktiv. Andernfalls wird das Register nicht gesetzt und der Zustand ist inaktiv.

Mit Hilfe des Basisbausteins lassen sich beliebige endliche Automaten auf einem FPGA realisieren, indem jeder Zustand des NFA durch einen oder mehrere Basisbausteine repräsentiert wird. Die Transitionen bestimmen die Verschaltung der Ausgänge von Basisbausteinen mit den Eingängen nachfolgender Basisbausteine [72, 73]. Somit kann jeder beliebige NFA unabhängig von den in dem korrespondierenden regulären Ausdruck vorkommenden Operatoren mit einer einzigen Grundstruktur in Hardware realisiert werden.

Die Realisierung nichtdeterministischer endlicher Automaten auf FPGAs kann zusätzlich durch Optimierungen, die speziell auf die Verwendungsmöglichkeiten der Hardwarebausteine auf FPGAs ausgelegt sind, hinsichtlich verschiedener Kriterien wie dem Ressourcenbedarf oder dem Durchsatz verbessert werden. Beispielsweise können für reguläre Ausdrücke mit bedingten Wiederholungen der Form $\alpha\{n\}, \alpha\{n, m\}$ und $\alpha\{n, \}$ Shiftregister und Zähler eingesetzt werden, wenn α ein einzelnes Zeichen oder eine Zeichenklasse ist. Dadurch können Ressourcen für die kombinatorische Logik sowie Zustandsregister eingespart werden [5, 59]. Zur Steigerung des Durchsatzes beim Pattern Matching können mehrere Eingabesymbole innerhalb eines Taktzyklus verarbeitet werden, wenn dafür ein erhöhter Ressourcenbedarf in Kauf genommen wird. Dies ist u. a. mit dem in [72, 73] vorgestellten Multi-Character-Matching möglich, welches auch mit den Shiftregistern für bedingte Wiederholungen kombiniert werden kann. Falls wie bei der lexikalischen Analyse in Compilern oder bei den NIDS Snort und Bro mehrere Automaten auf einem FPGA parallel und unabhängig voneinander eingesetzt werden sollen, kann Pipelining in Verbindung mit Staging zur Entkopplung von Signallaufzeiten und der Steigerung des Durchsatzes eingesetzt werden. Zwar müssen dabei Latenzen bei der Verarbeitung in Kauf genommen werde, im Gegenzug können aber Prioritäten zwischen den einzelnen NFAs definiert werden, sodass bestimmte reguläre Ausdrücke anderen Ausdrücken vorgezogen werden [72, 73]. Wenn mehrere endliche Automaten gleichzeitig für das Pattern Matching zum Einsatz kommen, können gemeinsame Präfixe, Infixe oder Suffixe der korrespondierenden regulären Ausdrücke zur Einsparung von Ressourcen auf dem FPGA zusammengefasst werden. Beim Teilen von Hardwareressourcen für gemeinsame Infixe oder Suffixe ist auf die Vermischung der einzelnen Ausdrücke zu achten [40]. Die Auflistung weiterer Optimierungen ließe sich beliebig weiterführen, sodass bei Bedarf auf die einschlägige Literatur verwiesen wird.

Mit der Abbildung nichtdeterministischer endlicher Automaten auf FPGAs liegen die wichtigsten Grundlagen vor, die zur Implementierung eines Parsergenerators für das Pattern Matching auf FPGAs benötigt werden. Im folgenden Kapitel kann daher die Archi-

tektur des entwickelten Parsergenerators präsentiert werden. Zudem werden die zentralen Entwurfsentscheidungen der Implementierung vorgestellt.

Kapitel 3

Architektur des Parsergenerators

Für das Pattern Matching auf FPGAs wird im Folgenden der im Rahmen dieser Arbeit implementierte Parsergenerator vorgestellt. Der Parsergenerator erzeugt für eine Spezifikation von regulären Ausdrücken in Verbindung mit Aktionscode eine VHDL-Beschreibung auf Basis endlicher Automaten. Die VHDL-Beschreibung des Pattern Matchers kann für das Parsen von Datenströmen auf FPGAs synthetisiert werden (siehe Abbildung 3.1). Im Unterschied zu Snowfall [61] werden dabei nichtdeterministische endliche Automaten in Hardware realisiert, die um Aktionscode und semantische Bedingungen angereichert sind.

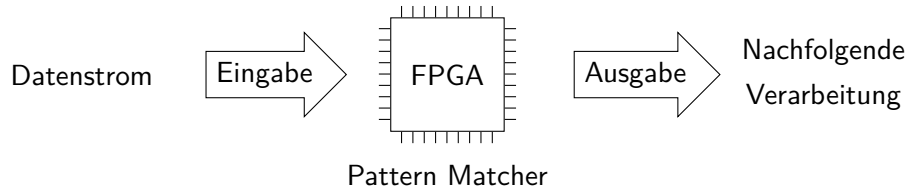


Abbildung 3.1: Einsatz eines generierten Pattern Matchers auf einem FPGA

Der implementierte Parsergenerator stellt für das Pattern Matching neben der reinen Erkennung von regulären Mustern auch Aktionen und semantische Prädikate zur Verfügung. Mit Aktionen kann ein Datenstrom zur Parsezeit beispielsweise komprimiert, gefiltert oder in einen Ausgabestrom transformiert werden, der an die nachfolgende Verarbeitung auf dem FPGA weitergereicht oder nach Übertragung über das Netzwerk auf anderen Geräten verarbeitet wird. Damit können generierte Pattern Matcher in verschiedensten Anwendungen wie der Überwachung des Netzwerkverkehrs oder zur Aggregation von Spalten von Datenbank-Tabellen eingesetzt werden. Mit semantischen Bedingungen, die Transitionen der endlichen Automaten zur Parsezeit dynamisch (de-)aktivieren, können zusätzlich Eingabeströme geparst werden, deren Verarbeitung kontextsensitive Informationen aus den Datenstrom benötigt. Ein Beispiel sind Paket-basierte Datenströme mit Feldern variabler Länge, bei denen die Feldlänge dem Feldinhalt vorangestellt ist [65].

Im Folgenden werden zunächst die Rahmenbedingungen für die Entwicklung des Parsergenerators vorgestellt. Anschließend folgt die Erläuterung der Struktur des Parsergenerators mit den zentralen Entwurfsentscheidungen sowie der nutzbaren Aktionen und semantischen Prädikaten. Schließlich wird das implementierte Automatenmodell für endliche Automaten mit Aktionscode und semantischen Bedingungen formal definiert sowie das Format der Spezifikationen für den Parsergenerator beschrieben.

3.1 Rahmenbedingungen

Der implementierte Parsergenerator orientiert sich am Tool Snowfall [61], welches ebenfalls ein Parsergenerator für das Pattern Matching auf FPGAs ist. Im Unterschied zu Snowfall basiert der implementierte Parsergenerator auf nichtdeterministischen endlichen Automaten zur effizienten Nutzung der Parallelität der Hardware und zur Vermeidung der Zustandsexplosion bei der Determinisierung. Dabei sind die theoretischen Grundlagen aus Kapitel 2 wie die Komplementierung oder die Minimierung von NFAs zu berücksichtigen. Der implementierte Parsergenerator ermöglicht wie Snowfall die Verwendung von Zeichenklassen an Transitionen der endlichen Automaten. Mit Zeichenklassen können Ressourcen für die Transitionslogik auf dem FPGA eingespart werden, wenn die niederwertigsten Bits der Zeichen des Datenstroms beim Matchen einer Zeichenklasse nicht benötigt werden. In diesem Fall werden weniger als acht Eingänge für das aktuelle Zeichen in der Transitionslogik benötigt, sodass kleinere Logikgatter verwendet werden können.

Der Parsergenerator ist in der Programmiersprache Java implementiert und verwendet das Build- und Dependency-Management-Tool Apache *Maven*[1] zum Bauen eines ausführbaren Jar-Archivs sowie zur Einbindung externer Bibliotheken. Der Parsergenerator nutzt einen von ANTLR generierten Parser zum Einlesen einer Spezifikation und für die Erzeugung eines Parsebaums. Die Generierung des ANTLR-Parsers aus einer ANTLR-Grammatik ist in den Build-Prozess mit Hilfe des ANTLR4 Maven Plugins integriert.

Der implementierte Parsergenerator verwendet neben Java 8 die folgenden externen Bibliotheken, die über die XML-Konfiguration von Maven eingebunden sind:

- Der generierte ANTLR-Parser benötigt die ANTLR4-Bibliothek als Laufzeitumgebung, welche unter der BSD Lizenz steht.
- Der Parser für die Kommandozeilenparameter innerhalb des Parsergenerators nutzt die Apache Commons CLI Bibliothek, welche unter der Apache Lizenz 2.0 steht.
- Für die interne Verarbeitung der Spezifikation und die Erzeugung der endlichen Automaten werden Utility-Klassen aus der Apache Commons Collections Bibliothek verwendet, die ebenfalls unter der Apache Lizenz 2.0 steht.

[1] Apache Maven, `https://maven.apache.org/`

Im nächsten Abschnitt wird die Struktur des implementierten Parsergenerators für das Pattern Matching auf FPGAs vorgestellt.

3.2 Struktur

Der implementierte Parsergenerator verarbeitet eine Spezifikation mit regulären Ausdrücken und Aktionscode in mehreren Phasen, beginnend beim Parsen der Spezifikation bis hin zur Generierung der VHDL-Beschreibung für das Pattern Matching. Für den Übergang der einzelnen Phasen werden definierte Datenstrukturen verwendet, mit denen die Ergebnisse der Berechnungen einer Phase den nachfolgenden Phasen zur Verfügung gestellt werden. Der Ablauf des Parsergenerators beim Verarbeiten einer Spezifikation ist in Abbildung 3.2 grafisch dargestellt, wobei die wesentlichen Datenstrukturen als Kantenbeschriftung angegeben sind. Dabei erfüllen die einzelnen Phasen folgende Teilaufgaben:

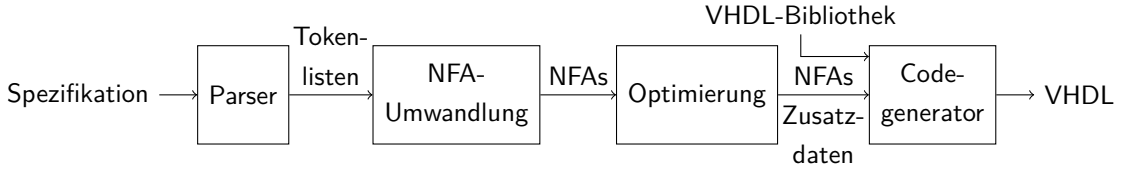


Abbildung 3.2: Struktur des Parsergenerators für das Pattern Matching auf FPGAs

- Parsen: Beim Parsen wird die Spezifikation bzgl. des Formats auf Korrektheit überprüft und anschließend in eine interne Datenstruktur umgewandelt, die aus dem Parsebaum erzeugt wird. Diese Datenstruktur enthält u. a. eine erweiterte Form der in [73] vorgestellten Tokenlisten für die Darstellung der regulären Ausdrücke.

- NFA-Umwandlung: Die regulären Ausdrücke aus den Tokenlisten werden durch ein angepasstes McNaughton-Yamada Verfahren [72, 73] in nichtdeterministische endliche Automaten umgewandelt, welche um die definierten Aktionen und semantischen Bedingungen erweitert werden. Aktionen und semantische Prädikate sind fortan Teil der erzeugten Automaten.

- Optimierung: In der Optimierungsphase werden die endlichen Automaten an sich sowie deren Abbildung auf die Hardware eines FPGAs optimiert. Darunter fallen u. a. die Reduzierung der Automatengröße sowie die effizientere Nutzung von Hardwarebausteinen auf FPGAs.

- Codgenerierung: In der Codegenerierungsphase wird unter Verwendung einer VHDL-Bibliothek für endliche Automaten eine VHDL-Beschreibung erzeugt, welche die regulären Ausdrücke der Spezifikation in Form mehrerer paralleler Automaten samt Ausführung des Aktionscodes und Auswertung der semantischen Prädikate enthält.

Bei der Implementierung des Parsergenerators spielen die nachfolgenden Entwurfsentscheidungen eine zentrale Rolle:

Innerhalb einer Spezifikation können wie bei Ragel[2] [13] mehrere reguläre Ausdrücke gleichzeitig als Top-Level-Ausdruck definiert werden. Die generierte VHDL-Beschreibung für den Pattern Matcher enthält für jeden Top-Level-Ausdruck einen separaten NFA, der nicht mit den NFAs der anderen Top-Level-Ausdrücke verbunden ist. Diese NFAs werden auf einem FPGA parallel ausgeführt und sind alle an die gleichen Eingangsports angebunden. Über Aktionscode und gemeinsam genutzte semantische Bedingungen können die einzelnen Automaten miteinander interagieren.

Die endlichen Automaten in der generierten VHDL-Beschreibung sind modular aufgebaut und bestehen vor allem aus Instanzen verschiedener VHDL-Entitäten für den Basisblock aus Kapitel 2. Die generierte VHDL-Beschreibung orientiert sich an den Arbeiten [72, 73], benötigt durch das implementierte lokale Matchen der Eingabezeichen in der Transitionslogik aber verschiedene Varianten von Basismodulen.

Die theoretischen Grundlagen aus Kapitel 2 beeinflussen die Implementierung des Parsergenerators an mehreren Stellen: Der Negationsoperator für reguläre Ausdrücke wird aufgrund der Schwierigkeiten bei der Komplementierung von NFAs durch deren Umwandlung in deterministische Automaten realisiert. Bei der Optimierung der Automaten durch Reduzierung der Automatengröße werden nur solche Verfahren eingesetzt, die auch bei größeren Automaten oder im Worst-Case-Fall von der Laufzeit her noch handhabbar sind. So wird beispielsweise auf die Implementierung eines Verfahrens zur Berechnung des minimalen NFAs wegen der hohen Komplexität und der damit verbundenen Laufzeit verzichtet.

Der Parsergenerator kann über Kommandozeilenparameter für die Generierung der VHDL-Beschreibung eines Pattern Matchers für FPGAs konfiguriert werden. Beispielsweise können Optimierungen deaktiviert oder zusätzliche Ausgaben eingeschaltet werden. Zur Visualisierung der intern erzeugten Tokenlisten und endlichen Automaten können Files für Graphviz Dot[3] generiert werden, die bei der Suche nach Fehlern in der Spezifikation behilflich sein können.

Im nächsten Abschnitt werden die im Parsergenerator integrierten Aktionen und semantischen Bedingungen für endliche Automaten detaillierter erläutert.

3.3 Aktionen und semantische Bedingungen

Mit Aktionen und semantischen Bedingungen für reguläre Ausdrücke einer Spezifikation kann ein Datenstrom komprimiert, gefiltert oder transformiert werden sowie eine Teilmen-

[2] Ragel State Machine Compiler (`http://www.colm.net/open-source/ragel/`) ist ein Tool zur Erzeugung ausführbarer endlicher Automaten aus regulären Ausdrücken für Sprachen wie C, C++ oder Java. Zudem bildet Ragel die Grundlage für das Tool Snowfall.

[3] Graphviz, `http://graphviz.org/`

ge der kontextsensitiven Sprachen geparst werden. Eine Aktion entspricht einem Block von VHDL-Code mit Anweisungen, der beim Schalten von Transitionen durch Matchen von Teilausdrücken ausgeführt wird. Semantische Bedingungen ermöglichen als spezielle Form von Aktionen die (De-)Aktivierung von Transitionen des Automaten zur Parsezeit. Im Folgenden werden Aktionen und semantische Bedingungen detaillierter vorgestellt.

Aktionen Die im Parsergenerator integrierten Aktionen orientieren sich an den Aktionen des Tools Ragel [13]. Wie Ragel unterstützt auch der implementierte Parsergenerator zwei Arten von Aktionen, die wiederum jeweils in mehrere Typen unterteilt sind:

Transitionsaktionen Transitionsaktionen werden für eine Menge von Transitionen spezifiziert und feuern beim Schalten einer dieser Transitionen.

Zustandsaktionen Zustandsaktionen werden für eine Menge von Zuständen spezifiziert und feuern beim Schalten von Transitionen, die in diese Zustände führen bzw. diese Zustände verlassen.

Transitionsaktionen gibt es in vier Aktionstypen, die jeweils eine Menge von Transitionen bestimmen, für die die jeweilige Aktion gilt. *Betretende Aktionen* werden einmalig beim Verlassen des Startzustandes eines Teilautomaten gefeuert und *Aktionen für alle Transitionen* werden bei jeder Transition des Teilautomaten aktiviert. *Beendende Aktionen* feuern bei Transitionen, die in einen akzeptierenden Zustand führen und *verlassende Aktionen* gelten für die Transitionen, die den Teilautomaten per Konkatenation oder Wiederholung verlassen. Zudem verhalten sich betretende Aktionen zugleich wie verlassende Aktionen, wenn die Sprache des regulären Ausdrucks das leere Wort ϵ enthält. Die Zustandsaktionen bestehen insgesamt aus zwölf Aktionstypen, je sechs Typen für das Betreten eines Zustandes (*betretende Zustandsaktionen*) und für das Verlassen eines Zustandes (*verlassende Zustandsaktionen*). Die Aktionstypen definieren die Menge der Zustände, für welche die jeweilige Aktion gilt: Alle Zustände, innere Zustände[4], Startzustand, akzeptierende Zustände, alle Zustände außer dem Startzustand und alle Zustände ohne Akzeptierende [65].

Der Aktionscode wird in einer Spezifikation als benannter Codeblock mit sequentiellen VHDL-Anweisungen definiert. In den regulären Ausdrücken wird dann nur der Aktionsname zusammen mit dem Aktionstyp angegeben. Durch die Trennung der Definition von der Referenzierung können Aktionen problemlos mehrfach verwendet werden. Mit Aktionscode kann der Eingabestrom gefiltert, in einen Ausgabestrom transformiert werden oder die Interaktion der verschiedenen endlichen Automaten der Top-Level-Ausdrücke realisiert werden. Es ist allerdings nicht empfehlenswert, mit Aktionscode direkt Einfluss auf die intern verwendeten VHDL-Signale der Automaten für das Pattern Matching zu nehmen. Im Unterschied zur Software-basierten Tools wie Ragel werden im implementierten

[4]Die inneren Zustände eines Automaten sind die Zustände, die weder Startzustand noch akzeptierender Zustand sind, d. h. für einen NFA $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ bildet die Menge $Q - F - \{s\}$ die inneren Zustände.

Parsergenerator einige Möglichkeiten von Aktionen wegen der Ausführung auf Hardware nicht unterstützt. Beispielsweise sind keine Aktionstypen für die Fehlerbehandlung implementiert, da diese auf Backtracking oder dem Vor- und Zurückspringen im Eingabestrom basieren. Solche Effekte lassen sich nur schwer in Hardware realisieren, wo insbesondere die Verarbeitungsgeschwindigkeit im Vordergrund steht.

Semantische Bedingungen Neben Aktionen, mit denen u. a. ein Ausgabestrom erzeugt werden kann, können in einer Spezifikation auch *semantische Bedingungen* (im Folgenden auch als *semantische Prädikate* bezeichnet) definiert werden. Diese ermöglichen das Parsen einiger kontextsensitiver Sprachen mit endlichen Automaten. Semantische Prädikate sind Aktionen, deren Aktionscode einem einzigen booleschen Ausdruck entspricht und erlauben die dynamische (De-)Aktivierung von Transitionen eines endlichen Automaten zur Parsezeit eines Datenstroms. Dadurch ist es beispielsweise möglich, Teilausdrücke nur solange zu matchen, bis eine maximale, zuvor im Datenstrom enthaltene Längenangabe noch nicht erreicht ist [65]. Es ist anzumerken, dass semantische Prädikate auch in Software-basierten Parsergeneratoren wie ANTLR eingesetzt werden können [52].

Für jeden Teilausdruck in der Spezifikation kann eine Menge von semantischen Prädikaten angegeben werden, wobei die Prädikate negiert oder nicht negiert einfließen können. Eine Transition des korrespondierenden Teilautomaten wird nur dann aktiviert, wenn alle semantischen Prädikate der Transition gleichzeitig erfüllt sind.

Verwendung Die Möglichkeiten des Pattern Matching in Verbindung mit Aktionen und semantischen Prädikaten werden anhand des folgenden Beispiels verdeutlicht.

3.3.1 Beispiel. Betrachtet wird ein fiktives Protokoll, welches aus einem Header mit Versionsinformationen gefolgt von einer Menge von Key-Value-Paaren besteht. Auf den Key eines Paares folgt die Angabe der variablen Länge des Values in Zeichen, sodass die Längeninformation für den Feldwert aus dem Kontext des zu parsenden Datenstroms stammen. Das Format des Datenstroms lässt sich mit den regulären Ausdrücken

$$
\begin{aligned}
Header &= \texttt{"V"}\ (\texttt{"1"}\ [\texttt{0-9}]\ \$ver \mid \texttt{"2"}\ [\texttt{0-9}]\{2\}\ \$ver);\\
Key &= ([\texttt{a-z}]\ when\ !inLen\ [\texttt{a-zA-Z0-9}]^*)\ \$id;\\
Length &= [\texttt{0-9}]^+\ \$len;\\
Value &= \texttt{.}^*\ \$val\ when\ inLen;\\
main &:= Header\ (Key\ \texttt{"␣"}\ Length\ \texttt{"="}\ Value\ \%out)^+\ \texttt{0x01}\ when\ !inLen;
\end{aligned}
$$

beschreiben, wobei `main` den Top-Level-Ausdruck definiert. Im Header können zwei verschiedene Versionsnummern für das Protokoll auftauchen. Nach der beliebig langen, nichtleeren Folge von Key-Value-Paaren folgt das ASCII-Zeichen SOH (mit dem Hex-Code 01) zur Markierung des Endes eines Pakets.

Mit regulären Ausdrücken alleine kann ein Datenstrom des fiktiven Protokolls nicht korrekt geparst werden, da beim Parsen des Value eines Key-Value-Paares nicht bestimmbar ist, ob nicht schon der Key des folgenden Paares vorliegt. Durch die Verwendung der Aktionen und semantischen Prädikaten des implementierten Parsergenerators kann dagegen die Längenangabe aus dem Kontext des Datenstroms ausgelesen und beim Parsen berücksichtigt werden. Dies wird anhand des Automaten aus Abbildung 3.3 für den Top-Level-Ausdruck `main` ersichtlich, wobei die Prädikate in Klammern angegeben sind und die Aktionen den letzten Teil der Transitionslabel bilden.

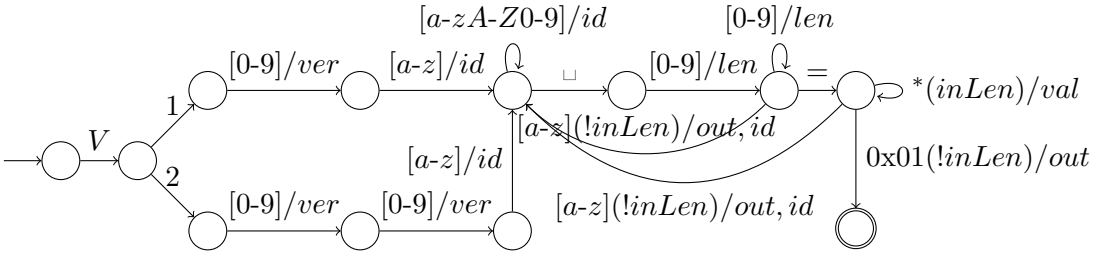


Abbildung 3.3: Beispiel: Automat eines Protokolls mit kontextsensitiven Informationen

Mit den Aktionen und semantischen Bedingungen sowie der Verwendung von Zeichenklassen handelt es sich bei den implementierten Automaten um eine erweiterte Form der in Kapitel 2 definierten NFAs. Das verwendete Automatenmodell wird daher im folgenden Abschnitt formal definiert.

3.4 Implementiertes Automatenmodell

Mit der Erweiterung der endlichen Automaten um Zeichenklassen, Aktionen und semantische Prädikate wird innerhalb des implementierten Parsergenerators ein angepasstes Automatenmodell verwendet, welches auf nichtdeterministischen endlichen Automaten basiert. Das implementierte Automatenmodell ist formal wie folgt definiert:

3.4.1 Definition. Sei A eine endliche Menge von Aktionslabeln, P eine endliche Menge von Aktionslabeln für boolesche Ausdrücke und $Pr = \{p, \neg p \,|\, p \in P\}$ die Menge der positiven und negierten Aktionslabel für boolesche Ausdrücke. Dann ist ein *Action-NFA* $\mathcal{A}$ definiert als ein Tupel $\mathcal{A} = (Q, \Sigma \cup Pr \cup A, \delta, s, F)$ mit

- einer endlichen Zustandsmenge Q,
- einem Alphabet Σ,
- einer Transitionsfunktion $\delta : Q \times ((2^{\Sigma} \times 2^{Pr} \times 2^{A}) \cup \{\epsilon\}) \to 2^{Q}$,
- einem Startzustand $s \in Q$ und einer Menge $F \subseteq Q$ akzeptierender Zustände.

Ein *Action-DFA* ist ein *Action*-NFA ohne Epsilon-Transitionen, für den für jeden Zustand $q \in Q$, jedes Eingabezeichen $\sigma \in \Sigma$ und jede Teilmenge $S \subseteq Pr$ der Prädikate gilt:

$$|\bigcup_{t=(t_1,S,t_2)\in 2^{\Sigma}\times 2^{Pr}\times 2^{A}:\sigma\in t_1} \delta(q,t)| \leq 1$$

Nach der Definition können *Action*-NFAs lesende Transitionen sowie Epsilon-Transitionen besitzen. Bei Epsilon-Transitionen erfolgt der Übergang ohne die Verarbeitung eines Eingabezeichens, ohne dass dabei Aktionen getriggert oder semantische Prädikate überprüft werden können. Transitionen, die Eingabesymbole verarbeiten, bestehen neben einer Menge zulässiger Eingabesymbole in Form einer Zeichenklasse aus einer Menge semantischer Prädikate und einer Menge zu triggernder Aktionen. Die Transition schaltet nur, wenn das wenn das aktuelle Eingabezeichen mit der Zeichenklasse matcht und zugleich alle semantischen Prädikate der Transition erfüllt sind. Nach dem Übergang in die Folgezustände werden die zur Transition gehörenden Aktionen ausgeführt. Wie bei NFAs sind auch bei *Action*-NFAs mehrere Folgezustände für ein Transitionslabel möglich. Ein *Action*-DFA dagegen darf für jedes Eingabezeichen und jede Menge semantischer Prädikate maximal einen Folgezustand besitzen. Zudem sind in *Action*-DFAs keine Epsilon-Transitionen erlaubt.

Im weiteren Verlauf der Arbeit werden für *Action*-NFAs die folgenden Bezeichnungen verwendet: Ein *Transitionslabel* bezeichnet ein Tripel $t = (\sigma, p, a)$ mit $\sigma \in 2^{\Sigma}$, $p \in 2^{Pr}$ und $a \in 2^{A}$. Für die einzelnen Elemente von t werden die Bezeichnungen $char(t)$ für σ, $pred(t)$ für p und $act(t)$ für a verwendet. Eine Menge $S \subseteq Pr$ semantischer Prädikate heißt *konsistent*, wenn für alle Prädikate p aus P gilt, dass $p \notin S \vee \neg p \notin S$. Folglich ist eine Menge semantischer Prädikate genau dann konsistent, wenn alle Prädikate der Menge gemeinsam erfüllbar sind.

Als nächstes wird das Format der Spezifikationen des implementierten Parsergenerators vorgestellt.

3.5 Format von Spezifikationen

Das Format der Spezifikationen für den implementierten Parsergenerator orientiert sich am Eingabeformat von Ragel [13] und Snowfall [61]. Hinzu kommen kleinere Veränderungen, u. a. für die Platzhalter zur Codegenerierung.

Das Spezifikationsformat wird anhand der regulären Ausdrücke von Beispiel 3.3.1 vorgestellt. In Listing 3.1 sind Ausschnitte der Spezifikation mit VHDL-Coderahmen (außerhalb des in `%%{` und `}%%` eingeschlossenen Blocks) sowie der Definition des Parsers (innerhalb von `%%{` und `}%%`) für das Beispiel angegeben. Das Format zur Referenzierung der Aktionen in regulären Ausdrücken ist in Tabelle 3.1 angegeben, wobei für Zustandsaktionen der betretende Aktionstyp vor dem verlassenden Aktionstyp aufgeführt ist.

Tabelle 3.1: Format zur Referenzierung von Aktionen in Spezifikationen (nach [65])

Art der Aktion	Aktionstyp	Referenzierung
Transitionsaktion	Betretende Aktion	$>action$
	Aktion für alle Transitionen	$\$action$
	Beendende Aktion	$@action$
	Verlassende Aktion	$\%action$
Zustandsaktion	Alle Zustände	$\$\tilde{}action, \$*action$
	Innere Zustände	$<>\tilde{}action, <>*action$
	Startzustand	$>\tilde{}action, >*action$
	Akzeptierende Zustände	$\%\tilde{}action, \%*action$
	Alle Zustände außer Startzustand	$<\tilde{}action, <*action$
	Alle Zustände ohne Akzeptierende	$@\tilde{}action, @*action$

Listing 3.1: Spezifikation des Parsergenerators für Beispiel 3.3.1

```
%%{
  machine my_parser;
  action ver { version := version*10 + dataIn_Buffer; };
  ...
  Header = "V" ("1" [0-9] $ver | "2" [0-9]{2} $ver);
  Key = ([a-z] when !inLen [a-zA-Z0-9]*) $id;
  Length = [0-9]+ $len;
  Value = .* $val when inLen;
  main := Header (Key "␣" Length "=" Value %out)+
          0x01 when !inLen;
}%%

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.state_module_components.all;

entity patternMatcher is
  port(clk : in std_logic;
       reset : in std_logic;
       dataIn : in std_logic_vector(7 downto 0);
       dataIn_Valid : in std_logic;
       dataOut : out std_logic_vector(7 downto 0);
       dataOut_Valid : out std_logic);
```

```
end entity patternMatcher;

architecture rtl of patternMatcher is
  %% write globals;
begin
  %% write instances;
  process (clk, reset)
    %% write procDecls;
    variable version : integer range 0 to 99 := 0;
  begin
    if reset = '1' then
      %% write resetExec;
      version := 0;
      dataOut <= (others => '0');
      dataOut_Valid <= '0';
    elsif rising_edge(clk) then
      if dataIn_Valid = '1' then
        %% write exec;
      end if;
    end if;
  end process;
end architecture rtl;
```

Die Definition des Parsers besteht aus den verwendeten Aktionen, semantischen Prädikaten und den regulären Ausdrücken. In der angegebenen Spezifikation ist nur beispielhaft eine Aktion angegeben, die weiteren Aktionen und die benötigten semantischen Prädikate sind aus Platzgründen nicht mit aufgeführt. Im VHDL-Coderahmen wird als Rumpf für den Parser eine Entität definiert, deren Architektur den generierten Code des Parsergenerators aufnimmt. Daher sind neben dem partiellen VHDL-Code auch Platzhalter für die Codegenerierung enthalten.

Einige der Ein-/Ausgabeports des VHDL-Coderahmens werden für die Anbindung des generierten Pattern Matchers verwendet. Benötigt werden die Eingabeports `clk` als Taktsignal, `reset` für eine asynchrone Reset-Funktion, `dataIn` zur Übergabe der Eingabezeichen des Datenstroms und `dataIn_Valid` als 1-Bit Signal zur Markierung der Gültigkeit des Eingabezeichens. Die Ausgabeports werden insbesondere durch den Aktionscode bestimmt und sind damit für jeden Pattern Matcher individuell zu definieren.

Im VHDL-Coderahmen der Spezifikation sind folgende Platzhalter für den VHDL-Code des generierten Pattern Matchers erlaubt, die bei der Codegenerierung durch den Parsergenerator ersetzt werden:

write globals Platzhalter zur Definition von globalen Signalen, Datentypen oder Komponenten

write instances Platzhalter für die Instanzen der Basisbausteine

write procDecls Platzhalter im Deklarationsblock eines VHDL-Prozesses für Variablen, die für die Aktionsausführung oder die semantischen Prädikaten genutzt werden

write resetExec Platzhalter innerhalb eines VHDL-Prozesses zur Ausführung der Logik bei einem Reset (z. B. Zurücksetzen der semantische Prädikate)

write exec Platzhalter innerhalb eines VHDL-Prozesses zur Ausführung von Aktionen und Berechnung der semantischen Prädikate

Mit den erforderlichen Eingabeports und den Platzhaltern ergibt sich ein Grundgerüst für den VHDL-Coderahmen, welches in jeder funktionsfähigen Spezifikation des implementierten Parsergenerators enthalten sein sollte.

Im nächsten Kapitel werden ausgewählte Details der Implementierung des Parsergenerators genauer betrachtet. Insbesondere stehen dabei die umgesetzten Lösungen für die Herausforderungen und Schwierigkeiten der einzelnen Phasen im Vordergrund.

Kapitel 4

Implementierung des Parsergenerators

In diesem Kapitel werden ausgewählte Aspekte der Implementierung des Parsergenerators für das Pattern Matching auf FPGAs detaillierter erläutert. Dabei wird genauer auf die einzelnen Phasen des Parsergenerators eingegangen und es werden die umgesetzten Lösungen für die aufkommenden Herausforderungen der einzelnen Phasen dargestellt. Die Phasen des Parsergenerators werden im Folgenden der Reihe nach betrachtet, beginnend mit dem Parsen der Spezifikation bis hin zur Generierung der VHDL-Beschreibung.

4.1 Parsing einer Spezifikation

Mit dem Parsing der Spezifikation wird aus der Eingabe des Parsergenerators eine interne Datenstruktur gewonnen, die u. a. die regulären Ausdrücke enthält. Diese Datenstruktur wird in den nachfolgenden Phasen auf dem Weg zu einem Pattern Matcher für FPGAs weiterverarbeitet. Das Parsen der Spezifikation erfolgt mit Hilfe eines von ANTLR generierten Top-Down Parsers, in den ein Lexer integriert ist. Durch die unterschiedlichen Bestandteile einer Spezifikation des Parsergenerators wie beispielsweise der VHDL-Coderahmen, der Aktionscode oder die regulären Ausdrücke ergeben sich verschiedene Anforderungen an das Parsing: Reguläre Ausdrücke sollen zeichenweise geparst werden, wogegen Aktionscode oder der VHDL-Coderahmen in Blöcken eingelesen und verarbeitet werden können. Zudem soll eine fehlerhafte Spezifikation bereits beim Parsen erkannt werden und zum Abbruch der Ausführung des Parsergenerators führen, sodass in nachfolgenden Phasen nicht auf fehlerhafte oder unvollständige Datenstrukturen aus der Spezifikation geachtet werden muss.

Zentraler Bestandteil einer Spezifikation sind die regulären Ausdrücke für das Pattern Matching. Für diese können neben den Standardoperatoren wie der Konkatenation, der Vereinigung und der Wiederholung auch Aktionen und semantische Prädikate verwendet

werden. Die Operatoren besitzen die in Tabelle 4.1 angegebenen Prioritäten für die Bindungsstärke. Die Gruppierung hat dabei die höchste Priorität und die Vereinigung weist zusammen mit den Operatoren für Schnitt und Differenz die niedrigste Priorität auf. In der Regel können keine Operatoren der gleichen Priorität direkt miteinander kombiniert werden. Eine Ausnahme bildet die Vereinigung, um Ausdrücke der Form $\alpha_1|\alpha_2|\cdots|\alpha_n$ spezifizieren zu können. Mit der Angabe von Aktionen, die beim Matchen eines regu-

Tabelle 4.1: Prioritäten der unterstützten Operatoren für reguläre Ausdrücke

Priorität	Operatoren	Beschreibung
1	(α)	Gruppierung
2	!	Negation
3	? $*$ $+$ $\{n\}$ $\{n,m\}$ $\{n,\}$	(Bedingte) Wiederholung
4	$>$ \$ @ %	Transitionsaktionen
	$>\tilde{}$ \$$\tilde{}$ %$\tilde{}$ $<\tilde{}$ @$\tilde{}$ $<>\tilde{}$	Betretende Zustandsaktionen
	$>*$ \$$*$ %$*$ $<*$ @$*$ $<>*$	Verlassende Zustandsaktionen
	when when !	Semantische Prädikate
5	$\circ$	Konkatenation
6	$\mid$ & $-$ --	Vereinigung, Schnitt, (Scharfe) Differenz

lären (Teil-)Ausdrucks feuern, kann ein zu parsender Datenstrom gefiltert, komprimiert oder transformiert werden. Zusätzlich können mit semantischen Prädikaten die Transitionen des korrespondierenden Teilautomaten dynamisch zur Laufzeit des Parsing auf dem FPGA (de-)aktiviert werden, um einige kontextsensitive Datenströme zu parsen.

Der generierte ANTLR-Parser erzeugt für eine gültige Spezifikation einen Parsebaum, welcher nach dem Visitor-Entwurfsmuster [21] durchlaufen und in eine Datenstruktur zur internen Verarbeitung umgewandelt wird. Diese Datenstruktur enthält u. a. die Codeblöcke für Aktionen und semantische Prädikate sowie den VHDL-Coderahmen mit Platzhaltern für die Codegenerierung. Ein zentraler Teil dieser Datenstruktur ist die Repräsentation der regulären Ausdrücke für das Pattern Matching, die als Tokenlisten[1] abgespeichert werden. Für jeden regulären Ausdruck wird eine *Tokenliste* erzeugt, bei der es sich um eine erweiterte Form der in [73] vorgestellten Datenstruktur handelt. Die Token einer Tokenliste stellen die Teilausdrücke eines regulären Ausdrucks dar. Ein Token besteht aus den folgenden Feldern:

- *Wert*: Zeichenklasse oder Operator für Subausdruck, Vereinigung, Schnitt oder Negation
- *Prädikate*: Semantische Prädikate für den Teilausdruck

[1] Der Begriff *Token* ist hier nicht mit dem Tokenbegriff eines Lexers gleichzusetzen, sondern bezeichnet die einzelnen Teilausdrücke eines regulären Ausdrucks.

- *Aktionen*: Aktionen für den Teilausdruck
- *Wiederholung*: (bedingte) Wiederholung des Teilausdrucks
- *Nachfolger*: Nachfolger-Token für Konkatenation oder Vereinigung
- *Kinder*: Subausdruck oder ein bzw. zwei Operand(en) für den *Wert*

Über das Attribut *Wert* wird den Token ein Typ zugeordnet, der die Bedeutung der anderen Felder bestimmt. Beispielsweise enthält bei Token mit Vereinigungsoperator als Wert das Nachfolger-Feld das nächste Element der Vereinigung, wogegen der Nachfolger bei allen anderen Typen den nachfolgenden, konkatenierten Teilausdruck enthält. Eine Tokenliste ist eine verkettete Liste von Token, die sich durch die Nachfolger- und Kinder-Zeiger über mehrere Ebenen erstreckt. Ein Beispiel einer Tokenlsite ist in Abbildung 4.1 dargestellt, wobei die schwarzen Pfeile die Nachfolger und die roten Pfeile die Kinder beschreiben.

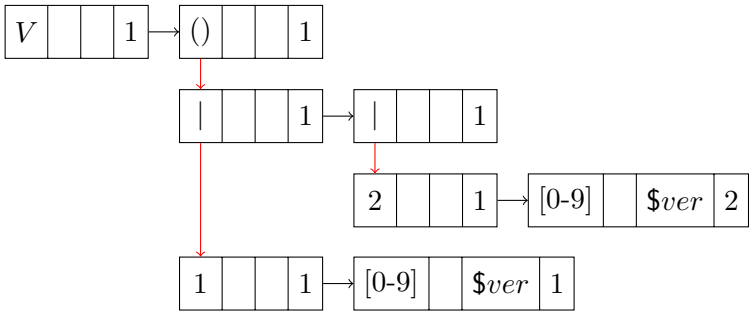


Abbildung 4.1: Tokenliste des regulären Ausdrucks $\alpha =$ `"V"` (`"1"` `[0-9]` ver | `"2"` `[0-9]{2}` ver)

Das Parsen der verschiedenen Bereiche einer Spezifikation erfolgt zeichen- bzw. blockweise mit dem generierten ANTLR-Parser, indem Parser und Lexer getrennt über Grammatiken definiert werden (weitere Details befinden sich in Anhang A). Dadurch können innerhalb des Lexers verschiedene *Modi* verwendet werden [50]. Ein Lexer-Modus entspricht einem Teillexer, der nur die dem jeweiligen Modus zugeordneten Token erkennt. Über verschiedene Trennzeichen wie dem Beginn der Definition des Parsers wird zwischen den einzelnen Modi des Lexers gewechselt. Die zeichen- bzw. blockweise Verarbeitung der Spezifikation korresponidert zu den Modi: Beispielsweise bestehen im Modus für reguläre Ausdrücke viele Token aus jeweils nur einem Zeichen, wogegen im Modus für Aktionscode die Token mehrere Zeichen oder sogar ganze Zeilen beschreiben.

Für gültige Spezifikationen bildet die interne Datenstruktur die Ausgabe der Parsingphase und wird an die nächste Phase zur Erzeugung der endlichen Automaten übergeben. Falls eine Spezifikation nicht zu dem geforderten Eingabeformat passt, wird die weitere Verarbeitung des Parsergenerators direkt abgebrochen, um nachfolgende Phasen nicht unnötig um Fehlerbehandlungen aufblähen zu müssen. Fehler sind beispielsweise ungültige Angaben regulärer Ausdrücke oder die Verwendung nicht definierter Aktionen. Für

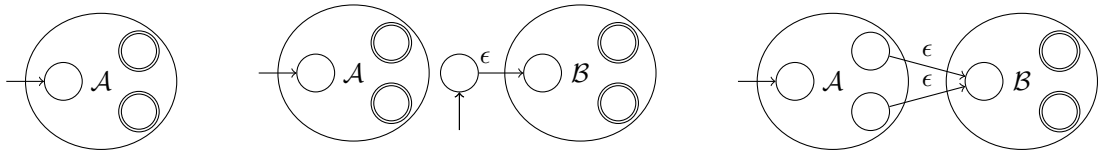
den Abbruch ist eine Fehlerbehandlungsstrategie [50] im generierten ANTLR-Parser integriert, die bei jedem Parsefehler eine `Exception` in Java wirft und damit zum Abbruch der Verarbeitung durch den Parsergenerator führt.

Auf das Parsen der Spezifikation folgt die Umwandlung der regulären Ausdrücke in endliche Automaten, die im folgenden Abschnitt erläutert wird.

4.2 Umwandlung in endliche Automaten

Die Umwandlung der regulären Ausdrücke der Tokenlisten in nichtdeterministische endliche Automaten erfolgt mit einer Variante des in Kapitel 2 betrachteten Verfahrens. Dabei besteht die Herausforderung in der Einbettung von Aktionen und semantischen Prädikaten in die Automaten, sodass deren Semantik korrekt wiedergegeben wird. Zusätzlich sind die theoretischen Grundlagen für NFAs bei der Umwandlung von regulären Ausdrücken mit Operatoren wie Negation, Schnitt oder (scharfer) Differenz zu berücksichtigen.

Die regulären Ausdrücke werden mit dem in [72, 73] vorgestellten modifizierten McNaughton-Yamada Verfahren in NFAs umgewandelt. Im Unterschied zu dem in Kapitel 2 skizzierten Verfahren werden dabei weniger Epsilon-Transitionen erzeugt. Das Verfahren ist wie in [73] über zwei rekursive Methoden implementiert, die verschiedene Felder (wie den *Wert* oder die *Wiederholung*) der Token einer Tokenliste bei der Umwandlung berücksichtigen. Die Einbettung von Aktionen erfordert dabei die Erweiterung des Verfahrens: Eine Tokenliste $T = t_0, \ldots, t_n$ für den regulären Ausdruck $\alpha\beta$ wird schrittweise in einen Automaten umgewandelt. Die bereits umgewandelten Token $t_0, \ldots, t_{n-1}$ des Ausdrucks α bilden den Teilautomaten $\mathcal{A}$. Das Token t_n mit dem Ausdruck β wird für eine korrekte Wiedergabe der Aktionssemantik zunächst getrennt von $\mathcal{A}$ in den NFA $\mathcal{B}$ umgewandelt. Nach Entfernen der Epsilon-Transitionen in $\mathcal{B}$ werden die Aktionen des Token t_n in $\mathcal{B}$ eingebettet. Anschließend werden im Fall von $\mathcal{L}(\alpha) \neq \emptyset$ die akzeptierenden Zustände von $\mathcal{A}$ mit dem Startzustand von $\mathcal{B}$ über Epsilon-Transitionen verbunden, um die Teilautomaten zu verknüpfen. Das Verfahren ist als Skizze in Abbildung 4.2 dargestellt.



(a) Beginn mit $\mathcal{A}$ (b) Konstruktion von $\mathcal{B}$ (c) Konkatenation von $\mathcal{A}$ und $\mathcal{B}$

Abbildung 4.2: Implementiertes Verfahren zur Umwandlung regulärer Ausdrücke in Automaten

Für Token von Ausdrücken mit bedingten Wiederholungen der Form $\gamma\{n, m\}$ sind in [73] zwei mögliche Realisierungen angegeben (siehe Abbildung 4.3), die zu einer unterschiedlichen Anzahl ein- bzw. ausgehender Transitionen in den erzeugten Automaten

führen - und damit die Schaltung auf dem FPGA beeinflussen. Von den letzten $m - n$ Kopien des NFAs $\mathcal{C}$ für γ können beim Matchen eines Wortes $w \in \mathcal{L}(\gamma)^p$ mit $n < p < m$ die ersten oder die letzten $m - p$ Kopien von $\mathcal{C}$ im optionalen Teil übersprungen werden. Die Realisierung der Einbettung von Aktionen wird durch die Nutzung der vorderen p

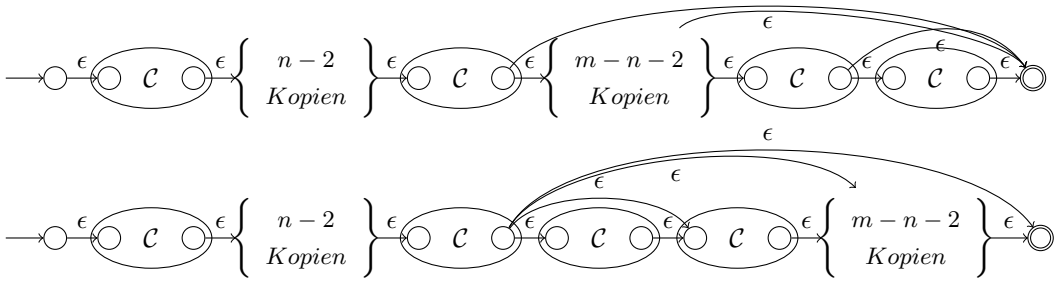


Abbildung 4.3: Varianten zur Umwandlung von Teilausdrücken der Form $\gamma\{n, m\}$ in NFAs [73]

Kopien von $\mathcal{C}$ (darunter die ersten $p - n$ Kopien im optionalen Teil) zum Matchen von w und dem Überspringen der folgenden, nicht benötigten $m - p$ Kopien vereinfacht.

Bei der Umwandlung der regulären Ausdrücke in endliche Automaten sind die Operatoren für Schnitt, Negation und Differenz gesondert zu betrachten, da diese nicht direkt mit dem Verfahren von McNaughton-Yamada abgedeckt werden. Die Besonderheiten der Implementierung dieser Operatoren werden im folgenden Abschnitt erläutert.

4.2.1 Schnitt, Negation und Differenz

Die Implementierung des Parsergenerators für die regulären Operatoren für Schnitt, Negation und Differenz basiert auf Produktautomaten für den Schnitt sowie der Determinisierung und Komplementierung für die Negation. Die Differenzoperatoren sind als eine Kombination aus Schnitt und Negation realisiert. Dabei werden die theoretischen Grundlagen bzgl. Schnitt und Negation aus Kapitel 2 auf *Action*-NFAs übertragen.

Schnitt Den Schnitt der Sprachen zweier regulärer Ausdrücke α und β mit *Action*-NFAs $\mathcal{A}$ und $\mathcal{B}$ beschreibt der Produktautomat $\mathcal{A}_{prod}$ mit $\mathcal{L}(\mathcal{A}_{prod}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$. Durch die Verwendung von Zeichenklassen, Aktionen und semantischen Prädikaten muss die Produktautomaten-Konstruktion aus [30] angepasst werden. Der Produktautomat für zwei *Action*-NFAs ist formal wie folgt definiert:

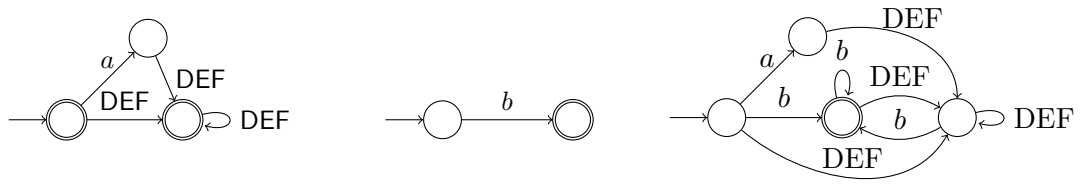
4.2.1 Definition. Seien $\mathcal{A} = (Q_1, \Sigma \cup Pr \cup A, \delta_1, s_1, F_1)$ und $\mathcal{B} = (Q_2, \Sigma \cup Pr \cup A, \delta_2, s_2, F_2)$ zwei *Action*-NFAs und sei $S \subseteq Q_1 \times Q_2$ eine Teilmenge des Kreuzproduktes der Zustandsmengen. Dann ist der Produktautomat $\mathcal{A}_{prod}$ von $\mathcal{A}$ und $\mathcal{B}$ definiert als $\mathcal{A}_{prod} = (Q_1 \times Q_2, \Sigma \cup Pr \cup A, \delta, (s_1, s_2), S)$, wobei $\forall q_1 \in Q_1 \forall q_2 \in Q_2 \forall t_1, t_2 \in (2^\Sigma \times 2^{Pr} \times 2^A)$ mit $char(t_1) \cap char(t_2) \neq \emptyset$ sowie miteinander konsistenten $pred(t_1)$ und $pred(t_2)$ gilt:

$$\delta((q_1, q_2), (char(t_1) \cap char(t_2), pred(t_1) \cup pred(t_2), act(t_1) \cup act(t_2)) = \delta_1(q_1, t_1) \times \delta_2(q_2, t_2)$$

Aus der Definition des Produktautomaten für *Action*-NFAs lässt sich direkt ein Verfahren zur Konstruktion von $\mathcal{A}_{prod}$ ableiten. Zunächst werden die *Action*-NFAs der beiden regulären Ausdrücke α und β berechnet und deren Epsilon-Transitionen entfernt. Anschließend wird der Produktautomat anhand der Definition berechnet. Dabei werden bei der Implementierung nur die Zustände des Produktautomaten berechnet, die vom Startzustand (s_1, s_2) aus erreichbar sind. Die Berechnung von $\mathcal{A}_{prod}$ beginnt beim Startzustand (s_1, s_2) und arbeitet mit Hilfe einer Warteschlange alle Zustände ab, die von dort aus erreichbar sind. Die Menge der akzeptierenden Zustände für den Schnitt von α und β entspricht dem Kreuzprodukt aus F_1 und F_2, sodass nur Wörter akzeptiert werden, die gleichzeitig in der Sprache von $\mathcal{A}$ und der Sprache von $\mathcal{B}$ enthalten sind.

Negation Die Implementierung des Negationsoperators für reguläre Ausdrücke kommt durch die theoretischen Grundlagen aus Kapitel 2 nicht ohne die Determinisierung der nichtdeterministischen endlichen Automaten aus. Für einen zu negierenden regulären Ausdruck α wird zunächst der korrespondierende *Action*-NFA $\mathcal{A}$ erzeugt, der dann determinisiert wird und nach Vertauschung akzeptierender und nicht akzeptierender Zustände einen Automaten für $\Sigma - \mathcal{L}(\alpha)$ bildet [30]. Für die Determinisierung von $\mathcal{A}$ werden zunächst die Epsilon-Transitionen entfernt. Anschließend werden die Zustände und Transitionen des *Action*-DFAs $\mathcal{A}_{det}$ mit Hilfe einer angepassten Teilmengenkonstruktion berechnet. Jede Teilmenge P von Zuständen von $\mathcal{A}$ bildet einen Zustand p von $\mathcal{A}_{det}$. Die Berechnung der Transitionen von p berücksichtigt dabei wie die Konstruktion des Produktautomaten die Zeichenklassen, semantischen Prädikate und Aktionen in den Transitionslabeln. Für P wird in einem vorbereitenden Schritt für jedes Transitionslabel t_1, welches von einem der Zustände aus P ausgeht, eine Partition $\Pi(t_1)$ von $char(t_1)$ anhand der Schnittmengen relevanter Zeichenklassen gebildet. Dabei werden alle Transitionen $t_2 \neq t_1$ mit $pred(t_1) = pred(t_2)$ betrachtet, die einen Zustand von P verlassen und jeder Block $X \in \Pi(t_1)$ der Partition wird durch $X \cap char(t_2)$ und $X - char(t_2)$ ersetzt. Somit werden die Zeichen aus $char(t_1)$ durch die Transitionen von P in verschiedene Mengen partitioniert. Jeder Zeichenklasse X der Partition $\Pi(t_1)$ ist eine Menge von Nachfolgezuständen des NFAs $\mathcal{A}$ sowie eine Menge von Aktionen zugeordnet. Nach der Vorberechnung werden die Transitionen zum Zustand p von $\mathcal{A}_{det}$ mit den Aktionen und semantischen Prädikaten hinzugefügt, wobei die Mengen der Nachfolgezustände in die jeweiligen Teilmengenzustände des DFAs übersetzt werden. Falls der Zustand p nicht für alle Zeichen aus Σ einen Folgezustand besitzt, erhält p für die fehlenden Zeichen eine Transition zum Zustand, der die leere Menge repräsentiert - dieser wird wegen möglicher doppelter Negation benötigt.

Wie bei der Konstruktion des Produktautomaten werden auch bei der Determinisierung in der implementierten Version nur die vom Startzustand von $\mathcal{A}_{det}$ aus erreichbaren Zustände berechnet. Anschließend werden für die Realisierung des Negationsoperators die akzeptierenden und nicht akzeptierenden Zustände von $\mathcal{A}_{det}$ vertauscht.



(a) DFA $\mathcal{A}_1$ für $\alpha_1 =$ `!"a"` (b) NFA $\mathcal{A}_2$ für $\alpha_2 =$ `"b"` (c) DFA $\mathcal{A}$ für $\alpha =$ `!"a" "b"`

Abbildung 4.4: Beispiel: Verwendung von Default-Transitionen für den Negationsoperator

Im Unterschied zu Ragel [13] werden bei der Determinisierung von NFAs keine Default-Transitionen verwendet, sondern die fehlenden Transitionen für die Zustände des DFAs explizit ergänzt. Hintergrund ist, dass ohne die im Allgemeinen fehlende Determinisierung des kompletten NFAs eine korrekte Auflösung der Default-Transitionen nur schwer realisierbar ist. Dies wird im folgenden Beispiel sichtbar.

4.2.2 Beispiel. Betrachtet wird der reguläre Ausdruck $\alpha =$ `!"a" "b"` über dem Alphabet $\Sigma = \{a, b, c\}$. Der DFA $\mathcal{A}_1$ mit Default-Transitionen für $\alpha_1 =$ `!"a"` und der NFA $\mathcal{A}_2$ für $\alpha_2 =$ `"b"` sind in den Abbildungen 4.4(a) und 4.4(b) dargestellt. An $\mathcal{A}_1$ kann der Automat $\mathcal{A}_2$ nicht direkt angefügt werden, ohne die Semantik der Default-Transitionen zu stören. Beispielsweise führt das Verbinden der akzeptierenden Zustände von $\mathcal{A}_1$ mit dem Startzustand von $\mathcal{A}_2$ zum Fehlen des Wortes $w = bcb$ in der Sprache des resultierenden Automaten. Ein korrekter Automat mit Default-Transitionen für $\alpha =$ `!"a" "b"` ist in Abbildung 4.4(c) dargestellt. Das Verbinden von DFAs mit Default-Transitionen und NFAs für die Konkatenation ist somit nicht problemlos möglich und ohne vollständige Determinisierung im Allgemeinen nicht bzw. nur schwer berechenbar.

Differenz Der Differenz-Operator für reguläre Ausdrücke besteht aus einer Kombination des Schnitts und der Negation. Für die Sprachen von zwei Automaten $\mathcal{A}$ und $\mathcal{B}$ gilt $\mathcal{L}(\mathcal{A}) - \mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{A}) \cap \overline{\mathcal{L}(\mathcal{B})}$ [30]. Dadurch kann der Differenz-Operator durch einen Schnitt-Operator ersetzt werden, bei dem der zweite Ausdruck negiert wird. Diese Ersetzung erfolgt bereits beim Erzeugen der Tokenlisten aus dem Parsebaum einer Spezifikation innerhalb des Parsingphase. Ähnlich sieht es bei der scharfen Differenz aus. Die scharfe Differenz α `--` β ist äquivalent zum regulären Ausdruck $\alpha - (.^*\beta.^*)$ [65], sodass auch hier die Ersetzung bereits beim Bilden der Tokenlisten stattfindet. Damit müssen die Differenz-Operatoren innerhalb des Algorithmus zur Umwandlung von regulären Ausdrücken in endliche Automaten nicht gesondert betrachtet werden.

Eine zentrale Herausforderung des Umwandlungsverfahrens ist die Einbettung der Aktionen und semantischen Prädikate. Diese Einbettung wird in den beiden folgenden Abschnitten beschrieben.

4.2.2 Einbettung von Aktionen

Im Umwandlungsprozess werden Aktionen über deren Aktionslabel als Teil der Transitionslabel und der Zustände (zur späteren Übernahme in anliegende Transitionen) in die *Action*-NFAs integriert. Dabei können an eine Transition oder einen Zustand mehrere Aktionen gebunden werden. Die Schwierigkeit besteht in der Bestimmung der für jede Art und jeden Typ von Aktion korrekten Transitions- bzw. Zustandsmenge des Automaten.

Im Unterschied zur Mustererkennung in Software werden bei der Ausführung eines Pattern Matchers auf Hardware wie FPGAs alle gefeuerten Aktionen parallel ausgeführt. Dadurch ist die Einhaltung einer Reihenfolge für Aktionen in Transitionslabeln und Zuständen eines Automaten ebenso überflüssig wie die in Ragel [13] enthaltene Unterscheidung von Transitionsaktionen und Zustandsaktionen bzgl. des Ausführungszeitpunktes. Somit werden in der Implementierung die Aktionslabel in den Transitionslabeln und Zuständen von Automaten stets in ungeordneten Mengen abgespeichert.

Für das Token t_n mit den regulären Ausdruck β der Tokenliste $T = t_0, \ldots, t_n$ sind neben den in t_n enthaltenen Aktionen auch einige Aktionen des bereits in $\mathcal{A}$ umgewandelten Teilausdrucks α zu berücksichtigen. Dies betrifft die verlassenden Transitions- und Zustandsaktionen auf den akzeptierenden Zuständen von $\mathcal{A}$. Zur korrekten Wiedergabe der Aktionssemantik werden die akzeptierenden Zustände von $\mathcal{A}$ nach diesen beiden Aktionstypen gruppiert. Für jede Zustandsmenge Z der Gruppierung wird eine Kopie des Automaten $\mathcal{B}$ für β erzeugt. Dadurch entfällt eine, ohne mehrfache Kopien erforderliche, gesonderte Betrachtung der vom Startzustand von $\mathcal{B}$ ausgehenden Transitionen für die Aktionen von $\mathcal{A}$. Dies gilt ggf. auch für folgende Teilausdrücke, wenn $\mathcal{L}(\beta)$ den Leerstring enthält.

Nach der Berechnung des Teilautomaten $\mathcal{B}$ für β und dem Entfernen der darin enthaltenen Epsilon-Transitionen werden die in t_n enthaltenen Aktionen sowie die Aktionen der Zustandsmenge Z ausgewertet und in $\mathcal{B}$ eingebettet. Das Entfernen der Epsilon-Transitionen vereinfacht die Aktionseinbettung, da beispielsweise Transitionen von Schleifen niemals in den Startzustand zurückkehren und somit u. a. betretende Aktionen zu weniger Sonderfällen führen. Die verlassenden Aktionen der gruppierten Zustandsmenge Z von $\mathcal{A}$ werden auf alle Transitionen angewendet, die den Startzustand von $\mathcal{B}$ verlassen. Die Transitionsaktionen des Token t_n werden wie folgt berücksichtigt: Betretende Aktionen werden auf alle Transitionen angewendet, die den Startzustand von $\mathcal{B}$ verlassen, sodass sie bzgl. β einmalig ausgeführt werden. Die Aktionen für alle Transitionen werden an alle Transitionen von $\mathcal{B}$ angefügt und beendende Aktionen an alle Transitionen, die in die akzeptierenden Zustände von $\mathcal{B}$ führen. Die in t_n enthaltenen verlassenden Aktionen finden erst für folgende, per Konkatenation angehängte Teilausdrücke Anwendung und werden daher in den akzeptierenden Zuständen von $\mathcal{B}$ zwischengespeichert. Damit werden sie bei der Berechnung des Automaten für folgende Teilausdrücke berücksichtigt (analog zu den aus $\mathcal{A}$ stammenden

und für $\mathcal{B}$ geltenden Aktionen). Für den Fall, dass $\mathcal{B}$ den Leerstring akzeptiert, erhalten betretende Aktionen aus t_n ebenfalls die Semantik von verlassenden Aktionen für den Startzustand. Sie werden dann zusätzlich als verlassende Aktionen im Startzustand von $\mathcal{B}$ abgespeichert.

Die Zustandsaktionen des Tokens t_n werden in den Zuständen von $\mathcal{B}$ abgespeichert und von dort auf die Label ein- bzw. ausgehender Transitionen übertragen. Beispielweise werden betretende Zustandsaktionen auf dem Startzustand von $\mathcal{B}$ bei der Konkatenation mit $\mathcal{A}$ auf die in akzeptierende Zustände von $\mathcal{A}$ führenden Transitionen übertragen. Bei der Semantik der Zustandsaktionen, die von Ragel übernommen wird, sind einige Besonderheiten zu beachten, die Auswirkungen auf die Einbettung haben:

- In Vereinigungen gelten betretende und verlassende Zustandsaktionen für den Startzustand in allen Alternativen, wenn sie für mindestens eine Alternative definiert werden.

- Schleifen in Automaten verlassen den Startzustand, ohne diesen je wieder zu betreten. Die Häufigkeit der Ausführung von Zustandsaktionen bei Schleifen ist davon abhängig, ob diese innerhalb oder außerhalb der Schleife angegeben werden. Beispielweise werden Zustandsaktionen für den Startzustand in jeder Iteration ausgeführt, wenn sie innerhalb der Schleife spezifiziert werden; bei der Spezifikation außerhalb der Schleife erfolgt die Ausführung nur vor bzw. während der ersten Iteration. Verlassende Zustandsaktionen, die den Startzustand ausschließen, feuern in Schleifen bei allen Transitionen mit Ausnahme der ersten, betretenden Transition.

- Zustandsaktionen für akzeptierende Zustände gelten auch für den Startzustand, sofern dieser akzeptierend ist und nicht explizit durch die jeweilige Zustandsmenge ausgeschlossen wird. Entsprechend gelten Zustandsaktionen für den Startzustand nicht, wenn dieser akzeptierend ist und akzeptierende Zustände ausgeschlossen sind.

- Verlassende Zustandsaktionen auf dem Startzustand von $\mathcal{B}$ betreffen auch Teilautomaten von Suffixen von α, wenn diese den Leerstring akzeptieren. Schließlich wird der Startzustand von $\mathcal{B}$ für einen solchen Teilautomaten verlassen (um später wieder zurückzukehren) und die Zustandsaktionen feuern entsprechend.

- Verlassende Zustandsaktionen auf den akzeptierenden Zuständen werden in den darauffolgenden Teilausdrücken genau einmal ausgeführt, auch wenn diese mit Schleifen beginnen.

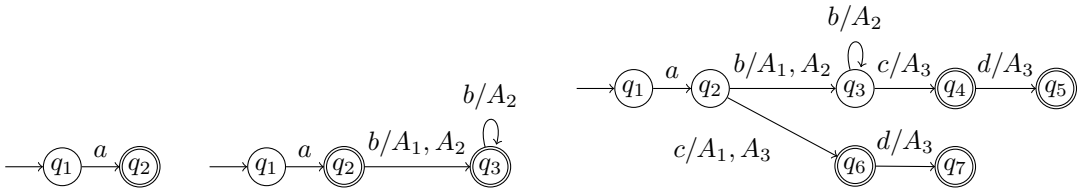
In der Implementierung der Einbettung durch das verwendete Umwandlungsverfahren werden bei Konkatenationen von Teilautomaten mit Hilfe von Epsilon-Transitionen einige Zustandsaktionen auf andere Zustände kopiert. Dazu kommt die nachträgliche Berücksichtigung von Zustandsaktionen beim Entfernen von Epsilon-Transitionen, da eine konsistente Handhabung über Epsilon-Transitionen hinweg schwieriger zu realisieren wäre.

Das folgende Beispiel verdeutlicht die Einbindung der Aktionen in den Umwandlungsprozess für reguläre Ausdrücke.

4.2.3 Beispiel. Betrachtet wird der reguläre Ausdruck

$$\alpha = \texttt{"a"}\ \%A_1\ \texttt{"b"}^*\ \$A_2\ (\texttt{"c"}\ \texttt{"d"}?)\ {<}\tilde{}\ A_3$$

mit Aktionen im Format einer Spezifikation des implementierten Parsergenerators. In Abbildung 4.5 ist die schrittweise Konstruktion des NFAs für α skizziert. Bei der Umwandlung wird die Aktion A_1 im Zustand q_2 im ersten Schritt als verlassende Aktion gespeichert und dann im zweiten Schritt auf die erste Transition der Schleife für b angewendet. Im dritten



(a) Nach `"a"` $\%A_1$ (b) Nach `"a"` $\%A_1$ `"b"`* $\$A_2$ (c) Nach `"a"` $\%A_1$ `"b"`* $\$A_2$ (`"c"` `"d"`?) ${<}\tilde{}\ A_3$

Abbildung 4.5: Beispiel: Konstruktion von NFAs mit Aktionen

Schritt liefert die Gruppierung der Zustände q_2 und q_3 jeweils zwei einelementige Mengen, sodass zwei Kopien des NFAs für `"c"` `"d"`? erzeugt werden. Bei der an q_2 angefügten Kopie muss wieder die Aktion A_1 berücksichtigt werden. In den Zuständen q_4, q_5, q_6 und q_7 wird die Aktion A_3 als eingehende Aktion gespeichert - auch wenn sie nicht mehr benötigt wird.

Im nächsten Abschnitt wird die Einbettung der semantischen Prädikate in die Transitionslabel der Automaten erläutert. Im Gegensatz zur Einbettung der Aktionen treten dabei aber keine Sonderfälle auf.

4.2.3 Einbettung von semantischen Prädikaten

Die semantischen Prädikate im Token t_n beziehen sich auf alle Transitionen des zum regulären Ausdruck β gehörenden *Action*-NFAs $\mathcal{B}$. Die Prädikate werden dabei stets konjunktiv verknüpft, sodass disjunktive Bedingungen innerhalb eines Prädikates realisiert werden müssen. Bei der Umwandlung der regulären Ausdrücke in endliche Automaten werden die semantischen Prädikate ebenfalls in den Transitionslabeln abgespeichert. Nach dem Erzeugen des Automaten $\mathcal{B}$, dem Entfernen der Epsilon-Transitionen und dem Einbetten der Aktionen werden die semantischen Prädikate aus t_n an alle Transitionen von $\mathcal{B}$ angefügt. Sonderfälle sind dabei nicht zu berücksichtigen. Da die semantischen Prädikate als ungeordnete Mengen in den Transitionslabeln abgespeichert werden, muss weder auf Dopplungen noch auf deren Reihenfolge Rücksicht genommen werden. Schließlich werden die Prädikate bei Realisierung in Hardware parallel ausgewertet.

Bei Verwendung von semantischen Prädikaten für das Pattern Matching auf FPGAs ist zu beachten, dass die booleschen Bedingungen in einem Taktzyklus berechnet werden und deren Auswertung erst im folgenden Taktzyklus für die (De-)Aktivierung von Transitionen berücksichtigt wird. Folglich ist bei der Definition von Prädikaten, die Parameter auswerten, welche über Aktionen verändert werden, genauer auf die Randfälle zu achten. Zudem kann das aktuelle Eingabezeichen in semantischen Prädikaten nicht genutzt werden, um Transitionen des Automaten für dieses Zeichen zu (de-)aktivieren. Diese Beschränkung lässt sich ggf. mit einer Änderung der Generierung der VHDL-Beschreibungen beseitigen.

Nach dem Erzeugen der endlichen Automaten werden in der nächsten Phase einige Optimierungen für das Pattern Matching auf FPGAs berechnet.

4.3 Optimierung des Pattern Matchings

Mit den Optimierungen des Parsergenerators für das Pattern Matching werden Einsparungen von Hardwareressourcen auf einem FPGA erreicht, sodass z. B. zusätzliche Automaten oder weitere, verarbeitende Logik auf dem FPGA platziert werden können. Die Optimierungsphase teilt sich dabei in zwei Schritte auf: Zunächst erfolgt die erste Stufe der Optimierung, welche unabhängig von FPGAs als Zielplattform ist und sich auf die Automaten selbst beschränkt. Darunter fällt u. a. die Reduzierung der Automatengröße. Anschließend werden in einer zweiten Optimierungsstufe gezielt Zusatzinformationen für die Codegenerierung berechnet, mit denen eine effizientere Abbildung der Automaten auf die Ressourcen des FPGAs möglich ist. Ein Beispiel ist die Berechnung von Informationen zur Verwendung von Lookup-Tabellen als Shiftregister für beschränkte Wiederholungen.

Die beiden Stufen der Optimierungsphase werden nacheinander vorgestellt.

4.3.1 Optimierungen für NFAs

Die erste Optimierungsstufe des Parsergenerators bezieht sich auf die endlichen Automaten an sich und verfolgt das Ziel der Reduzierung der Automatengröße, also die Verringerung der Anzahl an Zuständen und Transitionen. Dazu werden schrittweise verschiedene algorithmische Verfahren auf Automatenebene eingesetzt.

Im ersten Schritt der Optimierungsstufe werden alle Zustände eines *Action*-NFAs entfernt, von denen aus kein akzeptierender Zustand erreichbar ist. Solche Zustände können bei der Automatenkonstruktion entstehen, wenn die Operatoren für Negation und Differenz ineinander verschachtelt werden. Diese Zustände werden u. a. auch deswegen entfernt, weil sie bei Verfahren zur Reduzierung der Automatengröße stören.

Im zweiten Schritt wird die Größe der *Action*-NFAs reduziert. Dazu sind alternativ zwei verschiedene Verfahren implementiert. Eine Möglichkeit besteht in der Anwendung der in Kapitel 2 vorgestellten Äquivalenz-basierten Reduzierung. In der implementierten

Version wird das Verfahren zur Reduzierung iterativ angewendet [32]: Für einen Automaten $\mathcal{A}$ werden die Mengen Π_L und Π_R der Äquivalenzklassen von $\equiv_L$ und $\equiv_R$ berechnet und anschließend wird die Anzahl der Zustände von $\mathcal{A}$ durch Zusammenfassen von Zuständen anhand einer optimalen Lösung für Π_L und Π_R verringert. Sofern der entstandene Automat $\mathcal{A}_1$ nicht mehr $\mathcal{A}$ entspricht (also mindestens zwei Zustände von $\mathcal{A}$ zu einem Zustand in $\mathcal{A}_1$ zusammengefasst wurden), werden die Partitionen Π_L^1 und Π_R^1 für $\mathcal{A}_1$ berechnet, die zu einem reduzierten Automaten $\mathcal{A}_2$ führen. Dieses Verfahren wird solange fortgesetzt, bis keine weitere Reduzierung des Automaten mehr möglich ist. Für die effiziente Berechnung der Relationen $\equiv_L$ und $\equiv_R$ wird eine erweiterte Version des Verfahrens von Paige und Tarjan [18, 48] zur Verfeinerung von Partitionen bzgl. einer Menge von Relationen verwendet (für weitere Details siehe Anhang B). Der implementierte Algorithmus berechnet die Lösung des *Multiple Relational Coarsest Partitioning Problems*[2] für Automaten, wobei jedes Transitionslabel des NFAs eine Relation definiert [31].

Neben der Äquivalenz-basierten Reduzierung kann alternativ die Quasiordnung-basierte[3] Reduzierung nach [7, 8] zur Verringerung der Automatengröße verwendet werden. Vergleichbar mit den Äquivalenzrelationen $\equiv_L$ und $\equiv_R$ des vorherigen Verfahrens werden für dieses Verfahren zwei Quasiordnungen $\subseteq_L$ und $\subseteq_R$ nach den Axiomen aus Definition 2.1.6 definiert, wobei die Äquivalenz durch eine Quasiordnung zu ersetzen ist. Aus $p \subseteq_L q$ bzw. $p \subseteq_R q$ folgen dann die Teilmengenbeziehungen $\mathcal{L}_L(\mathcal{A}, p) \subseteq \mathcal{L}_L(\mathcal{A}, q)$ bzw. $\mathcal{L}_R(\mathcal{A}, p) \subseteq \mathcal{L}_R(\mathcal{A}, q)$ für die links- bzw. rechtsseitigen Sprachen. Die Umkehrung gilt analog zur Äquivalenz-basierten Reduzierung auch hier nicht, sodass lediglich Teilmengen der Zustände mit Teilmengenbeziehung der links- bzw. rechtsseitigen Sprachen berechnet werden. Im Verhältnis zur Äquivalenz $\equiv_R$ gilt, dass aus $p \equiv_R p$ zugleich $p \subseteq_R p$ und $q \subseteq_R p$ folgt - umgekehrt ist dies jedoch nicht zwingend der Fall [32]. Nach Berechnung von $\subseteq_L$ und $\subseteq_R$ können zwei Zustände p und q zusammengefügt werden, wenn mindestens eine der folgenden drei Bedingungen gilt [32]:

1. $p \subseteq_L q$ und $q \subseteq_L p$
2. $p \subseteq_R q$ und $q \subseteq_R p$
3. $p \subseteq_L q$, $p \subseteq_R q$ und $\mathcal{L}(\mathcal{A}, p, p) = \{\epsilon\}$

Dabei sei $\mathcal{L}(\mathcal{A}, p, p) = \mathcal{L}((Q, \Sigma \cup Pr \cup A, \delta, p, \{p\})$. Nach dem Zusammenfassen von p und q anhand der ersten beiden Fälle müssen $\subseteq_L$ und $\subseteq_R$ aktualisiert werden, um die Teilmengenbeziehungen für $\mathcal{L}_L$ und $\mathcal{L}_R$ zu erhalten [32]: Wenn die erste Bedingung zutrifft und p, q zu q zusammengefasst werden, müssen aus $\subseteq_R$ alle Zustandspaare (q, r) entfernt werden, für die $p \not\subseteq_R r$ gilt. Für die zweite Bedingung gilt dies analog für $\subseteq_L$.

[2] Multiple Relational Coarsest Partitioning Problem [18, 48]: Für eine Partition Π von Q und eine Menge $E_1, \ldots, E_k$ von binären Relationen über Q wird anhand von $E_1, \ldots, E_k$ die gröbste, stabile Verfeinerung von Π berechnet.

[3] Eine Quasiordnung (engl. *Preorder*) ist eine reflexive, transitive binäre Relation.

Die Berechnung der Relationen $\subseteq_L$ und $\subseteq_R$ erfolgt nach einer angepassten Version von Algorithmus 2.1. Für einen Automaten mit n Zuständen und m Transitionen gibt es auch einen Algorithmus mit $\mathcal{O}(|\Sigma| \times mn)$ Laufzeit, welcher aber $\mathcal{O}(|\Sigma| \times n^2)$ Speicherplatz benötigt [31] und damit nicht im implementierten Parsergenerator enthalten ist. Beim Zusammenfügen der Zustände ist im Unterschied zur Äquivalenz-basierten Reduzierung die Berechnung einer optimalen Lösung im Allgemeinen ein NP-vollständiges Problem und damit nicht in polynomieller Zeit möglich [32]. Das implementierte Verfahren geht somit die drei Bedingungen nacheinander durch und fügt die Zustände schrittweise zusammen.

Die beiden implementierten Verfahren zur Reduzierung der Automatengröße erkennen Duplikate von Teilautomaten, die durch die Berücksichtigung der verlassenden Transitions- und Zustandsaktionen auf akzeptierenden Zuständen entstehen und fassen diese, soweit möglich, zusammen. Dadurch entstehen durch das Konstruktionsverfahren der *Action*-NFAs keine unnötigen Nachteile bzgl. der Größe der zu synthetisierenden Automaten.

Ein drittes Verfahren zur Reduzierung der Automatengröße von NFAs wird in [2, 3] vorgestellt und basiert auf einer modifizierten Teilmengenkonstruktion. Allerdings ergibt die Evaluation aus [39], dass dieses Verfahren nur zu einer geringen Ressourceneinsparung führt. Somit ist dieses Verfahren zur Reduzierung der Automatengröße gar nicht als Teil des Parsergenerators implementiert. Innerhalb des Parsergenerators sind auch keine Verfahren zur vollständigen Minimierung von NFAs implementiert, da diese für Automaten mit n Zuständen selbst mit Heuristiken Laufzeiten von $2^{\mathcal{O}(n^2)}$ aufweisen können [42] und damit bereits für kleinere Automaten nicht mehr praktikabel sind. Zudem zeigen Košař et al. in [39], dass die Äquivalenz- und die Quasiordnung-basierte Reduzierung der Automatengröße vor der Generierung der VHDL-Beschreibung trotz einer geschickten Abbildung der Automaten auf Hardware und der Optimierungen von Synthesetools für FPGAs zur Ressourceneinsparung führen können.

Nach der Reduzierung der Automatengröße können Automaten zwischen zwei Zuständen p und q mehrere Transitionen besitzen, die sich in der Zeichenklasse, den semantischen Prädikaten oder den Aktionen unterscheiden. Dies ist möglich, da in der Reduzierung lediglich identische Transitionslabel als äquivalent angesehen werden. Daher werden in einem an die Reduzierung anschließenden Schritt zwei Transitionslabel t_1 und t_2 mit $q \in \delta(p, t_1)$, $q \in \delta(p, t_2)$ und $t_1 \neq t_2$ zusammengefasst, wenn mindestens eine der folgenden Bedingungen gilt:

1. $pred(t_1) = pred(t_2) \wedge act(t_1) = act(t_2)$ - t_1 und t_2 unterscheiden sich lediglich bei der Zeichenklasse

2. $char(t_1) = char(t_2) \wedge pred(t_1) = pred(t_2)$ - t_1 und t_2 unterscheiden sich lediglich bei den Aktionsmengen

Die beiden Bedingungen werden für alle Zustände p mit deren Nachfolgern q nacheinander zum Zusammenfassen von Transitionen überprüft. Im ersten Fall werden die Zeichenklassen vereinigt, im zweiten Fall die Aktionsmengen.

Anschließend folgt der zweite Optimierungsschritt für eine optimierte Abbildung der bzgl. der Größe reduzierten *Action*-NFAs auf Hardwarebausteine von FPGAs.

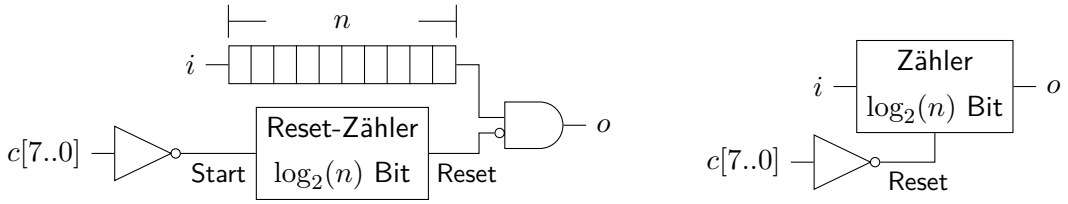
4.3.2 Optimierungen für FPGAs

Im zweiten Optimierungsschritt werden Ressourceneinsparungen durch eine geschickte Abbildung der optimierten Automaten auf Hardwarebausteine des FPGAs erreicht. Trotz der in Kapitel 2 vorgestellten Abbildung von Automaten auf Logikblöcke können durch weitere Optimierungen zusätzliche Ressourcen eingespart werden oder die Effizienz der Schaltungen verbessert werden [5, 39, 59, 72, 73]. In diesem Optimierungsschritt werden keine direkten Veränderungen der Automaten vorgenommen, sondern lediglich Zusatzinformationen für die Automaten berechnet. Diese Zusatzinformationen werden dann bei der Codegenerierung zur optimierten Abbildung der Automaten auf FPGAs genutzt. Der Fokus der zweiten Optimierungsstufe liegt auf bedingten Wiederholungen.

Für bedingte Wiederholungen der Form $\alpha\{n\}$, $\alpha\{n,\}$ und $\alpha\{n,m\}$ von einzelnen Zeichen oder Zeichenklassen führt die Nutzung von Lookup-Tabellen als Shiftregister sowie von Zählern zur Ressourceneinsparung [5, 59]. Durch die Abbildung solcher Teilausdrücke auf Shiftregister und bzw. oder Zähler auf dem FPGA kann der Ressourcenbedarf deutlich verringert werden. Beispielsweise werden für den regulären Ausdruck $[a\text{-}z]\{10000\}$ nur 625 Shiftregister a 16 Bit benötigt, anstelle von 10000 Registern bei der Realisierung als einzelne Register. Im zweiten Optimierungsschritt wird daher in den NFAs nach linearen Pfaden (ggf. mit Schleife am Ende oder optionaler Fortsetzung) gesucht, bei denen alle Transitionslabel auf dem Pfad identisch sind und jeder Zustand des linearen Teils (bis auf den ersten und letzten Zustand) genau einen Vorgängerzustand und einen Nachfolgezustand besitzt. Für Ausdrücke der Form $\alpha\{n,\}$ besitzt der letzte Zustand eine Transition mit dem gleichen Transitionslabel zu sich selbst. Bei Ausdrücken der Form $\alpha\{n,m\}$ besitzen im optionalen Teil jeweils benachbarte Zustände (bis auf den jeweiligen Folgezustand des Pfades) vergleichbare Nachfolgezustände. Alle für einen NFA berechneten Pfade werden gesammelt und für die Codegenerierung in einer geeigneten Datenstruktur bereitgestellt.

Die Hardware-Realisierung eines Ausdrucks der Form $\alpha\{n\}$ basiert auf einem Shiftregister der Länge n, welches die einzelnen Register des linearen Pfades im Automaten zusammenfasst. Bei einem Mismatch des Eingabezeichens mit α muss das Shiftregister resettet werden. Allerdings unterstützen die in Lookup-Tabellen realisierten Shiftregister auf FPGAs in der Regel keine Reset-Funktion [70], sodass die Reset-Funktion des Shiftregisters über einen Zähler simuliert wird [5, 59]. Bei einem Mismatch wird der Zähler aktiv und sorgt in den folgenden n Taktzyklen für einen Reset. Die Realisierung ist in Abbildung

4.6(a) grafisch dargestellt. Mit der Verwendung von Shiftregistern werden auch überlap-



(a) Shiftregister für $\alpha\{n\}$ (b) Zähler für $\alpha\{n,\}$

Abbildung 4.6: Optimierte Realisierung für Ausdrücke der Form $\alpha\{n\}$ und $\alpha\{n,\}$ (nach [5, 59])

pende Matchings erkannt, wie sie beispielsweise für den regulären Ausdruck $\beta = \texttt{.*[a-z]}^n$ vorkommen können. Dies ist mit einem einzelnen Zähler alleine nicht möglich [19].

Für reguläre Ausdrücke der Form $\alpha\{n,\}$ reicht zur Realisierung ein einzelner Zähler (siehe Abbildung 4.6(b)), der den einmal erreichten, maximalen Wert n beibehält und dann ein aktives Ausgabesignal liefert. Bei einem Mismatch wird der Zähler resettet. Ein einziger Zähler reicht für Ausdrücke dieser Form aus, da bei einer Folge von Eingabezeichen, die alle in $\mathcal{L}(\alpha)$ enthalten sind, lediglich das erste, mit α matchende Zeichen nach dem letzten Reset als Startpunkt des Zählers beachtet werden muss [59].

Für reguläre Ausdrücke der Form $\alpha\{n,m\}$ wird neben einem Shiftregister und einem Zähler für die ersten n Vorkommen von α ein weiterer Zähler für die folgenden m Vorkommen von α eingesetzt (siehe Abbildung 4.7) [5, 59]. Der erste Zähler wird bei einem Match aktiviert, zählt bis n und behält seinen Wert anschließend bei. Der Zähler liefert bei einem Wert von n ein aktives Ausgabesignal. Wenn gleichzeitig das Shiftregister einen Match liefert, wird der zweite Zähler aktiviert und auf 0 gesetzt. Solange der zweite Zähler einen Wert kleiner als $m - n$ hat, liefert er ein aktives Ausgabesignal, sonst ein inaktives Signal. Bei einem Mismatch werden beide Zähler resettet.

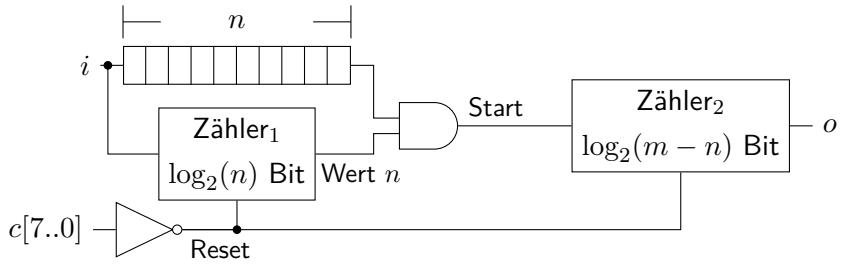
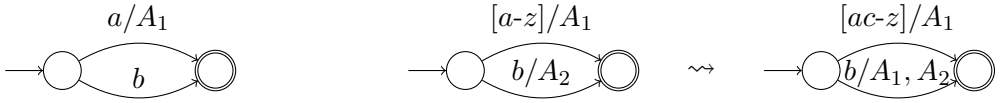


Abbildung 4.7: Optimierte Realisierung für Ausdrücke der Form $\alpha\{n,m\}$ [5, 59]

Die Bausteine für bedingte Wiederholungen innerhalb des Parsergenerators sind so implementiert, dass sie die Verwendung von Aktionen und semantischen Prädikaten unterstützen. Dazu besitzen alle drei Bausteine einen weiteren Eingang für die semantischen Prädikate und einen zweiten Ausgang für das Feuern von Aktionen. Zu beachten ist, dass

alle Transitionen des linearen Pfades in einem Automaten die selben Aktionen und Prädikate besitzen müssen, damit die Optimierungen eingesetzt werden können.

Mit der Verwendung von Aktionen und semantischen Prädikaten ist der Effekt einiger Optimierungen für die Abbildung von endlichen Automaten auf FPGAs bzgl. der Ressourceneinsparung kritisch zu beurteilen. Dies wird bei der in [2, 3] vorgestellten Reduktion der Alphabetgröße für NFAs deutlich, welche beim lokalen Matchen der Eingabezeichen in der Transitionslogik zur Einsparung von Ressourcen führen kann. Dabei werden die 8-Bit Eingabezeichen des erweiterten ASCII-Zeichensatzes in Codewörter mit weniger Bits übersetzt, welche dann in der Transitionslogik verwendet werden. Zwei Eingabezeichen $\sigma_1, \sigma_2 \in \Sigma$ verhalten sich identisch, wenn für alle Zustände $q \in Q$ eines NFAs $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ gilt, dass $\delta(q, \sigma_1) = \delta(q, \sigma_2)$ ist. Sie erhalten dann ein gemeinsames Codewort. Mit Aktionen und semantischen Prädikaten ergeben sich jedoch Sonderfälle, welche in Abbildung 4.8 grafisch dargestellt sind. Für den Fall aus Abbildung 4.8(a) werden für die Eingabezeichen a und b zwei verschiedene Codewörter benötigt, da sonst die Aktion A_1 auch für b ausgeführt wird. Folglich müssen Aktionen und semantische Prädikate bei der Abbildung von Eingabezeichen auf kürzere Codewörter berücksichtigt werden. Bei dem in Abbildung 4.8(b) dargestellten Fall müsste die Zeichenklasse [a-z] vor Abbil-



(a) Einzelne Aktion (b) Verschiedene Aktionen

Abbildung 4.8: Auswirkungen von Aktionen auf die Reduktion der Alphabetgröße nach [2, 3]

dung der Zeichen auf kürzere Codewörter zur Reduktion der Alphabetgröße wegen der Aktionen aufgespalten werden - schließlich kann einem Eingabezeichen b nur ein Codewort zugeordnet werden. Mit dem Aufteilen der Zeichenklasse werden jedoch zusätzliche Ressourcen für das Matchen der neuen Zeichenklasse [ac-z] benötigt, sodass die Reduktion der Alphabetgröße nicht zwingend zu einer Ressourceneinsparung führen muss. Aus diesem Grund ist dieses Verfahren nicht im implementierten Parsergenerator enthalten, weil der Nutzen bzgl. der Ressourceneinsparung zweifelhaft und schwer vorhersehbar ist.

An die Optimierungsphase schließt sich mit der Codegenerierung die letzte Phase des Parsergenerators an, die im nächsten Abschnitt detaillierter vorgestellt wird.

4.4 Codegenerierung

In der Phase zur Codegenerierung wird eine VHDL-Beschreibung für die gewonnenen *Action*-NFAs unter Berücksichtigung der in der Optimierungsphase berechneten Zusatzinformationen für das Pattern Matching auf FPGAs erzeugt. Die Abbildung der Automa-

ten auf FPGAs orientiert sich an [72, 73] und basiert auf einheitlichen Zustandsmodulen mit lokalen Matching der Zeichen in der Transitionslogik. In [73] besitzen die Automaten ein einheitliche Struktur, bei der jeder lesenden Transition genau eine Epsilon-Transition vorangeht. Daher werden in einem vorbereitenden Schritt Epsilon-Transitionen in die Automaten eingefügt. Anschließend erfolgt die eigentliche Codegenerierung mit Erzeugung der VHDL-Beschreibung.

4.4.1 Einfügen von Epsilon-Transitionen

Das Einfügen der Epsilon-Transitionen in die Automaten soll die Anzahl der benötigten Zustände nur minimal erhöhen, um den Optimierungen aus der vorherigen Phase nicht entgegenzuwirken. Daher werden die Epsilon-Transitionen für jeden Zustand q eines *Action*-NFAs $\mathcal{A}$ anhand der eingehenden Transitionen berechnet. Jedes Transitionslabel t, welches in einer in q eingehenden Transition vorkommt, führt zu einem neuen Zustand q_t, der über $\delta(q_t, t) = \{q\}$ mit q verknüpft wird. Die bisherigen Vorgänger von q werden mit dem jeweiligen Zustand q_t über Epsilon-Transitionen verbunden. Somit „sammelt“ der Zustand q_t die eingehenden Transitionen für q mit dem Transitionslabel t (siehe Abbildung 4.9). Zusätzlich wird ein neuer Zustand q_f erzeugt, der alleiniger akzeptierender Zustand wird. Alle akzeptierenden Zustände von $\mathcal{A}$ werden über Epsilon-Transitionen mit q_f verbunden und verlieren ihre akzeptierende Eigenschaft.

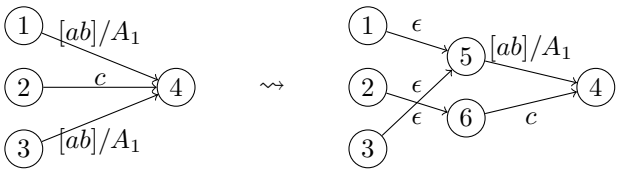


Abbildung 4.9: Einfügen von Epsilon-Transitionen als Vorbereitung der Codegenerierung

Durch die eingefügten Epsilon-Transitionen besitzen die *Action*-NFAs der Top-Level-Ausdrücke eine einheitliche Struktur, vergleichbar mit den Automaten in der Arbeit [73], sodass die Abbildung der Automaten auf einheitliche Zustandsmodule (und Bausteine für Optimierungen) möglich ist. Die Generierung der VHDL-Beschreibung für den Pattern Matcher wird im nächsten Abschnitt beschrieben.

4.4.2 VHDL-Umsetzung der Pattern Matcher

Die generierte VHDL-Beschreibung für das Pattern Matching auf FPGAs beschreibt eine getaktete Schaltung, bei der lediglich der Reset (z. B. nach dem Einschalten des FPGAs) asynchron erfolgt. Der Eingabestrom für das Pattern Matching wird in Form von 8-Bit Signalen (`data`-Leitung) erwartet. Zusätzlich signalisiert die spezielle 1-Bit `dataValid`-Leitung, dass ein gültiges Eingabezeichen vorliegt. Dadurch können u. a. Taktzyklen überbrückt werden, in denen keine Eingabezeichen vorliegen, weil z. B. keine weiteren Netz-

werkpakete mit Daten zu verarbeiten sind. Die Ausgabeports der generierten Schaltung werden über den VHDL-Coderahmen in der Spezifikation durch den Anwender festgelegt.

Die endlichen Automaten des Pattern Matchers werden in der generierten VHDL-Beschreibung standardmäßig, d. h. ohne Berücksichtigung der Optimierungen für FPGAs, durch Instanzen sogenannter *Zustandsmodule* dargestellt. Die Kommunikation der Zustandsmodule untereinander wird durch die Epsilon-Transitionen bestimmt. Die Ausführung der Aktionen sowie die Berechnung der semantischen Prädikate erfolgt innerhalb eines sequentiellen VHDL-Prozesses, der auf einem FPGA parallel zur Berechnung der Transitionen des Automaten ausgeführt wird. Ein Zustandsmodul repräsentiert ein Paar p, q von Zuständen mit einer Transition t eines *Action*-NFAs. Somit besteht ein Zustandsmodul aus kombinatorischer Logik zum Ermitteln aktiver Eingangsleitungen der Vorgängerzustände und dem Matchen des Eingabezeichens sowie einem Register für die Kodierung des Zustandes q (siehe Abbildung 4.10). Die Realisierung der Zustandsmodule orientiert sich an dem Basisbaustein aus Kapitel 2 und passt zur Architektur der Logikeinheiten von FPGAs [72, 73]. Die VHDL-Entitäten der Zustandsmodule unterscheiden sich lediglich

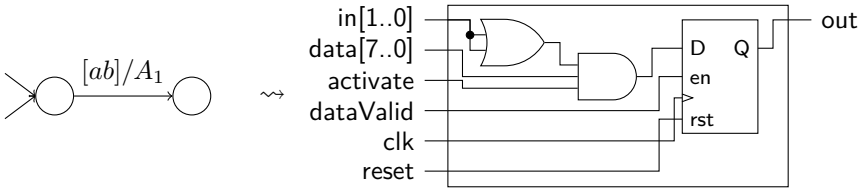


Abbildung 4.10: Zustandsmodul für endliche Automaten auf FPGAs

anhand der zu matchenden Zeichen. Für häufig anzutreffende Fälle wird eine Bibliothek mit VHDL-Entitäten genutzt, für alle anderen Zustandsmodule werden die Entitäten separat generiert. Vordefinierte Typen existieren für ein einzelne Zeichen σ, für invertierte Zeichen [ˆσ], für Zeichenbereiche [σ_1-σ_2], für invertierte Zeichenbereiche [ˆσ_1-σ_2] und für den Wildcard-Operator. Alle Zustandsmodule sind über einen generischen Parameter mit der Anzahl der Vorgängerzustände parametrisierbar. Die vordefinierten Zustandsmodule werden ggf. auch mit den matchenden Zeichen bzw. Bereichsgrenzen parametrisiert.

Durch die Abbildung von *Action*-NFAs auf eine Menge von Zustandsmodulen werden Zustände, in die mehrere Transitionen mit unterschiedlichen Transitionslabeln eingehen, in mehrere Zustandsmodule der VHDL-Beschreibung übersetzt. Nach der Optimierungsphase unterscheiden sich die eingehenden Transitionslabel anhand der Zeichenklasse, den Aktionen, oder den semantischen Prädikaten. Durch die Einbettung von semantischen Prädikaten als Eingangsport von Zustandsmodulen und die Kopplung der Aktionsausführung an den Ausgangsport eines Zustandsmoduls wird die Aufteilung eines Zustandes in mehrere Zustandsmodule auch notwendig.

Der Aktionscode wird in der generierten VHDL-Beschreibung in einen sequentiellen Prozess eingebettet, der nach der Synthese auf dem FPGA parallel zu der Verarbeitung des

Datenstroms durch die Automaten ausgeführt wird. Das Feuern der Aktionen von Transitionen erfolgt über das aktivierte Ausgangssignal der korrespondierenden Zustandsmodule. Dadurch wird der Aktionscode auf dem FPGA immer im Taktzyklus ausgeführt, der auf das Schalten der feuernden Transition folgt. Damit das zur Transition korrespondierende Eingabesymbol innerhalb des Aktionscodes genutzt werden kann, wird dieses im 8-Bit Vektor `dataInBuffer` gepuffert. Mit der Verlagerung der Aktionsausführung auf den folgenden Taktzyklus werden zudem die Signallaufzeiten für das Eingabezeichen entkoppelt, sodass die Timingvorgaben beim Synthesevorgang für die generierte VHDL-Beschreibung leichter einzuhalten sind.

Mit der Verwendung nichtdeterministischer Automaten können mehrere Transitionen gleichzeitig schalten und dabei eine Aktion mehrfach feuern. Damit die Häufigkeit der Aktionsausführung für den Anwender vorhersehbar bleibt, wird jede Aktion pro Taktzyklus maximal einmal ausgeführt. Aus Anwendersicht ist zudem zu beachten, dass die Ausführung aller gefeuerten Aktionen auf dem FPGA parallel erfolgt. Dadurch sind konkurrierende Zuweisungen für VHDL-Signale oder -Variablen im Aktionscode zu vermeiden.

Die Auswertung der semantischen Prädikate erfolgt ebenfalls im sequenziellen VHDL-Prozess, der auch die Aktionsausführung enthält. Im generierten Code werden die booleschen Ausdrücke der semantischen Prädikate jeden Taktzyklus ausgewertet und die Auswertungsergebnisse in lokalen Variablen des VHDL-Prozesses abgelegt. Anschließend erfolgt die konjunktive Verknüpfung der ggf. negierten Variablen für die Zuweisungen zu den `activate`-Eingangsports der Zustandsmodule. Die Ergebnisse der Auswertungen fließen somit im folgenden Taktzyklus beim Schalten der Transitionen des endlichen Automaten ein. Bei einem inaktiven `activate`-Eingang wird das Register eines Zustandsmoduls ebenfalls inaktiv - und die Transition damit zur Parsezeit deaktiviert. Nur bei einem aktiven Signal für den `activate`-Eingang ist die Transition des Zustandsmoduls aktiviert und kann schalten.

Die während der Optimierungsphase berechneten Zusatzinformationen für bedingte Wiederholungen der Form $\alpha\{n\}$, $\alpha\{n,\}$ und $\alpha\{n,m\}$ von einzelnen Zeichen oder Zeichenklassen fließen durch weitere, anstelle von Zustandsmodulen verwendete VHDL-Entitäten in die generierte VHDL-Beschreibung ein. Für die linearen Pfade in Automaten (ggf. mit Schleife am Ende oder optionaler Fortsetzung) werden VHDL-Entitäten unter Verwendung von Shiftregistern und Zählern nach dem Prinzip aus Abschnitt 4.3.2 erzeugt. Diese Entitäten werden bei Bedarf für jede Zeichenklasse separat generiert und mit mit den Grenzen n und m parametrisiert. In der VHDL-Beschreibung des Automaten wird anstelle vieler Instanzen ähnlicher bzw. identischer Zustandsmodule für einen solchen linearen Pfad eine einzige Instanz von einer der erzeugten VHDL-Entitäten verwendet. Mit einem Eingangsport für semantische Prädikate und einem zweiten Ausgabeport für die Aktionsausführung können die Entitäten der bedingten Wiederholungen wie Zustandsmodule im Gesamtkontext des *Action*-NFAs eingesetzt werden.

Bezüglich des Startzustands der *Action*-NFAs der Top-Level-Ausdrücke ist Folgendes zu beachten: Die für das Pattern Matching generierte VHDL-Beschreibung beinhaltet ein spezielles Signal für den Startzustand, welches initial beim Upload der Schaltung auf ein FPGA oder durch ein Reset einmalig aktiv wird. Mit dem ersten zu verarbeitenden, gültigen Eingabezeichen wird dieses Signal wieder deaktiviert. Dadurch werden von den Automaten nur solche Pattern erkannt, die direkt mit dem ersten Zeichen des Datenstroms beginnen. Durch Voranstellung des regulären Teilausdrucks `.*` in den Top-Level-Ausdrücken der Spezifikation können auch alle Vorkommen des Pattern im Datenstrom erkannt werden. Dadurch liegt die Entscheidung über die gewünschte Variante beim Anwender.

Die asynchrone Reset-Funktion der Automaten kann zur Initialisierung der Schaltung auf dem FPGA, z. B. nach dem Einschalten des Boards, genutzt werden. Bei einem Reset werden alle Zustände des Automaten in den Zustandsmodulen sowie den Entitäten der linearen Pfade inaktiv. Zudem wird das Signal für den Startzustand aktiviert. Für das erste Eingabezeichen nach einem Reset werden alle Transitionen, die semantische Prädikate nutzen, aktiviert.

Die vom implementierten Parsergenerator generierte VHDL-Beschreibung wird in einem Coderahmen (wie eine VHDL-Entität) eingebettet. Dieser Coderahmen wird in der Spezifikation durch den Anwender definiert und enthält Platzhalter, die bei der Codegenerierung mit dem erzeugten Code für das Pattern Matching ersetzt werden. Durch die Angabe des Coderahmens in der Spezifikation liegt die Definition der Ein- und Ausgabeports, die Instanziierung weiterer VHDL-Entitäten oder die Verwendung sequentieller Logik in der Hand des Anwenders. Der Pattern Matcher wird über fest definierte Eingabeports an den Coderahmen angebunden. Dies sind die Ports `clk` für den Takt, `reset` für die asynchrone Resetfunktion, `dataIn` für die Eingabezeichen und `dataIn_Valid` für das Vorliegen eines gültigen Eingabezeichens. Dazu kommen die in Kapitel 3 angegebenen Platzhalter, die bei der Codegenerierung verwendet werden und die ihrerseits Anforderungen an einen funktionsfähigen VHDL-Coderahmen stellen.

Die Vorstellung ausgewählter Aspekte der Implementierung des Parsergenerators ist damit abgeschlossen. Im nächsten Kapitel wird der implementierte Parsergenerator anhand einiger praktischer Beispiele evaluiert.

Kapitel 5

Evaluation

In diesem Kapitel werden die nichtdeterministischen endlichen Automaten des implementierten Parsergenerators für das Pattern Matching auf FPGAs bezüglich des Ressourcenbedarfs und des erreichbaren Durchsatzes anhand einiger praktischer Beispiel evaluiert. Dabei werden die Auswirkungen der implementierten Optimierungen auf den Ressourcenbedarf und den Durchsatz betrachtet. Zudem werden die *Action*-NFAs des Parsergenerators mit den deterministischen Automaten von Snowfall [61] bzgl. des Ressourcenbedarfs und des maximalen, erreichbaren Durchsatzes verglichen.

Zunächst werden das Setup für die Evaluierung des Pattern Matchings auf FPGAs sowie die verwendeten Beispiele detaillierter vorgestellt. Anschließend werden die Messungen für den Ressourcenbedarf und den Durchsatz dargestellt und ausgewertet.

5.1 Evaluierungssetup

Für die Evaluierung werden die generierten Pattern Matcher an eine Netzwerkschnittstelle des FPGAs angebunden, sodass der Eingabestrom für die Mustererkennung über UDP-Pakete an das FPGA gesendet wird. Die Ausgabe eines Pattern Matchers wird ebenfalls mit UDP-Paketen über das Netzwerk versendet. Mit der Anbindung der evaluierten Pattern Matcher an die I/O-Schnittstellen des FPGAs wird zugleich sichergestellt, dass die Pattern Matcher nicht durch die Optimierungen von Synthesetools ignoriert werden. Das Setup für die Evaluation ist in Abbildung 5.1 grafisch dargestellt.

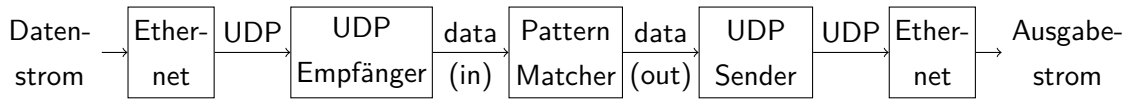


Abbildung 5.1: Evaluierungssetup für das Pattern Matching auf FPGAs

Für die Evaluation wird ein Virtex-5 XC5VLX110T-1FF336 FPGA von Xilinx verwendet. Das Board besitzt 17.280 Slices mit je vier Lookup-Tabellen und Registern pro Slice,

von denen ein Teil für die Realisierung von Shiftregistern oder verteilten RAM genutzt werden kann [71]. Somit stehen auf dem FPGA insgesamt 69.120 Register und LUTs für das Pattern Matching zur Verfügung. Die Lookup-Tabellen der Slices besitzen jeweils sechs Eingänge und können eine boolesche Funktion mit sechs Parametern oder zwei boolesche Funktionen mit jeweils fünf Eingängen realisieren, sofern deren Eingänge nahezu identisch sind. Zur Synthese von Hardwareschaltungen für das Pattern Matching auf dem FPGA wird das Tool *ISE* von Xilinx in der Version 14.7 eingesetzt.

Die Evaluation des Parsergenerators erfolgt anhand folgender Anwendungsbeispiele: Ein praktisches Beispiel ist ein Parser für XML-Datenströme, der einen solchen Datenstrom auf der Basis von XPath-Anfragen filtert. Die gesuchten XPath-Anfragen sind selbst Bestandteil des zu filternden Datenstroms (siehe Listing 5.1 für einen beispielhaften Datenstrom zur Filterung) und konfigurieren den Parser auf dem FPGA. Die Ausgabe des Parsers entspricht dem anhand der XPath-Anfragen selektierten Teil des XML-Datenstroms. Im angegebenen Datenstrom (Listung 5.1) werden beispielsweise beliebige `<b>`-Tags mit Kind `<c>` sowie `<i>`-Tags mit Kind `<j>` gesucht.

Listing 5.1: Beispiel: Datenstrom als Eingabe für den XML-Parser

```
<?query reset?>
<?query fn:root()/descendant::b/child::c #?>
<?query fn:root()/descendant::i/child::j #?>

<a>
  <b>
    <c>
      <d>Inhalt von d</d>
    </c>
    <e>Inhalt von e</e>
  </b>
  <f>
    <b>
      <g>Inhalt von g</g>
    </b>
  </f>
  <h>
    <i>
      <j>Inhalt von j</j>
    </i>
  </h>
</a>
```

Die weiteren evaluierten Beispiele sind verschiedene, von Emerging Threats[1] frei verfügbare Regelsätze für das Network Intrusion Detection System *Snort* [12, 55]. Mit einem im Rahmen der Evaluation entwickelten Konvertierungstool können die regulären Ausdrücke aus den Snort-Regelsätzen extrahiert und in das Eingabeformat des implementierten Parsergenerators bzw. das Eingabeformat von Snowfall transformiert werden. Die evaluierten Regelsätze sind in Tabelle 5.1 zusammen mit einigen wichtigen Informationen aufgeführt. Als Ausgabe erzeugt ein Pattern Matcher für einen Regelsatz mit m Regeln einen Bitvektor der Länge m, wobei Bit i angibt, ob die i-te Regel im vorherigen Taktzyklus erfolgreich gematcht wurde. Für den Versand in einem UDP-Paket wird der Bitvektor mit Hilfe einer Folge von `xor`-Operationen schrittweise zu einem 8-Bit Signal zusammengefasst. Dabei werden nach jeweils drei `xor`-Operationen Pufferregister eingesetzt, um die Signallaufzeiten für die `xor`-Operationen nicht zu einem Flaschenhals bei der Synthese werden zu lassen.

Die in den regulären Ausdrücken verwendeten Operatoren (wie z. B. Vereinigung oder Wiederholungen) beeinflussen den Ressourcenbedarf und den Durchsatz der generierten Pattern Matcher auf dem FPGA. Beispielsweise führen Vereinigungen in der Regel zu einem höheren Ressourcenbedarf als reine Konkatenationen, wenn die Länge der regulären Ausdrücke vergleichbar ist. Daher werden für eine detailliertere Auswertung der Messungen für den Ressourcenbedarf und den Durchsatz die folgenden Metriken für die endlichen Automaten hinzugezogen [72, 73]:

Anzahl der Zustände Gesamtanzahl der Zustände des Automaten

Maximaler Fan-In Maximale Anzahl an Transitionen, die in einen einzelnen Zustand des Automaten führen

Die beiden Metriken haben einen direkten Bezug zum Ressourcenbedarf und Durchsatz eines auf einem FPGA realisierten Pattern Matchers [72, 73]: Mit steigender Anzahl an Zuständen werden für die NFAs des implementierten Parsergenerators weitere Zustandsmodule in der VHDL-Beschreibung erzeugt. Eine größere Menge an Zustandsmodulen führt zu zusätzlich benötigten Registern und Lookup-Tabellen auf dem FPGA. Bei den DFAs von Snowfall erhöht sich mit der Anzahl der Zustände unabhängig von der konkreten Kodierung der Zustände in der Hardware auch die Anzahl der benötigten Register. Dabei kann die erforderliche Anzahl der Register logarithmisch mit der Anzahl der Zustände zusammenhängen [69]. Der maximale Fan-In eines Automaten wirkt sich auf die Anzahl der Eingänge von Oder-Gattern in den Zustandsmodulen aus. Eine größere Anzahl an Eingängen führt auf einem FPGA zur Verwendung weiterer Lookup-Tabellen, mit denen die booleschen Funktionen realisiert werden. Damit steigen die Signallaufzeiten zwischen zwei Registern an, die wiederum negative Auswirkungen auf den erreichbaren Durchsatz haben.

[1] Emerging Threats ist nun Teil von Proofpoint, `https://www.proofpoint.com/`

Tabelle 5.1: Verwendete Snort-Regelsätze mit Metriken für die Automaten

		DFA		NFA		NFA (reduziert)	
Regelsatz	Regeln	Zustände	Fan-In	Zustände	Fan-In	Zustände	Fan-In
ActiveX	80	nicht bekannt		5302	4	2473	99
Attack-Response	4	411	155	77	2	73	3
Chat	3	46	46	51	2	30	2
DNS	1	13	13	14	4	12	3
Dos	14	nicht bekannt		3586	180	3323	180
Exploit	99	nicht bekannt		10793	12	8780	102
FTP	41	nicht bekannt		2758	2	2580	41
Games	1	nicht bekannt		260	2	259	1
IMAP	15	nicht bekannt		1914	2	1846	15
Misc	2	nicht bekannt		158	2	156	2
Mobile-Malware	23	1900	1897	566	11	388	30
Netbios	146	nicht bekannt		1514	2	175	6
P2P	2	55	55	36	2	34	2
Policy	54	nicht bekannt		1777	13	1010	105
POP3	7	nicht bekannt		737	2	714	7
Scada	11	632	632	676	4	266	7
Scan	19	303718	198066	627	3	451	33
Shellcode	7	434	434	127	30	111	30
SMTP	7	nicht bekannt		636	2	604	9
User-Agents	9	601	601	202	2	126	9
Voip	3	nicht bekannt		1015	3	1011	3
Worm	1	22	22	23	3	22	3

Bei den von Snowfall erzeuten DFAs werden mit steigendem Fan-In weitere VHDL-Signale verwendet, die zu zusätzlich beanspruchten Register führen.

Im nächsten Abschnitt werden die Ergebnisse der Evaluation des implementierten Parsergenerators dargestellt, mit Messungen für DFAs, die von Snowfall erzeugt werden, verglichen und im Kontext existierender Arbeiten betrachtet.

5.2 Auswertung

Im Folgenden werden die Evaluationsergebnisse der verschiedenen Beispiele vorgestellt und im Kontext existierender Arbeiten interpretiert und ausgewertet. Die Evaluation des Ressourcenbedarfs erfolgt anhand der benötigten Lookup-Tabellen und Register für die

Pattern Matcher auf dem Virtex-5 FPGA. In den Messungen ist neben den eigentlichen Pattern Matchern auch die Verarbeitung der UDP-Pakete enthalten, sodass als zusätzliche Referenzgröße der Ressourcenbedarf für die reine Netzwerkverarbeitung angegeben wird. In allen Beispielen werden die Parser mit 125 MHz getaktet und verarbeiten 8-Bit Eingabezeichen eines erweiterten ASCII-Zeichensatzes, sodass eine Netzwerkbandbreite von einem Gbps für einen zu parsenden Datenstrom bereitsteht.

Die Messung des erreichbaren Durchsatzes erfolgt über die Bestimmung der maximalen Taktfrequenz, mit der die Parser auf dem Virtex-5 FPGA synthetisiert werden können. Dabei wird der Teil der Schaltung zur Netzwerkverarbeitung weiterhin mit 125 MHz getaktet und nur die Taktrate der Pattern Matcher angepasst. Anhand der gemessenen maximalen Taktfrequenz kann der erreichbare Durchsatz für die Parser bestimmt werden, wenn der zu parsende Datenstrom entsprechend mit der passenden Geschwindigkeit bereitgestellt werden kann.

Zunächst werden die Messungen dargestellt und genauer betrachtet. Anschließend folgen die Interpretation der Messungen und die Einbettung der Ergebnisse in den Kontext existierender Arbeiten.

5.2.1 Messdaten

Als erstes wird das Beispiel des XML-Parsers betrachtet. Anschließend folgen die Messungen für die Snort-Regelsätze. In allen Fällen wird auf die Quasiordnung-basierte Reduzierung der Automaten verzichtet, da der Algorithmus zur Berechnung der Relationen $\subseteq_L$ und $\subseteq_R$ bei großen Automaten viel Laufzeit in Anspruch nimmt und keine optimale Lösung zum Zusammenfügen der Zustände berechnet werden kann. Zudem zeigen sich auf Automatenebene bei den kleineren NFAs der Snort-Regelsätze (bis zur Größe NFAs für den Regelsatzes Voip mit etwa 1000 Zuständen) sowie dem XML-Parser im Vergleich zur Äquivalenz-basierten Reduzierung keine signifikanten Unterschiede in der Größe der reduzierten Automaten. Dies zeigen auch die Ergebnisse aus [39], bei denen die Äquivalenz-basierte und die Quasiordnung-basierte Reduzierung miteinander vergleichen werden[2].

XML-Parser Der XML-Parser besitzt einen Konfigurationsparameter für die maximale Pfadlänge der XPath-Anfragen, die zur Filterung des Datenstroms verwendet werden können. Zugleich ist der Parameter eine Begrenzung für die Gesamtlänge aller XPath-Anfragen, die gleichzeitig für die Filterung eingesetzt werden können. Dieser Parameter wird für die Evaluierung auf einen kleinen Wert von fünf gesetzt, damit der Ressourcenbedarf für die endlichen Automaten im Vergleich zur in allen Fällen identischen Vor- und Nachbearbeitung des Datenstroms (wie z. B. der Serialisierung der Ausgabe) besser sichtbar wird. Der Ressourcenbedarf, die maximal erreichbare Taktfrequenz und die Metriken

[2] In der Theorie erreichen Quasiordnung-basierte Algorithmen gegenüber Äquivalenz-basierten Algorithmen bessere Ergebnisse bei der Reduktion der Automatengröße [39].

für den XML-Parser sind in Tabelle 5.2 für einen mit Snowfall erzeugten, minimalen DFA sowie für NFAs, die mit dem implementierten Parsergenerator erzeugt wurden, dargestellt. Bei den NFAs wird ein Automat ohne Optimierungen, ein Äquivalenz-reduzierter NFA und ein für FPGAs optimierter NFA verwendet.

Tabelle 5.2: Ressourcenbedarf, Taktung und Metriken des XML-Parsers auf einem Virtex-5 FPGA

Ressource/Metrik	DFA	NFA	NFA (reduziert)	NFA (optimiert)
Register	2832	2913	2660	2660
Lookup-Tabellen	3937	3842	3492	3497
Slices	1774	1728	1731	1540
max. Taktfrequenz	≈172 MHz	≈200 MHz	≈198 MHz	≈200 MHz
Zustände	169	522	169	169
Transitionen	333	2895	290	290
max. Fan-In	164	21	12	12

Anhand der Messungen ist erkennbar, dass mit nichtdeterministischen Automaten gegenüber dem minimalen DFA Ressourcen in Form von Registern und Lookup-Tabellen auf dem FPGA eingespart werden können. Zudem lassen sich die NFAs etwas schneller takten als der minimale DFA und weisen somit einen höheren Durchsatz auf. Auffällig ist, dass der NFA ohne Optimierungen mehr Register, aber weniger Lookup-Tabellen als der minimale DFA benötigt. Insbesondere geben die für den NFA zusätzlich benötigten Register die Differenz bei der Zustandsanzahl (bzgl. des DFAs) nicht proportional wieder. Ein Grund könnte in der VHDL-Beschreibung des DFAs liegen, bei der zur Berechnung der Transitionen VHDL-Signale verwendet werden, die typischerweise als Register synthetisiert werden. Durch die Reduzierung der Größe des NFAs können weitere Einsparungen bei den Registern und Lookup-Tabellen erreicht werden, sodass der Ressourcenbedarf in beiden Fällen unter denen des minimalen DFAs entfällt. Die Optimierungen für bedingte Wiederholungen auf FPGAs haben beim XML-Parser keine Auswirkungen auf den Ressourcenbedarf, da die regulären Ausdrücke des Parsers keine solchen Wiederholungen enthalten.

Snort-Regelsätze Für die Messung des Ressourcenbedarfs der Snort-Regelsätze werden die regulären Ausdrücke eines Regelsatzes zu einem einzigen Top-Level-Ausdruck zusammengefasst, der aus der Vereinigung der einzelnen regulären Ausdrücke der Regeln gebildet wird. Dieses Verfahren ist einerseits bedingt durch Snowfall, da das Tool lediglich einen einzigen Top-Level-Ausdruck unterstützt. Andererseits werden die Optimierungen des implementierten Parsergenerators auf jeden Top-Level-Ausdruck separat angewendet, sodass Ressourcen gemeinsamer Teilausdrücke bislang nur bei Verwendung eines einzigen Top-Level-Ausdrucks geteilt werden können.

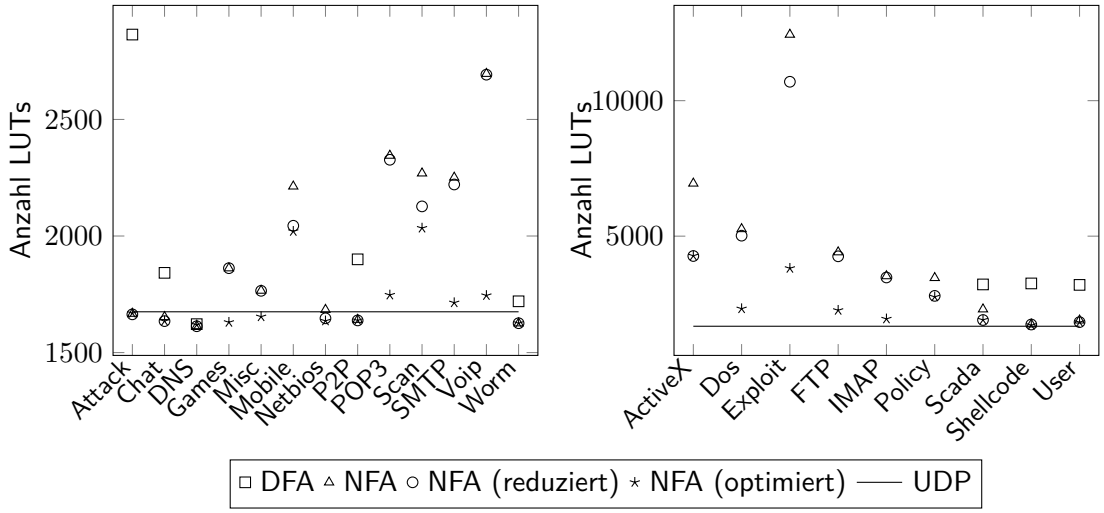


Abbildung 5.2: Ressourcenbedarf (LUTs) für Snort-Regelsätze auf einem Virtex-5 FPGA

Zur Messung des Ressourcenbedarfs werden für die Snort-Regelsätze neben dem minimalen DFA die gleichen Varianten von NFAs verwendet, die bereits beim XML-Parser betrachtet wurden: ein NFA ohne Optimierungen, ein Äquivalenz-reduzierter NFA und ein für FPGAs optimierter NFA. Allerdings liegen nicht für alle Regelsätze Messdaten für deterministische Automaten vor, falls diese sehr große Zustandsmengen besitzen (beispielsweise hat der DFA für den Scan-Regelsatz über 300.000 Zustände) und deren Berechnung durch Snowfall oder die Synthese auf dem FPGA viel Laufzeit in Anspruch nimmt bzw. nicht möglich ist. Hinzu kommt, dass der erwartete Ressourcenbedarf großer DFAs über den verfügbaren Ressourcen des verwendeten Virtex-5 FPGAs liegt. In Tabelle 5.1 ist für die Metriken von DFAs der Text „nicht bekannt“ angegeben, wenn die Automaten durch Snowfall gar nicht erzeugt werden konnten.

Der Snort-Regelsatz Exploit besitzt als einziges Beispiel der Evaluation eine Backreferenz in einem regulären Ausdruck, die sich auf ein einzelnes, zuvor gematchtes Zeichen bezieht. Da Backreferenzen von Snowfall und dem implementierten Parsergenerator nicht unterstützt werden, wird diese Backreferenz durch den referenzierten regulären Ausdruck ersetzt. Dadurch wird die Semantik des Snort-Regelsatzes zwar nicht korrekt wiedergegeben, allerdings lässt sich die Semantik der Backreferenz auch nicht ohne wesentlichen Aufwand durch Verwendung von semantischen Prädikaten und Aktionen nachstellen. Denn bei der Auswertung der semantischen Prädikate kann in der vorliegenden Implementierung das aktuelle Eingabezeichen nicht in gleichen Taktzyklus berücksichtigt werden, was aber für das Deaktivieren der Transition des Automaten erforderlich wäre. Hinzu kommt, dass Snowfall keine semantischen Prädikate unterstützt.

Durch Optimierung der FPGA-Abbildung bedingter Wiederholungen von Zeichenklassen im implementierten Parsergenerator sind Snort-Regelsätze bei der Auswertung der Messdaten besonders zu berücksichtigen, wenn sie solche Wiederholungen enthalten. Vor

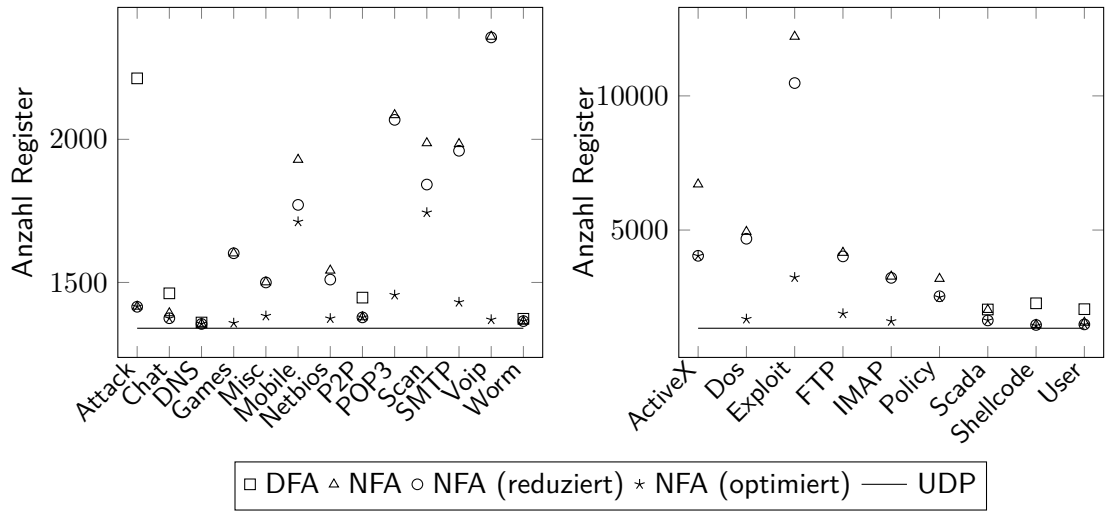


Abbildung 5.3: Ressourcenbedarf (Register) für Snort-Regelsätze auf einem Virtex-5 FPGA

allem in den Regelsätzen Dos, Exploit, FTP, Games, IMAP, Misc, POP3, Scan, SMTP und Voip sind bedingte Wiederholungen enthalten, bei denen die implementierten Optimierungen zur Einsparung von Ressourcen führen können. Der Anteil dieser Wiederholungen an der Gesamtgröße der regulären Ausdrücke variiert dabei von Regelsatz zu Regelsatz.

Die Messungen des Ressourcenbedarfs für die Snort-Regelsätze werden für Lookup-Tabellen und Register getrennt in Form von Grafiken angegeben. Der Bedarf an Lookup-Tabellen für die Pattern Matcher der Snort-Regelsätze auf dem Virtex-5 FPGA ist in Abbildung 5.2 dargestellt. Die Abbildung 5.3 enthält den Ressourcenbedarf an Registern für die Snort-Regelsätze. In den Abbildungen ist der Ressourcenbedarf für die Verarbeitung der UDP-Pakete als Referenzwert mit aufgetragen. Die exakten Messwerte sind zusätzlich in Anhang C in tabellarischer Form angegeben.

An den Messungen für den Ressourcenbedarf ist erkennbar, dass durch die Verwendung von NFAs gegenüber minimalen DFAs Ressourcen eingespart werden. Dabei weisen selbst NFAs ohne eine Form der Optimierung einen geringeren Bedarf an Lookup-Tabellen und Registern auf als der jeweilige minimale DFA. Mit Äquivalenz-reduzierten NFAs sowie der verbesserten Abbildung der Automaten durch Nutzung von Shiftregistern und Zählern können weitere Ressourcen eingespart werden, wobei die Auswirkungen dieser Optimierungen stark vom jeweiligen Regelsatz abhängig sind. Beispielsweise führt die Reduzierung der Automatengröße bei den Regelsätzen für ActiveX, Exploit und Mobile-Malware zur signifikanten Einsparung von Registern und Lookup-Tabellen, wogegen die Effekte der Reduzierung bei den Regelsätzen FTP oder SMTP zu vernachlässigen sind. Die Auswirkungen der optimierten Abbildung von Automaten auf FPGAs durch Verwendung von Shiftregister und Zähler auf den Ressourcenbedarf hängen vom Umfang des Vorkommens bedingter Wiederholungen in den Regelsätzen ab. Beispielsweise führen Zähler und Shiftregister bei den Snort-Regelsätzen für die Protokolle FTP und SMTP und sowie die Regelsätze für Ex-

Tabelle 5.3: Maximale Taktfrequenz von Parsern für Snort-Regelsätze auf dem Virtex-5 FPGA

Regelsatz	DFA	NFA	NFA (reduziert)	NFA (optimiert)
Attack-Response	≈189 MHz	≈217 MHz	≈217 MHz	≈206 MHz
Chat	≈212 MHz	≈217 MHz	≈203 MHz	≈209 MHz
DNS	≈207 MHz	≈202 MHz	≈219 MHz	≈209 MHz
P2P	≈211 MHz	≈207 MHz	≈215 MHz	≈214 MHz
Scada	≈184 MHz	≈212 MHz	≈212 MHz	≈215 MHz
Shellcode	≈156 MHz	≈207 MHz	≈218 MHz	≈212 MHz
User-Agents	≈192 MHz	≈211 MHz	≈206 MHz	≈218 MHz
Worm	≈214 MHz	≈216 MHz	≈216 MHz	≈211 MHz

ploit und Voip zu teilweise erheblichen Einsparungen an Ressourcen. Dagegen sind diese Optimierungen bei den Regelsätzen Mobile-Malware und Policy aus Sicht des Ressourcenbedarfs zu vernachlässigen. Folglich sind bei den regulären Ausdrücken der Regelsätze stets die jeweils verwendeten Operatoren und deren relativer Anteil bei der Bewertung der Effekte der implementierten Optimierungen zu berücksichtigen.

Neben dem Ressourcenbedarf wird für einige Snort-Regelsätze auch der maximale Durchsatz gemessen. Da der Vergleich der maximalen Taktfrequenz von deterministischen und nichtdeterministischen Automaten im Vordergrund der Evaluation steht, werden nur solche Regelsätze betrachtet, bei denen die DFAs in akzeptabler Laufzeit berechenbar und synthetisierbar sind. Die Messungen für die maximal erreichbare Taktfrequenz der Pattern Matcher auf dem verwendeten Virtex-5 FPGA sind in Tabelle 5.3 angegeben. An den Messdaten ist erkennbar, dass die Entscheidung für oder gegen die Verwendung von Nichtdeterminismus bei der Realisierung endlicher Automaten auf FPGAs auch den Durchsatz beeinflussen kann. Bei den kleinen Automaten sind die gemessenen Unterschiede in der Taktfrequenz von DFAs und NFAs nur marginal und können vernachlässigt werden. Hier dürfte die Ähnlichkeit von NFA und DFA einen großen Beitrag zu den geringen Unterschieden in der maximalen Taktfrequenz leisten. Bei größeren Automaten ist eine signifikante Abhängigkeit der maximalen Taktfrequenz von der Realisierung der Automaten zu erkennen. Beispielsweise lassen sich die Parser für die Snort-Regelsätze Scada und Shellcode bei Verwendung nichtdeterministischer Automaten schneller takten als Parser mit DFAs. Zudem wird die maximale Taktfrequenz durch das Aussehen der jeweiligen Automaten beeinflusst und hängt weniger von der Automatengröße ab. Dies wird u. a. an den Messungen für die deterministischen Automaten der Regelsätze Scada, Shellcode und User-Agents sichtbar. Während die DFAs der beiden Regelsätze Scada und User-Agents etwa 600 Zustände haben und sich die Automaten auf etwa 184 MHz bzw. 192 MHz takten lassen, besitzt der DFA für den Regelsatz Shellcode lediglich 434 Zustände und lässt sich nur auf

156 MHz takten. Dagegen liegt die maximale Taktfrequenz der Parser mit nichtdeterministischen Automaten stabil in allen gemessenen Fällen bei etwas über 200 MHz. Die leichten Schwankungen bei den Taktfrequenzen der NFAs könnten mit dem maximalen Fan-In einzelner Zustände sowie der Komplexität zum Matchen der während der Optimierungsphase zusammengefügten Zeichenklassen zusammenhängen.

Bei der Gegenüberstellung der Werte der Metriken aus Tabelle 5.1 mit dem Ressourcenbedarf ist zu erkennen, dass die Metriken ein Indikator für den Ressourcenbedarf darstellen. So führt eine größere Anzahl an Zuständen in einem Automaten auch zu einem gesteigerten Bedarf an Registern und Lookup-Tabellen auf dem FPGA. Ebenfalls lässt sich vereinzelt ein gesteigerter Bedarf an Lookup-Tabellen registrieren, wenn einzelne Zustände einen größeren Fan-In besitzen. Dabei kann ein selten auftretender, hoher Fan-In durch Synthesetools häufig kompensiert werden [73]. Bei den erzeugten NFAs kann ein hoher Fan-In außerdem zu weiteren Zustandsmodulen und damit zu zusätzlich benötigten Registern führen, wenn die eingehenden Transitionen viele verschiedene Transitionslabel besitzen. Dieser Effekt ist bedingt durch das Format der generierten VHDL-Beschreibung für einen Pattern Matcher durch den implementierten Parsergenerator. Weitere Details dazu finden sich in Kapitel 4.

Im folgenden Abschnitt werden die Messdaten genauer interpretiert und im Kontext existierender Arbeiten betrachtet.

5.2.2 Interpretation der Messungen

Beim Vergleich des Ressourcenbedarfs von deterministischen endlichen Automaten mit nichtdeterministischen Automaten ist die exakte Realisierung der DFAs auf FPGAs genauer zu betrachten, da diese den Bedarf an Registern und Lookup-Tabellen beeinflusst: Bei der Synthese der Beispiele durch Xilinx ISE werden die deterministischen Automaten praktisch immer wie nichtdeterministische Automaten auf dem FPGA realisiert, indem die Kodierung der Zustände mittels One-Hot Encoding erfolgt. Dadurch führt die in der Regel gegenüber einem NFA größere Zustandsmenge eines DFAs direkt zu einer größeren Anzahl an benötigten Registern. Alternativ kann jeder der n Zustände eines DFAs mit einem Integer mit $\log(n)$ Bits kodiert werden. Da zu jedem Zeitpunkt des Pattern Matchings in einem DFA genau ein Zustand aktiv ist, kann dieser aktive Zustand in $\log(n)$ Registern gespeichert werden. So kann die im Vergleich zu einem NFA ggf. exponentiell große Zustandsmenge des DFAs mit der gleichen Anzahl an Registern realisiert werden wie die Zustandsmenge des NFAs. Allerdings wird die Transitionslogik durch die Kodierung des aktiven Zustandes als Integer komplexer, da zur Bestimmung des Folgezustandes stets alle $\log(n)$ Bits berücksichtigt werden müssen. Für reguläre Ausdrücke, die den Worst-Case für die Größe eines DFAs darstellen, lässt sich auch mit der Kodierung des aktiven Zustandes als Integer der exponentielle Bedarf an Lookup-Tabellen nicht vermeiden [69]. In Bezug

auf die Messungen der Evaluation bedeutet dies, dass selbst eine kompakte Repräsentation eines DFAs auf dem FPGA die Ressourceneinsparungen der nichtdeterministischen Automaten bei den Lookup-Tabellen nicht ausgleichen kann. Bei Registern dagegen kann der Ressourcenbedarf der DFAs auf das Niveau der NFAs (wegen der Minimierung gegenüber der Reduzierung der NFAs ggf. auch darunter) gesenkt werden.

Neben dem Unterschied von Snowfall und dem implementierten Parsergenerator bzgl. der Automatenvarianten sowie dem damit verbundenen Ressourcenbedarf und maximalen Durchsatz zeigt die Evaluation nebenbei Unterschiede bei der Laufzeit des Tools ISE für die Synthese der Pattern Matcher auf. Im Vergleich zur Laufzeit für die Synthese von DFAs wird für die Synthese der VHDL-Beschreibungen der NFAs deutlich weniger Laufzeit benötigt. So ist beispielsweise der Syntheseprozess mit ISE für einen NFA mit mehreren tausend Zuständen wesentlich schneller abgeschlossen als die Synthese eines DFAs mit einigen hundert Zuständen. Zudem benötigt das Tool ISE zur Synthese der von Snowfall generierten DFAs viele Hardwareressourcen in Form von Arbeitsspeicher. Ein möglicher Grund für die Unterschiede bei den Laufzeiten der Synthese könnte im Format der generierten VHDL-Beschreibungen liegen. Während der implementierte Parsergenerator eine modulare VHDL-Beschreibung für die Automaten auf Basis der Zustandsmodule erzeugt, die das Layout auf dem FPGA praktisch vordefiniert, generiert Snowfall eine Beschreibung des Automaten mit zwei sequentiellen VHDL-Prozessen. Dabei erfolgt die Repräsentation der Zustände als ein Aufzählungstyp von VHDL. Die beiden VHDL-Prozesse bestehen aus vielen einzelnen Anweisungen für die Auswertung der Transitionslogik und die Berechnung des Folgezustandes, sodass sie für die längere Synthesezeit verantwortlich sein könnten.

Mit den Messungen zum Ressourcenbedarf können die Ergebnisse einer existierenden Arbeit bestätigt werden: Die Anwendung der Äquivalenz-basierten Reduzierung auf die NFAs führt häufig zu einem verminderten Ressourcenbedarf, auch wenn die Automaten durch eine geeignete Struktur (wie die zur Architektur des FPGAs korrespondierenden Zustandsmodule) auf das FPGA abgebildet werden. Auch die Optimierungen des Synthesetools ISE von Xilinx lassen diese Einsparung nicht vollständig verschwinden. Folglich können die Ergebnisse aus [39] bestätigt werden. Als Konsequenz daraus sollten Optimierungen auf der Ebene der nichtdeterministischen Automaten unabhängig von der Effizienz der Abbildung von Automaten auf die Hardware beim Pattern Matching auf FPGAs betrachtet werden, wenn Ressourcen eingespart werden sollen.

Bezüglich der Genauigkeit der Messungen ist anzumerken, dass der Ressourcenbedarf in allen Beispielen bei wiederholten Durchführungen des Syntheseprozesses mit dem Tool ISE Schwankungen unterliegen kann. Diese Schwankungen lassen sich ohne weitere Constraints für die Pattern Matcher oder ein nahezu vollständig ausgelastetes FPGA nicht vermeiden. Die Schwankungen zeigen sich auch in den Messdaten der Beispiele, da der Ressourcenbedarf einiger sehr kleiner Snort-Regelsätze (insbesondere bei Lookup-Tabellen) geringer ausfällt als der Ressourcenbedarf der UDP-Verarbeitung ohne Parser. Zusätzlich ist zu

berücksichtigen, dass der Ressourcenbedarf von den bei der Synthese mit ISE eingeschalteten Optimierungen abhängt. Bei den zur Evaluierung verwendeten Beispielen werden in allen Fällen identische Optimierungen genutzt, um die Vergleichbarkeit der Messungen innerhalb der einzelnen Regelsätze und zwischen den Regelsätzen zu gewährleisten. Die Aktivierung anderer Optimierungen kann jedoch zu anderen Messergebnissen für den Ressourcenbedarf und den Durchsatz führen. Beispielsweise haben vereinzelte Messungen einer Version des Synthesetools für die Kommandozeile mit anderen Optimierungen auch andere Messwerte für den Ressourcenbedarf ergeben.

Damit ist die Evaluierung des implementierten Parsergenerators abgeschlossen. Im nächsten Kapitel erfolgt die Zusammenfassung der Ergebnisse der Arbeit und es wird ein Ausblick für eine mögliche Weiterentwicklung des implementierten Parsergenerators zum Pattern Matching auf FPGAs gegeben.

Kapitel 6

Zusammenfassung

Mit dem implementierten Parsergenerator für das Pattern Matching wird das Parsen von Datenströmen auf FPGAs innerhalb vieler Anwendungen vereinfacht, wenn die gesuchten Muster über reguläre Ausdrücke beschrieben werden können. Durch die Verwendung von Aktionscode in Form von VHDL-Anweisungen ist neben der reinen Mustererkennung auch die Filterung, Transformation oder Komprimierung eines zu parsenden Datenstroms möglich. Mit der durch Aktionen erzeugten Ausgabe eines Parsers können weitergehende Verfahren zur Verarbeitung der geparsten Datenströme direkt in Hardware realisiert und an den Parser angebunden werden. Unter Nutzung von semantischen Prädikaten können zudem einige Datenströme mit endlichen Automaten geparst werden, bei denen kontextsensitive Informationen aus dem Datenstrom zur Parsezeit zu berücksichtigen sind.

Die Verwendung einer erweiterten Form von nichtdeterministischen, endlichen Automaten für Pattern Matcher auf Basis regulärer Ausdrücke durch den implementierten Parsergenerator führt im Vergleich zu den von Snowfall verwendeten deterministischen Automaten zur Einsparung von Ressourcen auf FPGAs. Zudem entfällt (mit Ausnahme der Realisierung für den Negationsoperator) die Determinisierung der Automaten, sodass die Zustandsexplosion in der Regel vermieden werden kann. Dadurch können größere reguläre Ausdrücke als Teil eines Pattern Matchers auf FPGAs realisiert werden, für die eine Realisierung als DFA nicht oder nur mit erheblichen Mehraufwand bzgl. der Laufzeit möglich ist, wie die Evaluation zeigt. Außerdem hält sich die Laufzeit für die Synthese der generierten VHDL-Beschreibungen des implementierten Parsergenerators im Vergleich zu den DFAs von Snowfall in Grenzen.

Durch die implementierten Optimierungen innerhalb des Parsergenerators können zusätzliche Ressourcen auf dem FPGA eingespart werden, wobei die Auswirkungen bei Nutzung der Optimierungen auf den Ressourcenbedarf stark von den verwendeten regulären Ausdrücken und Operatoren abhängig ist. Wie die Evaluation zeigt, können mit den Optimierungen auf Automatenebene und für die Abbildung auf FPGAs bei geeigneten regulären Ausdrücken signifikante Einsparungen bei den Ressourcen erreicht werden. Bei

der Implementierung von Optimierungen ist stets auf die Auswirkungen der Aktionen und semantischen Prädikate von *Action*-NFAs zu achten. Dadurch können Ressourceneinsparungen wie im Falle der betrachteten Alphabetreduktion [2, 3] schwer vorhersagbar werden. Unter Umständen führt die Synthese von Automaten nach Anwendung von Optimierungen durch die Einbettung von Aktionen und semantischen Prädikaten sogar zu einem höherem Ressourcenbedarf als die Synthese von Automaten, die ohne diese Optimierungen auskommen.

Neben dem verringerten Ressourcenbedarf der nichtdeterministischen Automaten des implementierten Parsergenerators gegenüber den DFAs von Snowfall zeigt die Evaluierung auch Unterschiede in der erreichbaren Taktfrequenz auf. Pattern Matcher, die auf NFAs basieren, lassen sich häufig schneller takten als Pattern Matcher auf Basis von DFAs, sodass mit den ersteren einen höheren Durchsatz beim Parsen von Datenströmen erreicht werden kann. Die Unterschiede in der maximalen Taktfrequenz werden dabei erst signifikant, wenn sich DFA und NFA nicht mehr sehr ähnlich sind.

Der implementierte Parsergenerator kann in Zukunft erweitert werden, um einerseits noch bessere Ergebnisse bzgl. des Ressourcenbedarfs oder des Durchsatzes von nichtdeterministischen endlichen Automaten auf FPGAs zu erzielen und andererseits die Mächtigkeit und damit die Einsatzmöglichkeiten des Tools zu vergrößern. Einige Möglichkeiten zur Erweiterung des implementierten Tools werden im Folgenden kurz vorgestellt.

Die Optimierung der Abbildung von *Action*-NFAs auf FPGAs ist mit der im Rahmen dieser Arbeit implementierten Version des Parsergenerators nicht ausgeschöpft, sodass der Parsergenerator in Zukunft um weitere, in der Literatur existierende Optimierungen zur Realisierung endlicher Automaten auf FPGAs erweitert werden kann. Beispielsweise kann die parallele Nutzung mehrerer getrennter Automaten verbessert werden, indem Lösungen für das Staging und Pipelining [72, 73] zur Begrenzung der Signallaufzeiten und Steigerung des Durchsatzes sowie die Berücksichtigung gemeinsamer regulärer Teilausdrücke (Präfixe, Infixe oder Suffixe) [40] integriert werden. Zur allgemeinen Steigerung des Durchsatzes der Pattern Matcher können auf Kosten eines erhöhten Ressourcenbedarfs auch mehrere Eingabezeichen in einem Taktzyklus gematcht werden [2, 3, 73]. Dazu müssten die nichtdeterministischen Automaten auf FPGAs etwas langsamer getaktet werden als es die maximale Taktfrequenz bei einem Zeichen pro Takt erlaubt, um insgesamt den Durchsatz zu steigern. Bei der Implementierung von Optimierungen für FPGAs sind immer die Einflüsse von Aktionscode und semantischen Prädikaten sowie die Effekte auf dieselben zu berücksichtigen, welche die Optimierungsziele verändern oder zunichte machen können.

Darüber hinaus können die Möglichkeiten des Pattern Matchings mit der Realisierung von FPGA-basierten Parsern für kontextfreie Sprachen erweitert werden. Dazu existieren in der Literatur mehrere Lösungen für die Realisierung von Parsern auf Basis des CYK-Algorithmus [10, 11] oder des Verfahrens von Earley [53], die auf FPGAs ausgerichtet sind und in den implementierten Parsergenerator integriert werden könnten. Beim Par-

sen kontextfreier Sprachen könnte Aktionscode ausgeführt werden, wenn die Realisierung auf Attibutgrammatiken [47] basiert - eine solche Implementierung wird in [15] vorgestellt und ermöglicht die Nutzung von synthetisierten Attributen mit der Realisierung über einen Stack. Als deterministische Parser von kontextfreien Sprachen werden in [9] Realisierungen für $LL(1)$- und $LR(1)$-Parser in Hardware vorgestellt, welche auf FPGAs übertragen werden könnten. Für $LL(1)$-Parser mit Nichtterminalen in Form einzelner Zeichen ist innerhalb des implementierten Parsergenerators eine solche Umsetzung samt Codegenerierung in einem experimentellen Status enthalten. Die Verknüpfung der endlichen Automaten mit kontextfreien Parsern unter Verwendung von Aktionscode ermöglicht Parsing (mit Bildung von Token und (impliziten) Aufbau einer grammatikalischen Struktur) in Hardware.

Mit dem Austausch der Codegenerierung und der zweiten Optimierungsstufe des implementierten Parsergenerators könnte die Implementierung zur Verarbeitung von endlichen Automaten außerdem für andere Zielsprachen oder -plattformen weiterverwendet werden. Dadurch könnten nichtdeterministische endliche Automaten für das Pattern Matching mit relativ geringen Aufwand auch in anderen Programmiersprachen oder zur Ausführung auf anderen Hardwareplattformen wie GPUs bereitgestellt werden.

Anhang A

ANTLR v4 Grammatik zum Parsen von Spezifikationen

Im Folgenden wird die ANTLR v4 Grammatik zum Parsen einer Spezifikation für den implementierten Parsergenerator vorgestellt. Dabei wird die Grammatik in einer erweiterten Backus-Naur-Form (kurz EBNF) angegeben, wobei die in der Implementierung zusätzlich vorhandenen Aktionen (in Form von Java-Code) für die Validierung einer Spezifikation aus Gründen der besseren Lesbarkeit ausgelassen werden. Aus der Grammatik wird mit dem Maven-Plugin von ANTLR im Build-Prozess ein Parser zum Verarbeiten der Spezifikationen für den implementierten Parsergenerators generiert.

Listing A.1: ANTLR-Gramamtik zum Parsen von Spezifikationen

```
specification : CODEFRAME? specificationBlock CODEFRAME? EOF;
specificationBlock : '%%{' (MACHINE IDENTIFIER ';')?
    actionCode* specificationRule* '%%}';
actionCode : ACTION IDENTIFIER '{' CODELINE* '}';
specificationRule : IDENTIFIER '=' regexp ';'
    | IDENTIFIER ':=' regexp ';';
regexp : concatenation ('|' concatenation)*
    | concatenation '&' concatenation
    | concatenation ('-' | '--') concatenation;
concatenation : quantifiedAtom*;
quantifiedAtom : '!'? atom ([*+?] | '{' INT (',' INT?)? '}')?
    (ACTIONREFERENCE | SEMANTICPREDICATE)*;
atom : SINGLECHARACTER | '.' | CHARACTERSTRING
    | '[' '^'? (CCLASSCHARACTER | CCLASSCHARACTER
      '-' CCLASSCHARACTER)+ ']'
    | '(' regexp ')' | RULEREFERENCE;
```

Die ANTLR-Grammatik mit dem Startsymbol `specification` ist in Listing A.1 angegeben. Dabei bestehen alle Bezeichner von Token des Lexers lediglich aus Großbuchstaben und die Nichtterminale der Grammatik beginnen stets mit einem Kleinbuchstaben. So steht beispielsweise der Bezeichner `MACHINE` für ein Token und der Bezeichner `actionCode` für ein Nichtterminal. Aus Gründen der besseren Lesbarkeit sind alle Token, die aus einer festen Zeichenfolge (wie einem einzigen Zeichen) bestehen, direkt als String in Hochkommata angegeben. Das Token `EOF` ist ein von ANTLR vordefiniertes Token, welches dafür sorgt, dass die Regel für das Startsymbol eine komplette Datei parsen muss und nicht nur einen Präfix verarbeiten darf [50].

Listing A.2: Lexer-Regeln zum Parsen einer Spezifikation

```
ACTION : [aA][cC][tT][iI][oO][nN];
ACTIONREFERENCE : ([>$@%] | ([><$%@] | '<>') ('~' | '*'))
    [␣\t]* IDENTIFIER;
CCLASSCHARACTER : ~[-^\]] | '0x' [0-9a-fA-F] [0-9a-fA-F];
CHARACTERSTRING : '\'' ~[\']+ '\'' | '"' ~["]+ '"';
CODEFRAME : (~'%' | '%' ~'%' | '%%' ~'{')+;
CODELINE : (~[}\r\n] | '\\}')+;
IDENTIFIER : [a-zA-Z] [a-zA-Z0-9_]*;
INT : [0-9]+;
MACHINE : [mM][aA][cC][hH][iI][nN][eE];
RULEREFERENCE : IDENTIFIER;
SEMANTICPREDICATE : [wW][hH][eE][nN] [␣\t]+ '!'? IDENTIFIER;
SINGLECHARACTER : '0x' [0-9a-fA-F] [0-9a-fA-F];
```

In Listing A.2 sind die Lexer-Regeln für die Token ohne Berücksichtigung der verschiedenen Lexer-Modi in lexikographischer Reihenfolge angegeben, sofern sie nicht direkt aus einer festen Zeichenkette bestehen. An den Regeln `ACTION` oder `MACHINE` ist beispielsweise erkennbar, dass Schlüsselwörter einer Spezifikation beliebig mit Klein- und Großbuchstaben geschrieben werden können. Zudem verwenden einige Regeln des Lexers (wie z. B. die Regel `SEMANTICPREDICATE`) wiederum andere Regeln, was bei ANTLR problemlos möglich ist. Der Operator ~ innerhalb der Regeln negiert das folgende Zeichen bzw. die folgende Zeichenklasse und kann nur auf Zeichenketten der Länge eins angewendet werden [50].

Neben den angegebenen Lexer-Regeln werden einige weitere Regeln verwendet, die Leerzeichen, Tabulatoren und Zeilenumbrüche der Spezifikation erkennen. Diese Zeichen werden nicht als Token an den Parser weitergegeben (indem sie in einen anderen Kanal geleitet werden), sodass zwischen den Token der Regeln beliebig viele Leerzeichen als Trennzeichen verwendet werden können. Ähnlich verhält es sich mit Kommentaren in der Spezifikation, deren Token ebenfalls nicht an den Parser weitergereicht werden, sondern wie Leerzeichen in einen zweiten Kanal geleitet werden. Einzeilige Kommentare in der

Spezifikation beginnen mit `//` oder `#`, mehrzeilige Kommentare beginnen mit `/*` und enden mit `*/`.

Aus der kontextfreien Grammatik für den generierten ANTLR-Parser lassen sich auch die Prioritäten der Operatoren für die Spezifikation von regulären Ausdrücken ableiten. Anhand der Grammatik werden die Operatoren als Knoten in einem Parsebaum angeordnet. Die Interpretation und Auswertung der Knoten des Parsebaums beginnt bei den Blättern und landet erst am Ende bei der Wurzel, sodass tiefer stehende Knoten mit ihren Operatoren stärker binden. Somit weist beispielsweise die Gruppierung die höchste Priorität auf.

Für die implementierte, experimentelle Version eines $LL(1)$-Parsers für FPGAs, bei dem lediglich einzelne Zeichen als Token verwendet werden können, sind in der ANTLR-Grammatik weitere Regeln enthalten, um kontextfreie Grammatiken parsen zu können. Entsprechend existieren Lexer-Regeln für Token von kontextfreien Grammatiken. Die Einbindung der Parserregeln für kontextfreie Grammatiken erfolgt über das oben aufgeführte Nichtterminal `specificationBlock`, welches nach beliebig vielen regulären Ausdrücken (anhand des Nichtterminals `specificationRule`) auch beliebig viele Regeln einer kontextfreien Grammatik parsen kann. Das Startsymbol einer kontextfreien Grammatik wird durch die letzte Regel der Grammatik festgelegt.

Anhang B

Verfahren zur effizienten Berechnung der Relation $\equiv_R$

Die effiziente Berechnung der Äquivalenzrelation $\equiv_R$ (bzw. $\equiv_L$) zur Berechnung eines Äquivalenz-reduzierten Automaten ist mit Hilfe eines Algorithmus für das *Multiple Relational Coarsest Partitioning Problem* (kurz *MRCP-Problem*) möglich [31]. Im Folgenden wird eine für endliche Automaten angepasste Formulierung des MRCP-Problems betrachtet. Zur Beschreibung der Problemstellung und eines effizienten Algorithmus werden zunächst einige Begriffe formal definiert.

B.1 Definition (nach [18, 48]). Sei $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ ein NFA, P eine Partition von Q und $T \subseteq Q$ eine Teilmenge der Zustände von $\mathcal{A}$. Für jedes Zeichen $\sigma \in \Sigma$ sei $D_\sigma^{-1}(T) = \{q \in Q \mid \exists p \in T : p \in \delta(q, \sigma)\}$ die Menge der σ-Vorgänger von T. Ein Block B der Partition P heißt genau dann *σ-stabil bzgl. T*, wenn $B \subseteq D_\sigma^{-1}(T)$ oder $B \cap D_\sigma^{-1}(T) = \emptyset$ ist. Somit liegt ein bzgl. T σ-stabiler Block $B \in P$ entweder komplett in den σ-Vorgängern von T oder komplett außerhalb dieser σ-Vorgänger. Die Partition P von Q heißt genau dann *σ-stabil bzgl. T*, wenn alle Blöcke $B \in P$ σ-stabil bzgl. T sind. Die Partition P heißt genau dann *stabil bzgl. T*, wenn P für alle Symbole $\sigma \in \Sigma$ σ-stabil bzgl. T ist. Die Partition P heißt genau dann *stabil*, wenn P bzgl. aller seiner Blöcke $B \in P$ stabil ist.

Nach der Definition ist eine Partition P der Zustandsmenge Q eines NFAs genau dann stabil, wenn für alle Zeichen $\sigma \in \Sigma$ und alle Blöcke $B_1, B_2 \in P$ gilt, dass B_1 σ-stabil bzgl. B_2 ist und B_1 damit komplett innerhalb oder komplett außerhalb der σ-Vorgänger von B_2 liegt. Mit Hilfe der Begriffe aus Definition B.1 kann das MRCP-Problem definiert werden: Das *Multiple Relational Coarsest Partitioning Problem für Automaten* bestimmt für einen NFA $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ und eine Partition P von Q die gröbste, stabile Verfeinerung[1] von P [18]. Der Begriff der Stabilität bezieht sich dabei auf die vorherige Definition.

[1] Seien P_1, P_2 zwei Partitionen einer Menge U. Dann ist P_1 eine *Verfeinerung* von P_2 und P_2 *gröber* als P_1, wenn $\forall B_1 \in P_1 \exists B_2 \in P_2 : B_1 \subseteq B_2$ gilt [18].

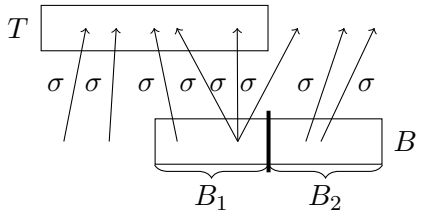


Abbildung B.1: Unterteilung eines Blocks B bzgl. T und σ in B_1 und B_2 (nach [18])

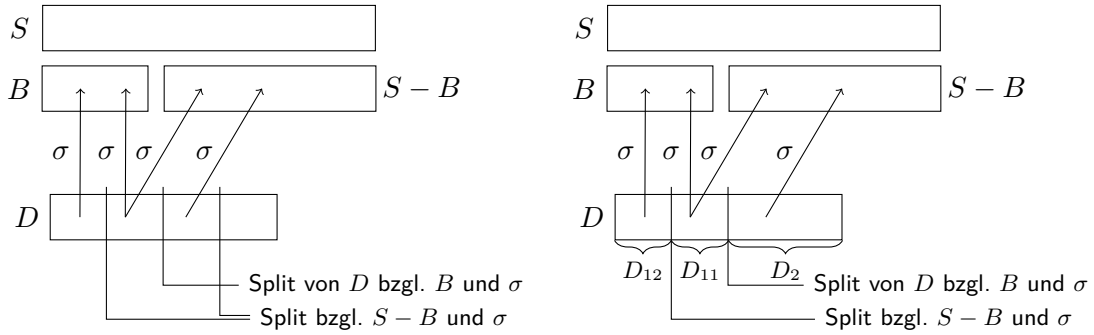
Der im Folgenden vorgestellte Algorithmus für das MRCP-Problem auf Automaten verfeinert die Partition P, die Teil der Eingabe ist, schrittweise, bis sie dem gesuchten Ergebnis entspricht. In einem Schritt wird jeder Block B der Partition P anhand einer Teilmenge $T \subseteq Q$ der Zustände und eines Symbols $\sigma \in \Sigma$ zur Sicherstellung der σ-Stabilität bzgl. T in $B_1 = B \cap D_\sigma^{-1}(T)$ und $B_2 = B - D_\sigma^{-1}(T)$ unterteilt (siehe Abbildung B.1). Sofern einer der beiden Teile B_1 oder B_2 leer ist, bleibt B unverändert. Diese Operation, die für alle Blöcke B von P ausgeführt wird, wird insgesamt mit $split(P, T, \sigma)$ bezeichnet und ist eine Verallgemeinerung des Verfeinerungsschrittes aus dem Minimierungsalgorithmus von Hopcroft [29] für DFAs. Die Menge T wird als *Splitter* von P bezeichnet, wenn $split(P, T, \sigma) \neq P$ für mindestens ein $\sigma \in \Sigma$ ist [18].

Innerhalb des Algorithmus für das MRCP-Problem werden aus Gründen der Laufzeiteffizienz nur solche Splitter gesucht, die eine Vereinigung mehrerer Blöcke der Partition P sind. Die Partition wird dann durch das Ergebnis von zwei Split-Operationen ersetzt. Dies geschieht solange, bis es keinen solchen Splitter mehr gibt [48]. Für eine effiziente Implementierung müssen die gesuchten Splitter geschickt bestimmt werden können. Dazu wird neben P eine zweite Partition X verwendet, sodass P stets eine Verfeinerung von X ist und P bzgl. jedem Block aus X stabil ist. Initial enthält X nur einen einzigen Block für die gesamte Zustandsmenge Q. Das Verfahren wiederholt den in Algorithmus B.1 angegebenen Verfeinerungsschritt solange, bis schließlich $P = X$ ist. Die Wahl von B als Block von P mit einer Größe von maximal $\frac{|S|}{2}$ trägt zur Laufzeiteffizienz bei (weitere Details sind in [48] zu finden). Vor und nach dem Verfeinerungsschritt gilt die Invariante, dass P

Eingabe: NFA $\mathcal{A} = (Q, \Sigma, \delta, s, F)$, Partitionen P, X von Q
Ausgabe: Partitionen P, X

$S \leftarrow$ Block von X, der kein Block von P ist und mehrere Blöcke von P enthält
$B \leftarrow$ Block von P mit $B \subseteq S$ und $|B| \leq \frac{|S|}{2}$
$X \leftarrow X - \{S\} \cup \{B, S - B\}$
for all $\sigma \in \Sigma$ **do**
  $P \leftarrow split(split(P, B, \sigma), S - B, \sigma)$
end for

Algorithmus B.1: Verfeinerungsschritt für das MCRP-Problem auf Automaten (nach [48])



(a) Allgemeine Situation

(b) Three-way Splitting

Abbildung B.2: Split von D bzgl. B und σ sowie $S - B$ und σ (nach [18])

stabil bzgl. aller Blöcke aus X ist und P eine Verfeinerung von X ist [18].

In einem Verfeinerungsschritt werden für jedes Symbol $\sigma \in \Sigma$ zwei Split-Operationen durchgeführt, die einen Block von P in maximal vier Teile unterteilen können. Allerdings sind dabei maximal drei der entstehenden Teile nicht leer, wie das folgende Lemma zeigt.

B.2 Lemma (nach [48]). *Sei P eine Partition von Q, die stabil bzgl. einer Menge $S \subseteq Q$ sei, welche wiederum die Vereinigung mehrerer Blöcke aus P sei. Angenommen, P wird zunächst bzgl. eines Blockes $B \subseteq S$ von P und dann bzgl. $S - B$ für ein Symbol $\sigma \in \Sigma$ verfeinert. Dann gelten die folgenden Aussagen:*

1. *Die Verfeinerung von P bzgl. B und σ ($split(P, B, \sigma)$) teilt einen Block $D \in P$ in zwei Teile $D_1 = D \cap D_\sigma^{-1}(B)$ und $D_2 = D - D_1$ genau dann, wenn $D \cap D_\sigma^{-1}(B) \neq \emptyset$ und $D - D_\sigma^{-1}(B) \neq \emptyset$ gilt.*

2. *Die Verfeinerung von $split(P, B, \sigma)$ bzgl. $S-B$ und σ ($split(split(P, B, \sigma), S-B, \sigma)$) teilt D_1 in zwei Teile $D_{11} = D_1 \cap D_\sigma^{-1}(S - B)$ und $D_{12} = D_1 - D_{11}$ genau dann, wenn $D_1 \cap D_\sigma^{-1}(S - B) \neq \emptyset$ und $D_1 - D_\sigma^{-1}(S - B) \neq \emptyset$ gilt.*

3. *Das Verfeinern von $split(P, B, \sigma)$ bzgl. $S - B$ und σ unterteilt D_2 nicht.*

4. $D_{12} = D_1 \cap (D_\sigma^{-1}(B) - D_\sigma^{-1}(S - B))$

Somit wird jeder Block D der Partition P durch die beiden Split-Operationen in maximal drei Teile unterteilt und nicht, wie bei den beiden Splits anzunehmen, in vier Teile. Dies wird in der englischsprachigen Literatur als *three-way splitting* bezeichnet [48] und ist durch die (wegen der Invariante) vorliegende Stabilität von D bzgl. S begründet. Schließlich gilt entweder $D_\sigma^{-1}(S) \cap D = \emptyset$ (und damit $D_\sigma^{-1}(B) \cap D = \emptyset = D_\sigma^{-1}(S-B) \cap D$), sodass D nicht unterteilt wird, oder es gilt $D \subseteq D_\sigma^{-1}(S) = D_\sigma^{-1}(B) \cup D_\sigma^{-1}(S - B)$ und führt zum Three-way Split [18]. In Abbildung B.2 ist die allgemeine Situation (ohne Stabilität von D bzgl. S) und das Three-way Splitting von D grafisch dargestellt. Mit Verwendung geeig-

neter Datenstrukturen können die beiden Split-Operationen eines Verfeinerungsschrittes effizient implementiert werden. Die Details zu den benötigten Datenstrukturen sind in [48] angegeben und werden hier nicht weiter aufgeführt.

Der vollständige Algorithmus für das MRCP-Problem für Automaten beginnt mit einer Vorverarbeitung und führt anschließend die bereits genannten Verfeinerungsschritte durch. Die Vorverarbeitung verändert die Partition P, sodass sie anschließend stabil bzgl. Q ist [18]. Diese Vorverarbeitung ist erforderlich, da in einem NFA $\mathcal{A}$ nicht jeder Zustand für jedes Zeichen $\sigma \in \Sigma$ eine ausgehende Transition besitzen muss, das von [48] auf Automaten übertragene Verfahren aber von vollständigen Automaten ausgeht. Das komplette Verfahren ist in Algorithmus B.2 angegeben. Die Laufzeit des Algorithmus liegt für einen

Eingabe: NFA $\mathcal{A} = (Q, \Sigma, \delta, s, F)$, Partition P von Q
Ausgabe: gröbste, stabile Verfeinerung von P

$X \leftarrow \{Q\}$
for all $\sigma \in \Sigma$ **do**
  $P \leftarrow split(P, Q, \sigma)$
end for
while $\exists S \in X$, sodass S mehr als einen Block von P enthält **do**
  Verfeinerungsschritt nach Algorithmus B.1
end while
return P

Algorithmus B.2: Algorithmus für das MRCP-Problem für Automaten [18, 48]

NFA $\mathcal{A}$ mit n Zuständen und m Transitionen mit Verwendung geeigneter Datenstrukturen bei $\mathcal{O}(|\Sigma| \times m \log n)$ [48].

Zur Berechnung der Relation $\equiv_R$ für einen NFA $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ wird der Algorithmus für das MRCP-Problem auf Automaten mit den Eingaben $\mathcal{A}$ und $P = \{F, Q - F\}$ aufgerufen. Das Ergebnis des Algorithmus entspricht den Äquivalenzklassen der gesuchten Relation $\equiv_R$ [31]. Die Äquivalenzklassen von $\equiv_L$ können mit dem Algorithmus entsprechend auf dem reversen Automaten $\mathcal{A}^r$ berechnet werden. Für die im implementierten Parsergenerator verwendeten *Action*-NFAs (ohne Epsilon-Transitionen) funktioniert die Berechnung der Relationen $\equiv_L$ und $\equiv_R$ praktisch analog. In der Vorverarbeitung und den Verfeinerungsschritten werden statt den Zeichen $\sigma \in \Sigma$ lediglich alle Transitionslabel t mit $t \in 2^{\Sigma} \times 2^{Pr} \times 2^{A}$ für einen *Action*-NFA $\mathcal{B} = (Q, \Sigma \cup Pr \cup A, \delta, s, F)$ verwendet.

Die Berechnung der Relation $\equiv_R$ mit dem vorgestellten Algorithmus wird anhand eines Beispiels verdeutlicht.

B.3 Beispiel. Betrachtet wird der NFA $\mathcal{A}$ aus Abbildung B.3, für den die Relation $\equiv_R$ berechnet werden soll. Der Aufruf des Algorithmus B.2 erfolgt mit dem NFA $\mathcal{A}$ und der initialen Partition $P = \{\{q_8, q_{11}\}, \{q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_9, q_{10}\}\}$ der Zustandsmenge Q.

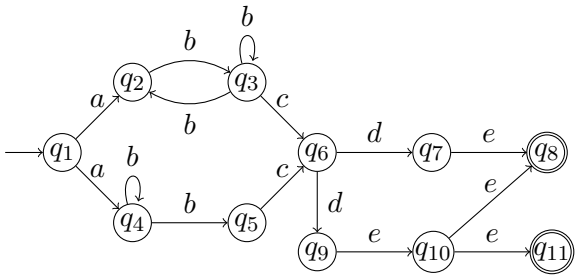


Abbildung B.3: Beispiel: Automat $\mathcal{A}$ zur Berechnung von $\equiv_R$

Die Vorverarbeitung des Algorithmus verfeinert die Partition P anhand der Zeichen a, b, c, d und e des Alphabets und führt zu den Partitionen X und P mit

$$X = \{\{q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}\}\} \text{ und}$$
$$P = \{\{q_8, q_{11}\}, \{q_1\}, \{q_2, q_4\}, \{q_3\}, \{q_5\}, \{q_6\}, \{q_7, q_9, q_{10}\}\}.$$

Im ersten Verfeinerungsschritt werden $S = \{q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}\}$ und $B = \{q_8, q_{11}\}$ gewählt. Der Algorithmus modifiziert die Partitionen X und P zu

$$X = \{\{q_8, q_{11}\}, \{q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_9, q_{10}\}\} \text{ und}$$
$$P = \{\{q_8, q_{11}\}, \{q_1\}, \{q_2, q_4\}, \{q_3\}, \{q_5\}, \{q_6\}, \{q_7, q_{10}\}, \{q_9\}\}.$$

Durch die Split-Operationen wird der Block $\{q_7, q_9, q_{10}\}$ von P anhand des Zeichens $e \in \Sigma$ wegen $D_e^{-1}(\{q_8, q_{11}\}) = \{q_7, q_{10}\}$ in die beiden Teilmengen $\{q_7, q_{10}\}$ und $\{q_9\}$ unterteilt. Dadurch gilt die Invariante, dass P stabil bzgl. aller Blöcke aus X ist, weiterhin.

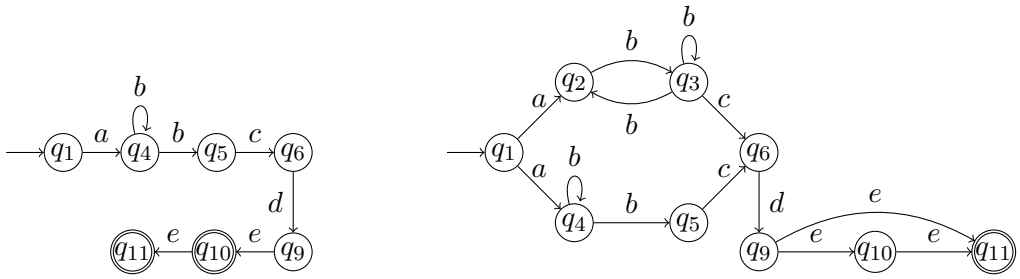
Der zweite Verfeinerungsschritt führt mit $S = \{q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_9, q_{10}\}$ und $B = \{q_2, q_4\}$ zu den verfeinerten Partitionen

$$X = \{\{q_8, q_{11}\}, \{q_2, q_4\}, \{q_1, q_3, q_5, q_6, q_7, q_9, q_{10}\}\} \text{ und}$$
$$P = \{\{q_8, q_{11}\}, \{q_1\}, \{q_2\}, \{q_4\}, \{q_3\}, \{q_5\}, \{q_6\}, \{q_7, q_{10}\}, \{q_9\}\}.$$

Dabei wird mit $D_b^{-1}(\{q_2, q_4\}) = \{q_3, q_4\}$ der Block $\{q_2, q_4\}$ von P in die beiden Teile $\{q_2\}$ und $\{q_4\}$ geteilt, um die Stabilität von P bzgl. der Blöcke aus X zu gewährleisten.

Die folgenden sechs Verfeinerungsschritten verändern die Partition P nicht weiter und unterteilen lediglich die Blöcke $\{q_1, q_3, q_5, q_6, q_7, q_9, q_{10}\}$ und $\{q_2, q_4\}$ der Partition X schrittweise in die Blöcke $\{q_1\}$, $\{q_3\}$, $\{q_5\}$, $\{q_6\}$, $\{q_7, q_{10}\}$ und $\{q_9\}$ sowie $\{q_2\}$ und $\{q_4\}$. Schließlich ist $X = P$ und der Algorithmus gibt P aus. Die Blöcke von P entsprechen den Äquivalenzklassen der gesuchten Relation $\equiv_R$ für den Automaten $\mathcal{A}$.

Am Beispiel des NFAs $\mathcal{A}$ aus Abbildung B.3 lässt sich auch gut erkennen, dass das Verfahren zur Äquivalenz-basierten Reduzierung im Allgemeinen keinen minimalen NFA liefert. Beispielsweise gilt $\mathcal{L}_L(\mathcal{A}, q_3) = \mathcal{L}_L(\mathcal{A}, q_5)$, aber die Relation $\equiv_L$ enthält das Tupel



(a) Minimaler NFA für $\mathcal{A}$

(b) Äquivalenz-reduzierter NFA für $\mathcal{A}$

Abbildung B.4: Beispiel: Vergleich des minimalen NFAs mit dem Äquivalenz-reduzierten NFA

(q_3, q_5) nicht. Ebenso gilt $\mathcal{L}_R(\mathcal{A}, q_2) = \mathcal{L}_R(\mathcal{A}, q_4)$ und $q_2 \not\equiv_R q_4$. Hintergrund ist, dass bei der Äquivalenz-basierten Reduzierung keine mehrelementigen Mengen aktiver Zustände betrachtet werden, sondern stets nur Entscheidungen bzgl. der Ununterscheidbarkeit auf Basis einzelner Zustände getroffen werden[2]. Schließlich wird durch das Verfahren lediglich eine Teilmenge ununterscheidbarer Zustände erkannt [33, 34]. Ein minimaler NFA für $\mathcal{A}$ kommt u. a. ohne die Zustände q_2 und q_3 aus. Der Äquivalenz-reduzierte NFA für $\mathcal{A}$ enthält die Zustände q_2, q_3, q_4 und q_5 jedoch weiterhin in unveränderter Form. Der minimale NFA sowie der Äquivalenz-reduzierte NFA zu dem Automaten $\mathcal{A}$ aus Abbildung B.3 sind in den Abbildungen B.4(a) bzw. B.4(b) dargestellt.

[2]Dies gilt auch für die Quasiordnung-basierte Reduzierung, die die genannten ununterscheidbaren Zustände ebenfalls nicht erkennt.

Anhang C

Messdaten der Evaluation für Snort-Regelsätze

Die exakten Messungen für die reine UDP-Verarbeitung ohne Parser, die in der Evaluierung als Referenzgröße verwendet wird, sind in Tabelle C.1 angegeben. Dabei ist (wie im Kapitel zur Evaluation angegeben) anzumerken, dass die Messungen Schwankungen unterliegen können und wiederholte Messungen zu leicht veränderten Werten führen können. Bei der Synthese wurden für die gemessenen Werte die gleichen Optimierungen des Synthesetools verwendet wie bei den Messungen für die Parser der Snort-Regelsätze.

Tabelle C.1: Ressourcenbedarf für die UDP-Verarbeitung auf einem Virtex-5 FPGA

Ressource	Ressourcenbedarf
Lookup-Tabellen	1675
Register	1340
Slices	619

Die exakten Messungen des Ressourcenbedarfs für die Snort-Regelsätze aus der Evaluation sind in Tabelle C.2 angegeben. Die Tabelle enthält die Messdaten für minimale DFAs, NFAs ohne Optimierungen, Äquivalenz-reduzierte NFAs und für FPGAs optimierte NFAs. Dabei bedeutet die Angabe „nicht bekannt“ bei den DFAs einiger Regelsätze, dass keine Daten vorliegen, weil die Berechnung der DFAs durch Snowfall oder die Synthese auf dem verwendeten Virtex-5 FPGA von Xilinx nicht in akzeptabler Laufzeit möglich ist. Alternativ kann auch der erwartete Ressourcenbedarf der Pattern Matcher mit deterministischen Automaten die Ressourcen des Virtex-5 XC5VLX110T-1FF336 FPGAs von Xilinx übersteigen und der Grund für die fehlenden Daten sein.

Tabelle C.2: Ressourcenbedarf (LUTs und Register) der Snort-Regelsätze auf einem Virtex-5 FPGA

Regelsatz	DFA		NFA		NFA (reduziert)		NFA (optimiert)	
	LUTs	Register	LUTs	Register	LUTs	Register	LUTs	Register
ActiveX	nicht bekannt		6946	6706	4267	4040	4276	4040
Attack-Response	2864	2213	1667	1416	1664	1415	1669	1416
Chat	1842	1462	1653	1393	1635	1375	1635	1375
DNS	1622	1360	1614	1356	1613	1355	1619	1356
Dos	nicht bekannt		5265	4938	5018	4697	2323	1870
Exploit	nicht bekannt		12440	12211	10698	10478	3814	3233
FTP	nicht bekannt		4409	4157	4256	4018	2264	1883
Games	nicht bekannt		1862	1602	1862	1602	1630	1358
IMAP	nicht bekannt		3527	3270	3475	3218	1946	1599
Misc	nicht bekannt		1766	1501	1765	1500	1654	1383
Mobile-Malware	nicht bekannt		2213	1929	2044	1771	2020	1712
Netbios	nicht bekannt		1684	1542	1647	1510	1636	1374
P2P	1900	1477	1641	1379	1638	1378	1638	1378
Policy	nicht bekannt		3460	3187	2796	2531	2760	2494
POP3	nicht bekannt		2345	2085	2327	2068	1747	1456
Scada	3218	2039	2303	2028	1905	1632	1909	1632
Scan	nicht bekannt		2269	1987	2127	1842	2034	1744
Shellcode	3255	2270	1746	1470	1739	1463	1713	1440
SMTP	nicht bekannt		2251	1984	2221	1960	1714	1431
User-Agents	3201	2051	1893	1552	1828	1484	1824	1478
Voip	nicht bekannt		2696	2359	2692	2356	1745	1370
Worm	1720	1373	1626	1365	1626	1364	1626	1365

Abbildungsverzeichnis

Algorithmenverzeichnis

Literaturverzeichnis

[1] Aho, Alfred V., Monica S. Lam, Ravi Sethi und Jeffrey D. Ullman: Compilers: Principles, Techniques, and Tools. Pearson Addison-Wesley, Boston, MA, USA, 2. Auflage, 2007.

[2] Becchi, Michela: Data Structures, Algorithms and Architectures for Efficient Regular Expression Evaluation. Doktorarbeit, Washington University in St. Louis, 2009.

[3] Becchi, Michela und Patrick Crowley: Efficient regular expression evaluation: theory to practice. In: Franklin, Mark A., Dhabaleswar K. Panda und Dimitrios Stiliadis (Herausgeber): Proceedings of the 2008 ACM/IEEE Symposium on Architecture for Networking and Communications Systems, ANCS 2008, San Jose, California, USA, November 6-7, 2008, Seiten 50–59. ACM, 2008.

[4] Berge, Claude: Graphs and Hypergraphs. North-Holland Publ. Comp, Amsterdam, 2. Auflage, 1973.

[5] Bispo, João, Ioannis Sourdis, João M. P. Cardoso und Stamatis Vassiliadis: Regular expression matching for reconfigurable packet inspection. In: Constantinides, George A., Wai-Kei Mak, Phaophak Sirisuk und Theerayod Wiangtong (Herausgeber): 2006 IEEE International Conference on Field Programmable Technology, FPT 2006, Bangkok, Thailand, December 13-15, 2006, Seiten 119–126. IEEE, 2006.

[6] Cascarano, Niccolo, Pierluigi Rolando, Fulvio Risso und Riccardo Sisto: iNFAnt: NFA pattern matching on GPGPU devices. Computer Communication Review, 40(5):20–26, 2010.

[7] Champarnaud, Jean-Marc und Fabien Coulon: NFA Reduction Algorithms by Means of Regular Inequalities. Theor. Comput. Sci., 327(3):241–253, November 2004.

[8] Champarnaud, Jean-Marc und Fabien Coulon: Erratum to NFA reduction algorithms by means of regular inequalities"[TCS 327 (2004) 241-253]. Theor. Comput. Sci., 347(1-2):437–440, 2005.

[9] Cho, Young H. und William H. Mangione-Smith: High-Performance Context-Free Parser for Polymorphic Malware Detection. Advanced Networking and Communications Hardware Workshop, 2005.

[10] Ciressan, Christian, Eduardo Sanchez, Martin Rajman und Jean-Cédric Chappelier: An FPGA-Based Coprocessor for the Parsing of Context-Free Grammars. In: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '00, Seiten 236–245, Washington, DC, USA, 2000. IEEE Computer Society.

[11] Ciressan, Cristian, Eduardo Sanchez, Martin Rajman und Jean-Cédric Chappelier: An FPGA-Based Syntactic Parser for Real-Life Almost Unrestricted Context-Free Grammars. In: Proceedings of the 11th International Conference on Field-Programmable Logic and Applications, FPL '01, Seiten 590–594, London, UK, UK, 2001. Springer-Verlag.

[12] Cisco: Snort. `https://snort.org/`, 2016. Letzter Zugriff: 20.02.2016.

[13] Colm Networks: Ragel State Machine Compiler. `http://www.colm.net/open-source/ragel/`. Letzter Zugriff: 21.02.2016.

[14] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest und Clifford Stein: Introduction to Algorithms (3. ed.). MIT Press, 2009.

[15] Dimopoulos, Alexandros C., Christos Pavlatos, Panagiota Karanasou und George Papakonstantinou: A generic platform for the SoC implementation of grammar-based applications. VLSI-SoC 2008: Proceedings of the 16th international conference on Very Large Scale Integration, October 13-15 2008, 2008.

[16] Dlugosch, Paul, Dave Brown, Paul Glendenning, Michael Leventhal und Harold Noyes: An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing. IEEE Trans. Parallel Distrib. Syst., 25(12):3088–3098, 2014.

[17] Domaratzki, Michael und Kai Salomaa: Lower bounds for the transition complexity of NFAs. J. Comput. Syst. Sci., 74(7):1116–1130, 2008.

[18] Estenfeld, Klaus, Hans-Albert Schneider, Dirk Taubner und Erik Tidén: Computer Aided Verification of Parallel Processes. In: Pfitzmann, Andreas und Eckart Raubold (Herausgeber): VIS'91, Verläßliche Informationssysteme, GI-Fachtagung, Darmstadt, 13.-15. März 1991, Proceedings, Band 271 der Reihe Informatik-Fachberichte, Seiten 208–226. Springer, 1991.

[19] Faezipour, Miad und Mehrdad Nourani: Constraint Repetition Inspection for Regular Expression on FPGA. In: 16th IEEE Symposium on High Performance Interconnects, 2008., Seiten 111–118, August 2008.

[20] Floyd, Robert W. und Jeffrey D. Ullman: The Compilation of Regular Expressions into Integrated Circuits. J. ACM, 29(3):603–622, 1982.

[21] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[22] Glushkov, V. M.: The abstract theory of automata. Russian Mathematical Surveys, 16(5):1–53, 1961.

[23] Gramlich, Gregor und Georg Schnitger: Minimizing Nfa's and Regular Expressions. J. Comput. Syst. Sci., 73(6):908–923, September 2007.

[24] Gruber, Hermann und Markus Holzer: Inapproximability of Nondeterministic State and Transition Complexity Assuming P=!NP. In: Harju, Tero, Juhani Karhumäki und Arto Lepistö (Herausgeber): Developments in Language Theory, 11th International Conference, DLT 2007, Turku, Finland, July 3-6, 2007, Proceedings, Band 4588 der Reihe Lecture Notes in Computer Science, Seiten 205–216. Springer, 2007.

[25] Hauck, Scott und André Dehon (Herausgeber): Reconfigurable Computing. Systems on Silicon. Morgan Kaufmann, Burlington, 2007.

[26] Hazel, Philip: PCRE - Perl Compatible Regular Expressions. `http://pcre.org/`. Letzter Zugriff: 20.02.2016.

[27] Holzer, Markus und Martin Kutrib: Nondeterministic Descriptional Complexity Of Regular Languages. Int. J. Found. Comput. Sci., 14(6):1087–1102, 2003.

[28] Holzer, Markus und Martin Kutrib: Nondeterministic Finite Automata-Recent Results on the Descriptional and Computational Complexity. In: Ibarra, Oscar H. und Bala Ravikumar (Herausgeber): Implementation and Applications of Automata, 13th International Conference, CIAA 2008, San Francisco, California, USA, July 21-24, 2008. Proceedings, Band 5148 der Reihe Lecture Notes in Computer Science, Seiten 1–16. Springer, 2008.

[29] Hopcroft, John: An n log n algorithm for minimizing states in a finite automaton. In: Kohavi, Zvi und Azaria Paz (Herausgeber): Theory of Machines and Computations, Seiten 189–196. Academic Press, 1971.

[30] Hopcroft, John E., Rajeev Motwani und Jeffrey D. Ullman: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, 3. Auflage, 2007.

[31] Ilie, Lucian, Gonzalo Navarro und Sheng Yu: On NFA Reductions. In: Karhumäki, Juhani, Hermann A. Maurer, Gheorghe Paun und Grzegorz Rozenberg (Herausgeber): Theory Is Forever, Essays Dedicated to Arto Salomaa on the Occasion of His 70th Birthday, Band 3113 der Reihe Lecture Notes in Computer Science, Seiten 112–124. Springer, 2004.

[32] Ilie, Lucian, Roberto Solis-Oba und Sheng Yu: Reducing the Size of NFAs by Using Equivalences and Preorders. In: Apostolico, Alberto, Maxime Crochemore und Kunsoo Park (Herausgeber): Combinatorial Pattern Matching, Band 3537 der Reihe Lecture Notes in Computer Science, Seiten 310–321. Springer Berlin Heidelberg, 2005.

[33] Ilie, Lucian und Sheng Yu: Algorithms for Computing Small NFAs. In: Diks, Krzysztof und Wojciech Rytter (Herausgeber): Mathematical Foundations of Computer Science 2002, 27th International Symposium, MFCS 2002, Warsaw, Poland, August 26-30, 2002, Proceedings, Band 2420 der Reihe Lecture Notes in Computer Science, Seiten 328–340. Springer, 2002.

[34] Ilie, Lucian und Sheng Yu: Reducing NFAs by invariant equivalences. Theor. Comput. Sci., 306(1-3):373–390, 2003.

[35] Jiang, Tao und Bala Ravikumar: Minimal NFA Problems Are Hard. SIAM J. Comput., 22(6):1117–1141, Dezember 1993.

[36] Kameda, Tsunehiko und Peter Weiner: On the State Minimization of Nondeterministic Finite Automata. IEEE Transactions on Computers, 19(7):617–627, 1970.

[37] Karp, Richard M.: Reducibility Among Combinatorial Problems. In: Miller, Raymond E. und James W. Thatcher (Herausgeber): Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York., The IBM Research Symposia Series, Seiten 85–103. Plenum Press, New York, 1972.

[38] Ko, Sang-Ki und Yo-Sub Han: Left is Better than Right for Reducing Nondeterminism of NFAs. In: Holzer, Markus und Martin Kutrib (Herausgeber): Implementation and Application of Automata - 19th International Conference, CIAA 2014, Giessen, Germany, July 30 - August 2, 2014. Proceedings, Band 8587 der Reihe Lecture Notes in Computer Science, Seiten 238–251. Springer, 2014.

[39] Košař, Vlastimil, Martin Žádník und Jan Kořenek: NFA Reduction for Regular Expressions Matching Using FPGA. In: 2013 International Conference on

Field-Programmable Technology, FPT 2013, Kyoto, Japan, December 9-11, 2013, Seiten 338–341. IEEE, 2013.

[40] Lin, Cheng-Hung, Chih-Tsun Huang, Chang-Ping Jiang und Shih-Chieh Chang: Optimization of Pattern Matching Circuits for Regular Expression on FPGA. IEEE Trans. VLSI Syst., 15(12):1303–1310, 2007.

[41] Malcher, Andreas: Minimizing finite automata is computationally hard. Theor. Comput. Sci., 327(3):375–390, 2004.

[42] Matz, Oliver und Andreas Potthoff: Computing Small Nondeterministic Finite Automata. In: Engberg, Uffe H., Kim G. Larsen und Arne Skou (Herausgeber): Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS (Aarhus, Denmark, 19–20 May, 1995), Notes Series, Seiten 74–88. BRICS, Mai 1995.

[43] McNaughton, R. und H. Yamada: Regular Expressions and State Graphs for Automata. IEEE Transactions on Electronic Computers, 9(1):39–47, März 1960.

[44] Melnikov, Boris F.: A new algorithm of the state-minimization for the nondeterministic finite automata. Korean Journal of Computational & Applied Mathematics, 6(2):277–290, 1999.

[45] Melnikov, Boris F.: Once more about the state-minimization of the nondeterministic finite automata. Korean Journal of Computational & Applied Mathematics, 7(3):655–662, 2000.

[46] Mulder, Michael und George S. Nezlek: Creating protein sequence patterns using efficient regular expressions in bioinformatics research. In: ITI 2006 Proceedings of the 28th International Conference on Information Technology Interfaces, Seiten 207–212, 2006.

[47] Paakki, Jukka: Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. ACM Comput. Surv., 27(2):196–255, 1995.

[48] Paige, Robert und Robert Endre Tarjan: Three Partition Refinement Algorithms. SIAM J. Comput., 16(6):973–989, 1987.

[49] Pangracious, Vinod, Zied Marrakchi und Habib Mehrez: Three-Dimensional Design Methodologies for Tree-based FPGA Architecture. Springer Publishing Company, Incorporated, 2015.

[50] Parr, Terence: The Definitive ANTLR 4 Reference. Pragmatic Bookshelf, 2. Auflage, 2013.

[51] Parr, Terence und Kathleen Fisher: LL(*): The Foundation of the ANTLR Parser Generator. SIGPLAN Not., 46(6):425–436, Juni 2011.

[52] Parr, Terence, Sam Harwell und Kathleen Fisher: Adaptive LL(*) Parsing: The Power of Dynamic Analysis. SIGPLAN Not., 49(10):579–598, Oktober 2014.

[53] Pavlatos, Christos, Alexandros C. Dimopoulos, Andrew Koulouris, Theodore Andronikos, Ioannis Panagopoulos und George Papakonstantinou: Efficient Reconfigurable Embedded Parsers. Comput. Lang. Syst. Struct., 35(2):196–215, Juli 2009.

[54] Ray, Soumya und Mark Craven: Learning Statistical Models for Annotating Proteins with Function Information using Biomedical Text. BMC Bioinformatics, 6(Supplement 1):S18, Mai 2005.

[55] Roesch, Martin: Snort: Lightweight Intrusion Detection for Networks. In: Parter, David W. (Herausgeber): Proceedings of the 13th Conference on Systems Administration (LISA-99), Seattle, WA, November 7-12, 1999, Seiten 229–238. USENIX, 1999.

[56] Rouchka, Eric C.: Pattern Matching Techniques and Their Applications to Computational Molecular Biology - A Review. Technischer Bericht WUCS-99-09, Computer Science and Engineering, Washington University in St. Louis, 1999.

[57] Salomaa, Kai: Descriptional Complexity of Nondeterministic Finite Automata. In: Harju, Tero, Juhani Karhumäki und Arto Lepistö (Herausgeber): Developments in Language Theory, 11th International Conference, DLT 2007, Turku, Finland, July 3-6, 2007, Proceedings, Band 4588 der Reihe Lecture Notes in Computer Science, Seiten 31–35. Springer, 2007.

[58] Sidhu, Reetinder P. S. und Viktor K. Prasanna: Fast Regular Expression Matching Using FPGAs. In: Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '01, Seiten 227–238, Washington, DC, USA, 2001. IEEE Computer Society.

[59] Sourdis, Ioannis, João Bispo, João M. P. Cardoso und Stamatis Vassiliadis: Regular Expression Matching in Reconfigurable Hardware. Signal Processing Systems, 51(1):99–121, 2008.

[60] Sutton, Peter: Partial character decoding for improved regular expression matching in FPGAs. In: Diessel, Oliver und John Williams (Herausgeber): Proceedings of the 2004 IEEE International Conference on Field-Programmable Technology, Brisbane, Australia, December 6-8, 2004, Seiten 25–32. IEEE, 2004.

[61] Teubner, Jens und Louis Woods: Snowfall: Hardware Stream Analysis Made Easy. In: Härder, Theo, Wolfgang Lehner, Bernhard Mitschang, Harald Schöning und Holger Schwarz (Herausgeber): Datenbanksysteme für Business, Technologie und Web (BTW), 14. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme"(DBIS), 2.-4.3.2011 in Kaiserslautern, Germany, Band 180 der Reihe LNI, Seiten 738–741. GI, 2011.

[62] Teubner, Jens und Louis Woods: Data Processing on FPGAs. Morgan & Claypool Publishers, 1. Auflage, 2013.

[63] The Bro Project: The Bro Network Security Monitor. `https://www.bro.org/`, 2014. Letzter Zugriff: 20.02.2016.

[64] Thompson, Ken: Regular Expression Search Algorithm. Commun. ACM, 11(6):419–422, Juni 1968.

[65] Thurston, Adrian: Ragel State Machine Compiler - User Guide. `http://www.colm.net/files/ragel/ragel-guide-6.9.pdf`. Version 6.9.

[66] Tsyganov, Andrey V.: Local Search Heuristics for NFA State Minimization Problem. International Journal of Communications, Network and System Sciences, 5(9A):638–643, 2012.

[67] Tsyganov, Andrey V.: ReFaM: A Software Tool for Minimizing Nondeterministic Finite Automata. In: Proceedings of the 2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC '12, Seiten 187–191, Washington, DC, USA, 2012. IEEE Computer Society.

[68] Vasiliadis, Giorgos, Michalis Polychronakis und Sotiris Ioannidis: Parallelization and characterization of pattern matching using GPUs. In: Proceedings of the 2011 IEEE International Symposium on Workload Characterization, IISWC 2011, Austin, TX, USA, November 6-8, 2011, Seiten 216–225. IEEE Computer Society, 2011.

[69] Woods, Louis, Jens Teubner und Gustavo Alonso: Complex Event Detection at Wire Speed with FPGAs. PVLDB, 3(1):660–669, 2010.

[70] Xilinx: Saving Costs with the SRL16E, Mai 2008. Rev. 1.0.

[71] Xilinx: Virtex-5 Family Overview, August 2015. Version 5.1.

[72] Yang, Yi-Hua E., Weirong Jiang und Viktor K. Prasanna: Compact Architecture for High-Throughput Regular Expression Matching on FPGA. In: Franklin, Mark A., Dhabaleswar K. Panda und Dimitrios Stiliadis (Herausgeber): Proceedings of the 2008 ACM/IEEE Symposium on Architecture for

Networking and Communications Systems, ANCS 2008, San Jose, California, USA, November 6-7, 2008, Seiten 30–39. ACM, 2008.

[73] Yang, Yi-Hua E. und Viktor K. Prasanna: High-Performance and Compact Architecture for Regular Expression Matching on FPGA. IEEE Trans. Computers, 61(7):1013–1025, 2012.

Eidesstattliche Versicherung

Schwedhelm, Heiko	140303
Name, Vorname	Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende ~~Bachelorarbeit~~/Masterarbeit mit dem Titel

Pattern Matching auf FPGAs

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Dortmund, den 1. März 2016	
Ort, Datum	Unterschrift

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/ die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Dortmund, den 1. März 2016	
Ort, Datum	Unterschrift