Universität Duisburg-Essen

Fakultät für Ingenieurwissenschaften

Master-Arbeit

im Studiengang Angewandte Informatik - Schwerpunkt Intelligente Technische Systeme und Wissenschaftliches Rechnen

zur Erlangung des akademischen Grades Master of Science

Thema:	Ein generisches Werkzeug für Sprachäquivalenz bei gewichteten Automaten
Autor:	Mika, Christine MatNr. 2242 695
Version vom:	8. Dezember 2015
Professorin:	Prof. Dr. König, Barbara
Betreuer:	M. Sc. Küpper, Sebastian

Inhaltsverzeichnis

1 Einleitung

Zustandsbasierte Systeme zeichnen sich dadurch aus, dass ihr Verhalten nicht nur von der Eingabe, sondern auch von den internen Zuständen abhängt. Interne Zustände sind für den Beobachter von außen nicht ersichtlich. Für einen Beobachter sind zwei Systeme dann äquivalent, wenn sie sich immer äquivalent verhalten. Die Frage, ob sich ein System oder ein bestimmter Systemabschnitt verglichen mit einem anderen System äquivalent verhält, spielt z.B. dann eine Rolle, wenn Programme zunächst entworfen und anschließend optimiert werden sollen. [1]
Zustandsbasierte Systeme können anhand von Transitionssystemen dargestellt werden. Transitionssysteme bestehen dabei in erster Linie aus den Zuständen und den Übergängen zwischen den Zuständen eines Systems. Die Automatentheorie, ein Teilgebiet der theoretischen Informatik, umfasst zustandsorientierte Modelle und deren Anwendungsbereiche. Ein Modell der Automatentheorie ist z.B. der gewichtete Automat. Gewichtete Automaten werden u.a. in der digitalen Text- und Sprachverarbeitung eingesetzt. Die Komposition von verschiedenen Transduktoren, endlichen Automaten, die eine Ausgabe erzeugen können, wird für die Repräsentation von phonetischen, akustischen und sprachlichen Informationen verwendet. Am Schluss wird ein gewichteter Automat eingefügt, der die Ausgaben überprüft. Um Rechenzeit und Speicherplatz zu sparen, werden Minimierungsalgorithmen für die gewichteten Automaten eingesetzt. [2]
Die Optimierung von zustandsbasierten Systemen erfolgt durch eine Minimierung der Zustände. Die Anzahl der Zustände kann dann verkleinert werden, wenn zwei Zustände verhaltensäquivalent sind. In zahlreichen Publikationen sind Algorithmen zur Minimierung verschiedenster Transitionssysteme zu finden. Der wohl bekannteste Algorithmus ist die Minimierung von deterministischen endlichen Automaten, die, anders als gewichtete Automaten, eine eindeutige minimale Darstellung besitzen. Die Frage nach generischen Ansätzen zur Minimierung von Transitionssystemen wird in [3] ausführlich untersucht. Neben einer Übersicht und Einführung in verschiedene Konzepte zur Minimierung von gewichteten Automaten enthält die Veröffentlichung von Barbara König und Sebastian Küpper eine Ausarbeitung eines generischen Partitionierungs-Algorithmus zur Überprüfung von Verhaltensäquivalenz verschiedener Transitionssysteme. Ähnliche Publikationen von Droste und Kuske zur Entscheidbarkeit von Sprachäquivalenz bei gewichteten Automaten beruhen auf der Arbeit von Schützenberger [4].
Der Algorithmus wird konkret für den Fall der gewichteten Automaten vorgestellt. Zunächst wird der Begriff der Verhaltensäquivalenz für gewichtete Automaten geklärt. Zwei Zustände eines gewichteten Automaten verhalten sich äquivalent, wenn

beide die gleiche gewichtete Sprache akzeptieren. Darüber hinaus wird bewiesen, dass Sprachäquivalenz von gewichteten Automaten mit der koalgebraischen Definition von Verhaltensäquivalenz zusammenfällt. Der Begriff der Koalgebra im Zusammenhang mit zustandsbasierten Systemen wird u.a. auch in [5] erläutert.
Der Algorithmus listet in gewisser Weise Wörter der Länge nach auf und berechnet für die einzelnen Zustände die Gewichte. Ein gewichteter Automat enthält zusätzlich zu den Beschriftungen der Transitionen noch Gewichte, die einem Semiring entnommen werden. Die Ergebnisse der Berechnungen in Abhängigkeit der Wortlänge werden in einer Matrix dargestellt. Im Falle der gewichteten Automaten basiert daher der Algorithmus zur Überprüfung der Sprachäquivalenz auf dem Lösen linearer Gleichungssysteme (LGS). Die Werte innerhalb eines linearen Gleichungssystems ergeben sich durch die Gewichte eines Automaten. Daher stellt sich die Frage, für welche gewichteten Automaten der Partitionierungs-Algorithmus eingesetzt werden kann. Ist ein Verfahren für einen beliebigen Semiring zur Lösung eines LGS der Form $Ax = b$ bekannt?
Zur Beantwortung dieser Frage werden zunächst verschiedene Semiringe betrachtet und es wird recherchiert, ob Lösungsverfahren bekannt sind. In Anbetracht der Vielfalt von Semiringen entsteht die Idee, einen Semiring-Generator zu implementieren, um Anwendern die Möglichkeit einzuräumen, Semiringe und ihnen bekannte Lösungsverfahren in das bestehende Programm einzufügen.

Die Arbeit unterteilt sich in zwei fachliche Hauptkapitel und zwei weitere zusammenfassende Kapitel. Im zweiten Kapitel werden zunächst die Grundlagen zu gewichteten Automaten vorgestellt. Eine Einführung wird durch die Definition formaler und gewichteter Sprachen gegeben. Im Anschluss an die klassische Definition von gewichteten Automaten wird die koalgebraische Sichtweise vorgestellt. Die zweite Definition basiert auf grundlegenden Begriffen der Kategorientheorie, die ebenfalls aufgeführt werden. Der Partitionsverfeinerungs-Algorithmus arbeitet auf Koalgebren, daher wird zusätzlich der Zusammenhang von Sprach- und Verhaltensäquivalenz im koalgebraischen Sinn aus [3] ausführlich vorgestellt und wiedergegeben. Im Anschluss daran wird der Algorithmus in zwei Ausführungen vorgestellt.
Der letzte Abschnitt von Kapitel 2 gibt die Ergebnisse der Recherche zu Semiringen und Lösungsverfahren für LGS wieder. In diesem Abschnitt werden die mathematischen Grundlagen zu bestimmten Semiringen aufgeführt und unter dem Aspekt der automatischen Generierung untersucht.
Das dritte Kapitel beschreibt den Prozess der Implementierung. Zunächst wird eine Einführung in die generative Programmierung gegeben und das verwendete CodeDOM .net-Framework vorgestellt. Anschließend wird ein Überblick über das bereits

am Anfang dieser Arbeit bestehende und finale Programm gegeben [3]. In den einzelnen Abschnitten bezüglich der Implementierung wird zunächst das Programm-Design erläutert und allgemeine Probleme und Konzepte werden vorgestellt. Im Anschluss daran werden Details zu den Umsetzungen der Semiringe aus dem Grundlagenkapitel aufgeführt. Zusätzlich wird die Anwendung des Programms an drei verschiedenen möglichen Fällen demonstriert.
Eine Evaluation des Partitionsverfeinerungs-Algorithmus ist im vierten Kapitel tabellarisch aufgeführt. Im Anschluss daran werden eine Zusammenfassung und ein Ausblick in Bezug auf die finalen Zuständen der Programme gegeben. Den Schluss dieser Arbeit bildet ein Fazit zu den verwendeten und eigenständig implementierten Lösungsverfahren für lineare Gleichungssysteme und den Semiring-Generator.

2 Theoretische Grundlagen

Grundlage dieser Arbeit ist ein generischer Partitionsverfeinerungs-Algorithmus zur Untersuchung von Verhaltensäquivalenzen in Transitionssystemen [3]. Am Beispiel der gewichteten Automaten wird eine konkrete Umsetzung des Algorithmus demonstriert. Auftauchende Probleme bilden den Rahmen der folgenden Kapitel.
Zunächst wird auf die mathematischen Grundlagen von zwei Definitionen gewichteter Automaten eingegangen. Im Anschluss daran wird der Partitionsverfeinerungs-Algorithmus vorgestellt, der im Falle der gewichteten Automaten auf der Lösung von linearen Gleichungssysteme beruht. Zum Schluss werden Eigenschaften und Lösungsverfahren für lineare Gleichungssysteme verschiedener mathematischer Strukturen vorgestellt.

2.1 Grundlagen gewichteter Automaten

In diesem Unterkapitel werden zwei Definitionen für gewichtete Automaten vorgestellt. Im Zusammenhang mit den Definitionen werden u.a. Begriffe aus der Kategorientheorie aufgeführt.

2.1.1 Gewichtete Automaten

Die Informatik bedient sich der Automaten sowohl in der Theorie als auch in der Praxis. Zum einen finden diese Modelle Verwendung in der Beschreibung von Sprachen [6], zum anderen werden sie z.B. im Compilerbau [7] oder bei Schaltkreiskonstruktionen [8] eingesetzt.
An dieser Stelle werden zunächst einige grundlegende Definitionen in Bezug zu Automaten gegeben. Klassische endliche Automaten und gewichtete endliche Automaten definieren Sprachen [9]. Um auf die Unterschiede zwischen den Sprachen von klassischen und gewichteten Automaten eingehen zu können, erfolgt zunächst die Definition einer formalen Sprache.

Definition 2.1.1 (Formale Sprache [6])**.** *Eine formale Sprache über einem Alphabet Σ, d.h. einer endlichen Menge, ist eine beliebige Teilmenge von Σ^*, d.h. der Menge aller Sequenzen von Elementen aus Σ.*

Klassische Automaten beschränken sich auf das Akzeptieren oder Verwerfen eines Wortes und definieren so die Sprache des Automaten. Gewichtete Automaten weisen jedem Wort $w \in \Sigma^*$ ein bestimmtes Gewicht zu. Das Gewicht wird dabei einem Semiring entnommen. Der Zusammenhang von Semiringen, gewichteten Automaten und Sprachäquivalenz ist Untersuchungsgegenstand dieser Arbeit.

Definition 2.1.2 (Halbgruppe [10]). *Eine Halbgruppe $(M, *)$ ist eine Menge M mit einer inneren assoziativen zweistelligen Verknüpfung $*\colon M \times M \to M, (a, b) \to a*b$. Es gilt $\forall a, b, c \in M\colon a * (b * c) = (a * b) * c$.*

Definition 2.1.3 (Semiring (vgl. [3])). *Ein Semiring ist eine mathematische Struktur mit einer nicht leeren Menge K und zwei zweistelligen Verknüpfungen $+, \cdot : K \times K \to K$ für die gilt:*
1. $(K, +)$ ist eine kommutative Halbgruppe mit neutralem Element $0 \in K$.
Eine Halbgruppe heißt kommutativ, wenn $a * b = b * a\ \forall a, b \in M$ gilt.
2. $(K, \cdot)$ ist eine Halbgruppe mit neutralem Element $1 \in K$.
3. Für 1 und 0 gilt $1 \neq 0$. Zusätzlich muss $\forall a \in K\ 0 \cdot a = 0 = a \cdot 0$ gelten.
4. Multiplikation ist distributiv über der Addition, d.h. $\forall a, b, c \in K$ gilt $(a + b) \cdot c = a \cdot c + b \cdot c$ und $c \cdot (a + b) = c \cdot a + c \cdot b$.

Ein Semiring ist ein Ring, ohne dass dabei das additive Inverse vorausgesetzt wird.

Definition 2.1.4 (Gewichteter Automat (vgl.[11])). *Sei S $(K, +, \cdot, 0, 1)$ ein Semiring, Z eine nicht leere Menge und Σ ein Alphabet. Ein Viertupel $\mathcal{A} = (Z, \Sigma, \delta, \theta)$ ist ein Automat mit einer Menge von Zuständen Z, Transitionen $\delta\colon Z \times \Sigma \times Z \to S$, und Ausstiegsgewichten $\theta\colon Z \to S$.*

Die Sprache, welche abhängig vom Semiring ist, beschreibt das Verhalten des Automaten [4]. Sind zwei Sprachen also gleich, impliziert dies, dass die Automaten verhaltensäquivalent sind.
Der Unterschied zwischen der Sprache eines klassischen Automaten und einer gewichteten Sprache liegt im Wertebereich des Semiringes für klassische Automaten. Ein solcher Automat kann als gewichteter Automat über dem booleschen Semiring $\mathcal{B} = (\{0, 1\}, \vee, \wedge, 0, 1)$ aufgefasst werden. Das Verhalten des klassischen Automaten ist eine Funktion $\|\mathcal{A}\|\colon Z \times \Sigma^* \to \mathcal{B}$.
Während bei einem klassischen endlichen Automaten jedem Wort entweder 1 oder 0 zugewiesen wird, assoziiert ein gewichteter Automat jedes Wort w mit einem Wert $x \in S$, indem alle Werte der Pfade mit dem Wort w summiert werden. Ein Pfad ist eine alternierende Sequenz $P = z_0 a_1 ... a_n z_n \in Z(\Sigma Z)^*$ mit dem Wort $w = a_1 ... a_n \in \Sigma^*$. Das Gewicht eines Pfades berechnet sich gemäß:

$$Gewicht(P) = (\Pi_{0 \leq i < n} \delta(z_i, a_{i+1}, z_{i+1})) \cdot \theta(n) \tag{1}$$

Die Multiplikation der einzelnen Gewichte bestimmt den Wert eines Pfades, dabei werden auch die Ausstiegsgewichte einbezogen. In den meisten Definitionen finden sich sogenannte Einstiegsgewichte, die bei der Berechnung mit dem Gewicht des Pfades multipliziert werden. In dieser Arbeit entfallen die Einstiegsgewichte, da dies

unüblich bei koalgebraischer Betrachtung ist. Bei Bedarf ließe sich ein Automat nach obiger Definition durch einen Initialgewichtsvektor um die Einstiegsgewichte erweitern.
Das Verhalten eines gewichteten Automaten wird durch die Addition und Multiplikation beeinflusst. Die Wörter $w \in \Sigma^*$ erhalten somit Koeffizienten und deren Summation ergibt das Verhalten des Automaten. Das Verhalten eines gewichteten Automaten $\mathcal{A}$ entspricht daher einer Funktion $\| \mathcal{A} \|: Z \times \Sigma^* \to \mathcal{S}$. Gewichtete Automaten erlauben es, im Gegensatz zu den klassischen weitere Informationen zu berechnen, z.B. maximale Kosten für eine bestimmte Folge von Aktionen.
Das Verhalten von zwei gewichteten Automaten kann unter bestimmten Voraussetzungen [4] auf Gleichheit überprüft werden. Wesentlich für das Thema Verhaltensäquivalenz und Sprachäquivalenz ist das Konzept der Minimierung. Zwei klassische Automaten, deren minimale Repräsentationen identisch sind, akzeptieren auch die gleiche Sprache. Die Existenz der eindeutigen minimalen Darstellung wie im Falle eines ungewichteten deterministischen endlichen Automaten [6] kann nicht bei gewichteten Automaten vorausgesetzt werden (vgl. *Handbook of Weighted Automata* und [4]). Mohris Algorithmus zur Minimierung von deterministischen gewichteten Automaten normiert die Verteilung der Gewichte (weight pushing) und vereint Gewichte und Transitions-Beschriftungen zu einer neuen Menge von ungewichteten Beschriftungen. Dieser Algorithmus ist auf Semiringe ohne Nullsumme anwendbar. Dabei können mehrere minimale Automaten erzeugt werden, welche sich in der Verteilung der Gewichte unterscheiden, jedoch die selbe Topologie haben (vgl. *Handbook of Weighted Automata*).
Ausgehend von Schützenbergers Arbeit ist die Entscheidbarkeit der Sprachäquivalenz für bestimmte Ringe bewiesen [4]. Neben anderen Grundideen für Algorithmen [3] zur Überprüfung von Sprachäquivalenz, z.B. der Konjugation [12], findet sich in zahlreichen Veröffentlichungen das Konzept der Koalgebra als Framework [13],[3].

2.1.2 Koalgebraische Sichtweise

Anhand der Definition von Verhaltensäquivalenz im koalgebraischen Sinn erhalten wir einen Algorithmus zur Überprüfung von Verhaltensäquivalenz, der unabhängig von der Art des Transitionssystems ist [3]. Diese Arbeit beschäftigt sich mit gewichteten Automaten, deren Verhalten eine gewichtete Sprache ist. An dieser Stelle werden zunächst einige grundlegende Definitionen und Zusammenhänge bezüglich der Definition von Koalgebra und Verhaltensäquivalenz aufgeführt.

Definition 2.1.5 (Kategorie [14])**.** *Eine Kategorie* $\boldsymbol{C}$ *ist ein Tupel aus einer Sammlung von Objekten* $\boldsymbol{O}$*, einer Sammlung von Pfeilen* $\boldsymbol{P}$ *(oder auch Morphismen) und einem Kompositionsoperator* $\circ$ *für die gilt:*

Jedem Pfeil f ist ein Definitionsbereich $dom(f) \in \boldsymbol{O}$ und ein Bildbereich $cod(f) \in \boldsymbol{O}$ zugeteilt. Für einen Pfeil f mit $dom(f) = A$ und $cod(f) = B$ schreiben wir auch $f\colon A \to B$. Der Kompositionsoperator $\circ$ verknüpft zwei Pfeile f und g genau dann als $g \circ f$, wenn $cod(f) = dom(g)$. Für den Kompositionsoperator gilt das Assoziativgesetz, das heißt für drei Pfeile $f\colon A \to B$, $g\colon B \to C$, $h\colon C \to D$ gilt:

$$h \circ (g \circ f) = (h \circ g) \circ f. \tag{2}$$

Zudem gibt es für jedes Objekt $A \in \boldsymbol{O}$ einen Identitätspfeil $id_A\colon A \to A$. Die Identitätspfeile genügen dem Identitätsgesetz, d.h., es gilt für jeden Pfeil $f\colon A \to B$:

$$id_B \circ f = f \tag{3}$$

und

$$f \circ id_A = f. \tag{4}$$

Ein Beispiel für den Begriff der Kategorie ist die Kategorie der Mengen, auch **Set** genannt. Mengen sind dabei die Objekte der Kategorie. Die Morphismen in **Set** entsprechen Abbildungen zwischen den Mengen. Die Identität ist in diesem konkreten Fall die Identitätsabbildung.
Abbildungen zwischen Kategorien werden als Funktoren bezeichnet. Die stukturerhaltende Eigenschaft von Funktoren macht einen Bestandteil der Partitionsverfeinerungs-Algorithmen A und B aus.

Definition 2.1.6 (Funktor [14])**.** *Gegeben seien zwei Kategorien $\boldsymbol{C}$ und $\boldsymbol{D}$. Ein Funktor $F\colon \boldsymbol{C} \to \boldsymbol{D}$ beschreibt eine Abbildung zwischen den beiden Kategorien $\boldsymbol{C}$ und $\boldsymbol{D}$. Jedes $\boldsymbol{C}$-Objekt A wird auf ein $\boldsymbol{D}$-Objekt $F(A)$ abgebildet und jeder $\boldsymbol{C}$-Pfeil $f\colon A \to B$ wird auf einen $\boldsymbol{D}$-Pfeil $F(f)\colon F(A) \to F(B)$ abgebildet, sodass für alle $\boldsymbol{C}$-Objekte und konkatenierbare $\boldsymbol{C}$-Pfeile f und g gilt: $F(id_A) = id_{F(A)}$ und $F(g \circ f) = F(g) \circ F(f)$.*

Ein Funktor, der innerhalb der gleichen Kategorie abbildet, wird als Endofunktor bezeichnet. Ein Funktor F ist injektiv (treu), wenn für zwei Pfeile $f, g\colon A \to B$, $F(f) = F(g)$ immer $f = g$ impliziert.

Definition 2.1.7 (Konkrete Kategorie [15])**.** *Eine konkrete Kategorie ist ein Paar $(\boldsymbol{C}, U)$, wobei $\boldsymbol{C}$ eine Kategorie und $U\colon \boldsymbol{C} \to \boldsymbol{Set}$ ein injektiver Funktor ist, der als Konkretisierungs-Funktor bezeichnet wird.*

Der Partitionsverfeinerungs-Algorithmus erwartet als Eingabe einen gewichteten Automaten, der eine Koalgebra ist.

Definition 2.1.8 (Koalgebra [3]). *Sei $F\colon \boldsymbol{C} \to \boldsymbol{C}$ ein Endofunktor auf der Kategorie $\boldsymbol{C}$. Eine F-Koalgebra ist ein Paar $(X, \alpha : X \longrightarrow FX)$, wobei X ein Objekt von $\boldsymbol{C}$ ist und α ein Pfeil in $\boldsymbol{C}$. Sind zwei F-Koalgebren $(X, \alpha), (Y, \beta)$ gegeben, ist ein Homomorphismus im koalgebraischen Sinne ein Morphismus $f\colon X \to Y$, sodass $Ff \circ \alpha = \beta \circ f$ gilt.*

Beispiel 1. *Eine Veranschaulichung für den Begriff der Koalgebra lässt sich am Beispiel des deterministischen Automaten erklären. Ein deterministischer Automat mit einer Menge X von Zuständen und einer Menge Σ von Eingabezeichen lässt sich durch eine Funktion $a : X \to X^{\Sigma} \times \{0, 1\}$ darstellen. Dabei definiert $a_1\colon X \times \Sigma \to X$ die Übergangsfunktion in den nächsten Zustand und $a_2\colon X \to \{0, 1\}$ beschreibt die Akzeptanz. Dies kann als Koalgebra für den Funktor: $FX \to X^{\Sigma} \times \{0, 1\}$ aufgefasst werden [16].*

Das Beispiel zeigt, dass klassische Automaten anhand von Koalgebren dargestellt werden können. Für die Definition von gewichteten Automaten im koalgebraischen Sinn fehlt an dieser Stelle noch die Definition der Kategorie der Matrizen.

Definition 2.1.9 (Kategorie der Matrizen). *Die Kategorie $M(\mathcal{S})$ der Matrizen über Semiringen enthält als Objekte Mengen und die Morphismen sind Matrizen $a : Y \to X$. Diese Matrizen entsprechen Funktionen der Form $X \times Y \to \mathcal{S}$. Die Verknüpfung wird mittels Matrizenmultiplikation durchgeführt und die Identität ist die Einheitsmatrix.*

Definition 2.1.10 (Gewichteter Automat [3]). *A sei eine endliche Menge von Aktionen. Es wird ein Endofunktor $F\colon M(\mathcal{S}) \to M(\mathcal{S})$ folgendermaßen definiert: $FX = A \times X + 1$ für eine Menge X. Für einen Morphismus $f : Y \to X$ ergibt sich für $Ff\colon A \times Y + 1 \to A \times X + 1$ und dies ist eine Funktion der Form $Ff\colon (A \times X + 1) \times (A \times Y + 1) \to \mathcal{S}$.*

$$Ff((a, x), (a, y)) = f(x, y) \text{ für } a \in A, x \in X, y \in Y$$

$$Ff(\bullet, \bullet) = 1$$

$$Ff(c, d) = 0 \text{ für alle weiteren } c \in A \times X + 1, d \in A \times Y + 1$$

Somit ist ein gewichteter Automat eine F-Koalgebra, ein Morphismus: $\alpha : X \to FX$ in der Kategorie $M(\mathcal{S})$. Dies entspricht einer Matrix der Form: $(A \times X + 1) \times X$ mit Einträgen aus $\mathcal{S}$. In Bezug auf die Definition 2.1.4 ist A das Eingabealphabet Σ und X die Zustandsmenge. Der Morphismus α beschreibt analog zu $\delta\colon Z \times \Sigma \times Z \to S$ die Transitionen zwischen den Zuständen und deren Gewichte.
Die folgenden Algorithmen arbeiten in der Kategorie $M(\mathcal{S})$, damit Verhaltensäquivalenz Sprachäquivalenz ist. Wie bereits erwähnt, ist eine gewichtete Sprache

eine Funktion der Form $\Sigma^* \to \mathcal{S}$, wobei $\mathcal{S}$ ein Semiring ist. Die Funktion ist dabei gleichzeitig das Verhalten des Automaten. Um den Bezug zwischen den beiden Äquivalenzen im koalgebraischen Sinn zu verdeutlichen, werden die Definition von gewichteten Sprachen und der Begriff der Sprachäquivalenz an dieser Stelle in formaler Schreibweise aufgeführt.

Definition 2.1.11 (Sprache und Sprachäquivalenz [3])**.** *Sei (X, α) ein gewichteter Automat über einem Alphabet Σ, einem Semiring $\mathcal{S}$ und einer endlichen Menge X. Die Sprache L_α ist folgendermaßen rekursiv definiert:*

$L_\alpha(\varepsilon)(x) = \alpha(\bullet, x)$

$L_\alpha(aw)(x) = \sum_{x' \in X} \alpha((a, x'), x) \cdot L_\alpha(w)(x')$ *mit* $a \in \Sigma$ *und* $w \in \Sigma^*$

Zwei Zustände x,y sind sprachäquivalent, wenn $\forall w \in \Sigma^*$ *gilt:* $L_\alpha(w)(x) = L_\alpha(w)(y)$.

Wir wollen nun koalgebraische Verhaltensäquivalenz definieren und dann sehen, dass im Falle von gewichteten Automaten Sprach- und Verhaltensäquivalenz zusammenfallen [3].

Definition 2.1.12 (Verhaltensäquivalenz [3])**.** *Sei $(\boldsymbol{C}, U)$ eine konkrete Kategorie und $(X, \alpha : X \longrightarrow FX)$ eine F-Koalgebra. Die Elemente $\bar{x} \in UX$ bezeichnen die Zustände von (X, α). Zwei Zustände $\bar{x}, \bar{y} \in UX$ sind verhaltensäquivalent ($\bar{x} \sim \bar{y}$), wenn eine F-Koalgebra (Y, β) existiert und dazu ein Koalgebra-Homomorphismus f: $(X, \alpha) \to (Y, \beta)$, sodass $Uf(\bar{x}) = Uf(\bar{y})$. Dies bedeutet für einen Morphismus f, dass $Uf(\bar{x}) = f \cdot \bar{x} = f \cdot \bar{y} = Uf(\bar{y})$.*

Bildet ein Koalgebra-Homomorphismus f zwei Zustände x, y der Koalgebra (X, α) auf einen Zustand in UY ab, dann impliziert dies Verhaltensäquivalenz. Sind zwei Zustände eines Automaten (X, α) verhaltensäquivalent, muss es einen Koalgebra-Homomorphismus $f\colon X \to Y$ und einen weiteren Automaten (Y, β) geben, der diese Zustände schon vereinigt hat, d.h., Uf(x) = Uf(y). Für f bedeutet dies, dass die Zeilen von x und y übereinstimmen. Die Berechnung von f und β, falls diese existieren, ermöglicht somit Aussagen über die Verhaltensäquivalenz von zwei Zuständen einer Koalgebra (X, α). Die Grundlage zur Berechnung eines Homomorphismus f anhand der Algorithmen A und B ist in Anbetracht der Tatsache, dass die zugrunde liegende Kategorie Pfeile in Matrixform beinhaltet, die Matrizenmultiplikation. Bevor der Zusammenhang zwischen Sprach- und Verhaltensäquivalenz in koalgebraischer Sichtweise erläutert wird, wird zunächst die Bedeutung der Verhaltensäquivalenz für zwei Zustände an einem Beispiel gezeigt.
Der Semiring (Abb. 1) ist in diesem Fall ein endlicher distributiver Verband (vgl.

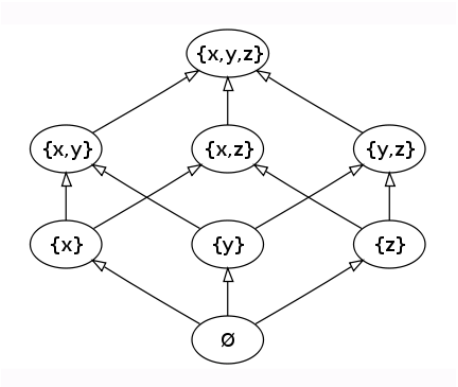


Abbildung 1: Distributiver endlicher Verband Quelle: [17]

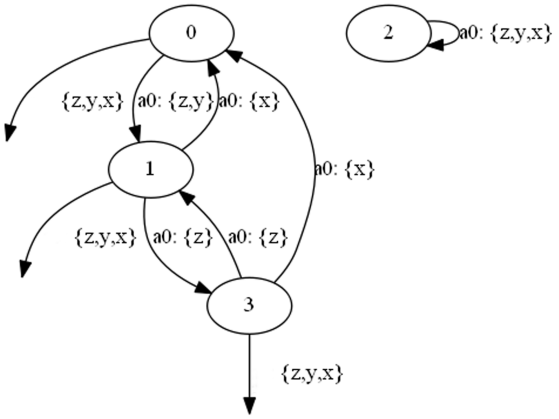


Abbildung 2: Beispiel für einen gewichteten Automaten über dem Verband aus Abb. 1.

$$
\begin{array}{lc}
\alpha: & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\
\begin{array}{l} (a_0,1) \\ (a_0,2) \\ (a_0,3) \\ (a_0,4) \\ \bullet \end{array} & \left(\begin{array}{cccc} \emptyset & \{x\} & \emptyset & \{x\} \\ \{z,y\} & \emptyset & \emptyset & \{z\} \\ \emptyset & \emptyset & \{z,x,y\} & \emptyset \\ \emptyset & \{z\} & \emptyset & \emptyset \\ \{z,x,y\} & \{z,x,y\} & \emptyset & \{z,x,y\} \end{array} \right)
\end{array}
$$

Abbildung 3: Koalgebra $(X, \alpha : X \longrightarrow FX)$ zu Abb. 2.

Definition 2.3.8). In Abb. 2 ist ein gewichteter Automat über dem Alphabet $\{a_0\}$ abgebildet. Die vier Zustände werden mittels Partitionsverfeinerung auf Sprachäquivalenz untersucht. Die Eingabe ist die Koalgebra aus Abb. 3. Das Ergebnis ist eine Koalgebra (Y, β) und ein Homomorphismus $f\colon (X, \alpha) \to (Y, \beta)$ (Abb.4) mit den Einschränkungen für zwei Zustände $x, y \in X$ aus Definition 2.1.12. Der Homomorphismus in Abb. 4 stimmt in zwei Spalten überein, denn die Einträge der zweiten Spalte sind gleich den Einträgen der vierten Spalte. Sind zwei Spalten von f gleich, sind die entsprechenden Zustände verhaltensäquivalent. Die Einheitsvektoren $\hat{2}, \hat{4} \in UX$ ergeben daher mit f multipliziert $f \cdot \hat{2} = f \cdot \hat{4}$. Die Einträge sind für den Einheitsvektor $\hat{x}$ für $x \in X$ definiert als $\hat{x}(x) = 1$, ansonsten null.

$$\begin{pmatrix} \{z,y\} & \{z,x\} & \emptyset & \{z,x\} \\ \{z,y,x\} & \{z,y,x\} & \emptyset & \{z,y,x\} \end{pmatrix}$$

Abbildung 4: Berechneter Homomorphismus f.

In Abb. 2 ist zu sehen, dass die Pfade von Zustand 2 aus über die gleichen Zustände entlanglaufen wie von Zustand 4. Dies geschieht zusätzlich mit denselben Gewichten für die übereinstimmenden Übergänge. Es ist kein Zufall, dass die beiden Zustände 2 und 4 ebenfalls sprachäquivalent sind. Die beiden Äquivalenzen stehen auf der Basis von $M(\mathcal{S})$ und der koalgebraischen Definition von gewichteten Automaten in folgendem Zusammenhang:

Definition 2.1.13 (Zusammenhang Sprach- und Verhaltensäquivalenz [15])**.** *Sei $(X, \alpha : X \longrightarrow FX)$ ein gewichteter Automat mit einer endlichen Menge X, einem Alphabet Σ und einem Semiring $\mathcal{S}$. Zwei Zustände x,y sind genau dann sprachäquivalent, wenn die Einheitsvektoren $\hat{x}, \hat{y}$ verhaltensäquivalent sind ($\hat{x} \sim \hat{y}$).*

Der Beweis wird hier für beide Richtungen wiedergegeben und basiert u.a. auf folgenden Lemmata und Definitionen [15].

Definition 2.1.14 (Semimodul, Matrizen [3])**.** *Es sei X eine Indexmenge und $\mathcal{S}$ ein Semiring. Bei der Menge $\mathcal{S}^X$ aller Funktionen $s : X \to \mathcal{S}$ handelt es sich um ein Semimodul, eine unter Addition und Multiplikation abgeschlossene algebraische Struktur. Jede Untermenge einer solchen Menge wird als Subsemimodul bezeichnet, wenn diese wiederum unter Addition und Multiplikation abgeschlossen ist.*
Für ein Erzeugendensystem $G \subseteq \mathcal{S}^X$ wird das Subsemimodul, das von G aufgespannt wird, mit $\langle G \rangle$ notiert.
Ebenso entspricht eine $X \times Y$ Matrix einem Subsemimodul, das durch die Zeilen der Matrix aufgespannt wird.

Angewandt auf gewichtete Automaten (X, α) nennen wir das Erzeugendensystem der Menge $\{L_\alpha(w) \mid w \in A^*, |w| \leq n\}$, die die Gewichte der Wörter der Länge kleiner oder gleich n enthält, G^n. Es besteht ein Zusammenhang zwischen der Anwendung des Funktors und der korrespondierenden Wortmenge mit den Worten der Länge n über einem Alphabet.

Definition 2.1.15 (Terminales Objekt und Final Chain [18]). *Es sei $\boldsymbol{C}$ eine Kategorie. Ein Objekt $T \in \boldsymbol{C}$ ist terminal, wenn für alle Objekte $X \in \boldsymbol{C}$ gilt, dass ein eindeutiger Morphismus $T_X \colon X \to T$ existiert.*
Es sei ein Endofunktor $F : \boldsymbol{C} \to \boldsymbol{C}$ gegeben. Eine Konstruktion der Sequenz $F^i 1$ erhält man durch die iterative Anwendung von F, beginnend mit dem eindeutigen Morphismus. Die Sequenz $1 \longleftarrow F^1 1 \longleftarrow F^2(1)...$ wird als Final Chain bezeichnet.

Da ein gewichteter Automat eine Koalgebra $\alpha \colon X \longrightarrow FX$ ist, kann anhand der iterativen Anwendung eines Funktors, beginnend mit dem eindeutigen Morphismus von $(X \times \emptyset)$ im Fall der Kategorie $M(\mathcal{S})$, eine Final Chain konstruiert werden. Durch die zusätzliche Verknüpfung mit α bei der iterativen Anwendung erhalten wir eine Sequenz von Pfeilen d_i.

Lemma 1 (Funktoren und Subsemimodule [15]). *Es sei $d_i : X \to F^i 1$ eine Sequenz von mittels des Funktors F erzeugten Pfeilen. Dabei entspricht 1 dem terminalen Objekt. $F^i 1$ enthält die Wörter über A der Länge $|w| < i$. Somit ist $L_\alpha(w) = (d_i)_w$. Das bedeutet, dass $\langle d_{i+1} \rangle = \langle G^i \rangle$ ist.*

Mittels der iterativen Anwendung des Funktors wird ein Pfeil gewonnen, der Aussagen über die Sprachäquivalenzen der einzelnen Zustände beinhaltet, falls dieser existiert. Bevor der Algorithmus vollständig vorgestellt wird, wird zunächst gezeigt, dass Verhaltens- und Sprachäquivalenz zusammenfallen. Für diesen Beweis benötigen wir jedoch noch die Definition des Post-Fixpunktes eines Funktors und den Begriff der Ordnung von Pfeilen.

Definition 2.1.16 (Relationen von Objekten und Pfeilen [15]). *Es seien X, Y zwei Objekte einer Kategorie $\boldsymbol{C}$. Wir notieren $X \leq Y$, wenn es einen Pfeil $f \colon X \to Y$ gibt. Weiterhin notieren wir $X \equiv Y$, falls $X \leq Y$ und $Y \leq X$ gelten.*
Es seien $a \colon X \to A$ und $b \colon X \to B$ zwei Pfeile mit der gleichen Domain. Wir notieren $a \leq^X b$, falls ein weiterer Pfeil mit $d : A \to B$ existiert, sodass $d \circ a = b$ gilt. Analog zu den Objekten notieren wir $a \equiv b$, falls $a \leq^X b$ und $b \leq^X a$ gelten.

In Bezug auf die Ordnung von Pfeilen gilt für Pfeile f, deren $cod(f)$ das terminale Objekt ist, dass jeder beliebige Pfeil g mit $dom(g) = dom(f) = X$ der selben Kategorie $g \leq^X f$ erfüllt.

Lemma 2 (Verhaltensäquivalenz und Post-Fixpunkt)**.** *Sei F ein Endofunktor auf einer konkreten Kategorie ($\boldsymbol{C}$,U) und sei $\alpha : X \to FX$ eine Koalgebra in $\boldsymbol{C}$. Weiterhin sei $f\colon X \to Y$ ein Morphismus. Es ist genau dann $f \leq^X Ff \circ \alpha$, wenn es eine Koalgebra $\beta\colon Y \to FY$ gibt, so dass also $\beta \circ f = Ff \circ \alpha$ gilt.*
Für jeden solchen Post-Fixpunkt f mit $\bar{x}, \bar{y} \in UX$ impliziert $Uf(\bar{x}) = Uf(\bar{y})$ auch $\bar{x} \sim \bar{y}$. Zusätzlich ist f unter der Voraussetzung, dass für jeden weiteren Post-Fixpunkt $g\colon X \to Y$ ebenfalls $g \leq^X f$ gilt, der größte Post-Fixpunkt. Damit induziert die Koalgebra f Verhaltensäquivalenzen: $Uf(\bar{x}) = Uf(\bar{y}) \Longleftrightarrow \bar{x} \sim \bar{y}$

Beweis 1. *1. Angenommen die Einheitsvektoren $\hat{x}, \hat{y}$ verhalten sich äquivalent ($\hat{x} \sim \hat{y}$), jedoch sind die Zustände x,y nicht sprachäquivalent. Aufgrund der Definition für Verhaltensäquivalenz auf Basis der Koalgebra wissen wir, dass es einen Homomorphismus $f\colon (X, \alpha) \to (Y, \beta)$ gibt, sodass $Uf(\hat{x}) = Uf(\hat{y})$. Somit sind die Einträge der korrespondierenden Zeilen x, y in f gleich. Jedoch existiert auch ein Wort w, sodass $L_\alpha(w)(x) \neq L_\alpha(w)(y)$. Somit muss aufgrund Lemma 1 ein Pfeil $d_{|w|+1}$ existieren, dessen entsprechende Spalten für x,y nicht gleich sind. Aufgrund von Lemma 2 ergibt dies jedoch einen Widerspruch, da $d_{|w|+1} \geq f$ und somit $Ud_{|w|+1}(\hat{x}) \neq Ud_{|w|+1}(\hat{y})$ gilt und gleichzeitig $Uf(\hat{x}) = Uf(\hat{y})$. Da $d_{|w|+1} \geq f$ gilt, impliziert dies, dass die Zeilen in $d_{|w|+1}$ als Linearkombinationen von den Zeilen in f gebildet werden können.[15]*

2. Angenommen x und y sind sprachäquivalent, so muss es einen Koalgebra-Homomorphismus $f\colon (X, \alpha) \to (Y, \beta)$ mit $Uf(\hat{x}) = Uf(\hat{y})$ geben. Um dies zu zeigen, wählen wir für $Y = A^*$ und β ist eine $(A \times A^* + 1) \times A^*$ Matrix. Die Einträge sind folgendermaßen definiert: $\beta((a, w), aw) = 1, \beta(\bullet, \varepsilon) = 1$, sonst 0. Es handelt sich um einen Pfeil der Kategorie $M(\mathcal{S})$. Nun wählen wir für $f : X \to A^*$ eine Matrix mit den Einträgen: $f(w, z) = L_\alpha(w)(z)$, wobei $w \in A^*$ und $z \in X$. Aufgrund der Tatsache, dass X eine endliche Menge ist, handelt es sich bei f um einen wohldefinierten Pfeil. Wegen der Sprachäquivalenz von x und y sind die Spalten von f für x,y gleich.

Nun verbleibt es zu zeigen, dass es sich bei f um einen Koalgebra-Homomorphismus handelt ($Ff \circ \alpha = \beta \circ f$):
$(Ff \circ \alpha)((a, w), z)$
$= \sum_{(a', z') \in A \times X} Ff((a, w), (a', z')) \cdot \alpha((a', z'), z) + Ff((a, w), \bullet) \cdot \alpha(\bullet, z)$
$= \sum_{z' \in X} f(w, z') \cdot \alpha((a', z'), z) = L_\alpha(aw)(z) = f(aw, z)$

Und weiter für $\beta \circ f((a, w), z) = \sum_{w' \in A*} \beta((a, w), w') \cdot f(w', z) = \sum_{w' = aw} \beta((a, w), w') \cdot f(w', z) = \beta((a, w), aw) \cdot f(aw, z) = 1 \cdot f(aw, z)$
Weiterhin gilt für ein beliebiges $z \in X$:

$(Ff \circ \alpha)(\bullet, z)$
$= \sum_{(a',z') \in A \times X} Ff(\bullet, (a', z')) \cdot \alpha((a', z'), z) + Ff(\bullet, \bullet) \cdot \alpha(\bullet, z)$
$= Ff(\bullet, \bullet) \cdot \alpha(\bullet, z) = \alpha(\bullet, z) = L_\alpha(\varepsilon)(z)$
und analog für $(\beta \bullet z) = \sum_{w' \in A*} \beta(\bullet, w') \cdot f(w', z)$
$= \sum_{w'=\varepsilon} \beta(\bullet, w') \cdot f(w', z) = \beta(\bullet, \varepsilon) \cdot f(\varepsilon, z) = L_\alpha(\varepsilon)(z)$*[15]*

Wie bereits erwähnt, basieren die Partitionsverfeinerungs-Algorithmen A und B auf der Konstruktion einer Sequenz $F^i 1$[18]. Aussagen über die Verhaltensäquivalenz von zwei Zuständen können dem größten Post-Fixpunkt entnommen werden. Die Umsetzung der Partitionsverfeinerung A [3] ist die Berechnung des größten Post-Fixpunktes.

2.2 Partitionsverfeinerungs-Algorithmus

Die Konstruktion der Final Chain [18] bedarf der Existenz eines terminalen Objektes. Für die Kategorie $M(\mathcal{S})$ ist dies die leere Menge. Veranschaulicht für Matrizen ist für jede beliebige Menge X der eindeutige Pfeil in die leere Menge eine $\emptyset \times X$-Matrix.

2.2.1 Partitionsverfeinerung A

Der Algorithmus berechnet für die Ausgabe den größten Post-Fixpunkt und eine Koalgebra, wenn er terminiert. Dies ist nicht zwangsläufig gegeben, was auf die Unentscheidbarkeit von Sprachäquivalenzen auf bestimmten Semiringen zurückzuführen ist [19]. Als Eingabe dient eine Koalgebra $\alpha : X \to FX$. Dabei handelt es sich in dieser Arbeit um gewichtete Automaten, dargestellt in einer Matrix der Form $(A \times X + 1) \times X$ mit Einträgen aus $\mathcal{S}$.

Algorithmus [15] 1 (A)**.** *Sei F ein Endofunktor auf einer konkreten Kategorie (**C**,U) und $\alpha : X \to FX$ eine Koalgebra in **C**.*

> *Schritt 0: Starte mit dem eindeutigen Morphismus $d_0^A : X \to 1$.*
>
> *Schritt $i+1$: Berechne $d_{i+1}^A = Fd_i^A \circ \alpha : X \to F^{i+1}1$*
> *Wenn ein Morphismus $\beta : F^i 1 \to F^{i+1}1$ existiert, für den gilt: $\beta \circ d_i^A = d_{i+1}^A$, also $d_i^A \leq d_{i+1}^A$, dann terminiert der Algorithmus und die Rückgabe setzt sich aus $\beta : F^i 1 \to F(F^i)$ und der Koalgebra $d_i^A : X \to F^i 1$ zusammen.*

Die Grafik verdeutlicht die Folge von $d_0^A \geq d_1^A \geq d_2^A$.... Tritt die Abbruchbedingung $d_i^A \leq d_{i+1}^A$ auf, die durch $\beta \circ d_i^A = d_{i+1}^A$ gegeben ist, beinhaltet d_i^A die Aussagekraft für Verhaltensäquivalenzen von Zuständen. Die Konstruktion der Sequenz der

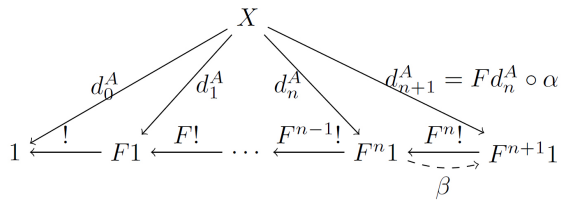


Abbildung 5: Konstruktion auf der terminalen Kette (Quelle: [15])

Morphismen d_i^A entspricht einer Art Partitionierung der Zustände UX in Äquivalenzklassen: $[\bar{x}] := \left\{\bar{y} \in UX \mid Ud_i^A\bar{x} = Ud_i^A\bar{y}\right\}$. Denn tatsächlich ist d_n^A der größte Post-Fixpunkt[3].
Angewandt auf einen gewichteten Automaten α berechnet der Algorithmus die Werte, die ein Zustand x den Worten $|w| \leq i$ zuordnet. Der Algorithmus stoppt, sobald das durch $L_\alpha(w)$ gegebene Semimodul der Wörter der Länge bis zu $i+1$ gleich dem der Wörter der Länge bis zu i ist.

2.2.2 Partitionsverfeinerung B

Eine Optimierung der Partitionsverfeinerung A wird durch die Verwendung von kleineren Zwischenergebnissen erreicht. Algorithmus B arbeitet ebenfalls auf der Final Chain. In jedem Schritt wird der Morphismus durch einen anderen Repräsentanten e_i aus der Äquivalenzklasse ersetzt: $d_i^B = m_i^B \circ e_i^B$
Repräsentanten einer Äquivalenzklasse sind dabei durch alle Pfeile e_i^B mit $d_i^B \equiv e_i^B$ gegeben. Da es sich bei den Pfeilen d_i^B, e_i^B um Pfeile in $M(\mathcal{S})$ handelt, bedeutet dies für $d_i^B \equiv e_i^B \Longrightarrow \langle d_i \rangle = \langle e_i \rangle$. In Bezug auf die Definition 2.1.16 folgt daher $d_i^B \leq e_i^B$ und $e_i^B \leq d_i^B$.

Algorithmus [15] 2 (B)**.** *Sei F ein Endofunktor auf einer konkreten Kategorie $(\boldsymbol{C},U)$ und $\alpha : X \to FX$ eine Koalgebra in $\boldsymbol{C}$. Außerdem sei $\mathcal{R}$, die Klasse der Repräsentanten, eine Klasse von Pfeilen der Kategorie $\boldsymbol{C}$, sodass wir für jeden Pfeil $d \in \boldsymbol{C}$ einen Pfeil $e \in \mathcal{R}$ haben, der äquivalent zu d ist ($d \equiv^X e$).*

> *Schritt 0: Starte mit dem eindeutigen Morphismus $d_0^B \colon X \to 1$.*
>
> *Schritt $i+1$: Berechne einen Repräsentanten $e_i^B \in \mathcal{R}$ für d_i^B mittels Faktorisieren $d_i^B = m_i^B \circ e_i^B$, wobei $\mathcal{R} \ni e_i^B \colon X \to Y_i$ und $m_i^B \colon Y_i \to FY_{i-1}$. Bestimme $d_{i+1}^B = Fe_i^B \circ \alpha \colon X \to FY_i$. Existiert ein Pfeil $\gamma \colon Y_i \to FY_i$, sodass $\gamma \circ e_i^B = d_{i+1}^B$, was $d_{i+1}^B \geq e_i^B$ bedeutet, dann terminiert der Algorithmus B und gibt $\gamma \colon Y_i \to FY_i$ und $e_i^B \colon X \to Y_i \in \mathcal{R}$ zurück.*

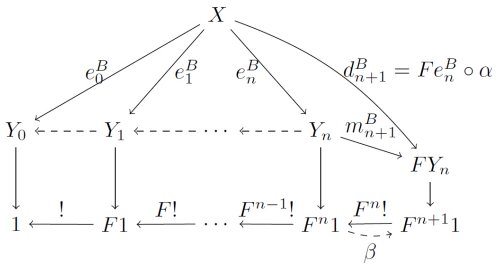


Abbildung 6: Einsparen von Berechnungen mittels äquivalenter Morphismen innerhalb einer Äquivalenzklasse (Quelle: [15]).

Abbildung 6 zeigt den Verlauf auf den Repräsentanten und als Ausgabe berechnet Algorithmus B den Morphismus e_n^B zusammen mit m_{n+1}^B, falls der Algorithmus B terminiert. Die Terminierung von Algorithmus B wird genau dann eintreten, wenn es ein endliches Subsemimodul $\langle G^n \rangle$ gibt, welches $\langle G^* \rangle, G^* = \{L_\alpha | w \in A^*\} = \cup_{n=0}^{\infty} G^n$, gleich ist [15]. Die Existenz eines endlichen Subsemimoduls für die Entscheidbarkeit von Sprachäquivalenz ist bereits für gewichtete Automaten über Ringen als Voraussetzung bekannt [4]. Diese Erkenntnis basiert auf der linearen Darstellung von Funktionen der Form $\Sigma^* \longrightarrow \mathcal{S}$ nach Schützenberger [4]. Die Voraussetzung gilt u.a. für Körper und Schiefkörper, da diese für eine endliche Menge X auch eine endliche Anzahl an Semimodulen beinhalten [4]. In dem Fall handelt es sich bei den Semimodulen um Vektorräume. Ebenfalls erfüllen sämtliche endliche Semiringe diese Eigenschaft und garantieren somit die Terminierung der Algorithmen. Die Algorithmen ermöglichen es für eine Vielzahl von Semiringen Sprachäquivalenz zu entscheiden. In Algorithmus B gelingt anhand der kompakteren Repräsentanten eine Einsparung von Rechenschritten, jedoch stellt eben die Berechnung der Repräsentanten auch das Problem des Verfahrens dar.
Eine Umsetzung des Partitionsverfeinerungs-Algorithmus B ist in Algorithmus 1 gegeben. Zunächst wird die Methode *AlgorithmB* aufgerufen. Die Berechnung startet mit dem eindeutigen Morphismus, der $\emptyset \times X$-Matrix und ruft solange den Funktor

und das Faktorisieren auf, bis $e_{i+1} \geq e_i$ (Zeilen 4-8 Algorithmus 1).

Algorithm 1: AlgorithmB(Matrix α) [15]

```
1 begin
2     e_0 : = the unique (∅ × X)-matrix;
3     i : = 0;
4     repeat
5         i : = i + 1;
6         d_i : Functor(e_{i−1}) · α;
7         (m_i, e_i) : = Factorize(d_i);
8     until e_i ≥ e_{i−1};
9 end
```

Das Faktorisieren von d_i^B wird für gewichtete Automaten mittels des Lösens von linearen Gleichungssystemen umgesetzt. Die Eingabe der Methode *Factorize* ist d_i, eine $n \times |X|$ Matrix. Zunächst wird die erste Zeile von der Matrix abgespalten und anschließend überprüft, ob diese sich als Linearkombination aus den anderen Zeilen darstellen lässt (Zeilen 6-8 Algorithmus 2). Falls dies nicht der Fall ist, wird die

Algorithm 2: Factorize(Matrix d) [15]

```
1  begin
2      e : = d;
3      m : = (n × n) unit matrix;
4      i : = 1;
5      while i ≤ m.rows do
6          e' = concatenate (e[1 . . . i − 1], [i + 1 . . . n]);
7          x : = findlinearCombination((e')^t, (e'[i])^t);
8          if x ≠ undefined then
9              e : = e';
10             m' : = (m.rows − 1 × m.rows) unit matrix;
11             insert x as new row into m' after the (i − 1) th row;
12             m : = m · m';
13         else
14             i : = i + 1;
15         end
16     end
17     return (m, e);
18 end
```

nächste Zeile aus der ursprünglichen Matrix heraus getrennt und überprüft. Andernfalls ist eine Linearkombination möglich und wir erhalten ein Zwischenergebnis, die um die Zeile reduzierte Matrix und für die Koalgebra (Y, β) die Matrix m mit $m \cdot e_i = d_i$ (Zeilen 9-12 Algorithmus 2). Sofern noch nicht alle Zeilen überprüft

worden sind, wird mit dem aktuellen Ergebnis und der korrespondierenden Matrix m fortgefahren.
Die Anzahl der Gleichungssysteme der Form $Ax = b$, die sich für die Berechnung eines Repräsentanten ergibt, hängt dabei von der Anzahl der Zeilen ab. Die Effizienz der einzelnen Verfahren spielt daher besonders für größere Matrizen eine Rolle. Weiterhin stellt sich die Frage, ob überhaupt ein Lösungsverfahren für den Semiring eines beliebigen gewichteten Automaten bekannt ist.
Das aus der Schulzeit bekannte Gauß-Jordan-Verfahren eignet sich für Körper, jedoch beinhaltet nicht jeder Semiring ein multiplikatives oder additives Inverses.

2.3 Semiringe und lineare Gleichungssysteme

Das Lösen von linearen Gleichungssystemen (LGS) über verschiedenen Semiringen benötigt Ansätze, die keine additiven und multiplikativen Inversen voraussetzen. Die Untersuchung von Lösungsverfahren für verschiedene LGS, mit der Absicht, den Partitionsverfeinerungs-Algorithmus B für weitere Semiringe verwenden zu können, ist Inhalt eines Teils dieser Master-Arbeit. Darüber hinaus stellt sich die Frage, wie Semiringe automatisiert generiert werden können, um einem Nutzer die Möglichkeit einzuräumen, einen gewichteten Automaten über einem beliebigen Semiring auf äquivalente Zustände untersuchen zu können. Während der Recherche zu dieser Frage fallen Eigenschaften von mathematischen Strukturen auf, die u.a. eine Konstruktion im programmiertechnischen Sinne von Addition, Multiplikation oder sogar der Division ermöglichen. Sind diese Operationen generierbar, lassen sich LGS der Form $Ax = b$ gegebenenfalls mit bekannten Verfahren lösen, was wiederum das Anwendungsfeld des Partitionsverfeinerungs-Algorithmus B erweitern würde.
Im Vorfeld der Aufführung von mathematischen Grundlagen verschiedener Semiringe wird zunächst auf die unterschiedlichen Methodiken bei Lösungsverfahren für LGS eingegangen. Lösungsverfahren für LGS werden in die Klassen der direkten und der iterativen Methoden unterteilt. Direkte Verfahren bestimmen die Lösung des LGS durch eine vorher abschätzbare Anzahl an binären Operationen. Neben Verfahren wie dem gaußschen Eliminationsverfahren, die auf Äquivalenzumformungen der Matrix A basieren, kann durch die Berechnung der Inversen von A der Lösungsvektor x ebenfalls ermittelt werden.[20]
Iterative Verfahren beginnen die Berechnung von einem Startwert aus und konvergieren im naheliegenden Bereich der Lösung, falls diese existiert. Jedoch garantieren iterative Verfahren nicht für jede Matrix zu konvergieren [20]. Ob ein Verfahren für eine Matrix konvergiert hängt von der Matrix nach der Umformung in eine Fixpunktgleichung ab.

Definition 2.3.1 (Fixpunkt [21]). *Sei $\mathcal{X}$ eine Menge und $f\colon \mathcal{X} \to \mathcal{X}$ eine Funktion. Dann heißt ein Punkt $x \in \mathcal{X}$ Fixpunkt von f, wenn er die Gleichung $f(x) = x$ erfüllt.*

Ein LGS der Form $Ax = b$ kann z.B. für Körper in $x = (I - A)x + b$ umgeformt werden. Die Matrix I ist dabei die Einheitsmatrix.

Definition 2.3.2 (Fixpunktverfahren zum Lösen von LGS [22]). *Es sei $A \in \mathbb{R}^{n\times n}$, sowie $b \in \mathbb{R}^n$ und $A^{-1} \approx C \in \mathbb{R}^{n\times n}$. Wobei C einer numerischen Näherung entspricht. Für einen beliebigen Startwert $x_0 \in \mathbb{R}$ iteriere für k=1,2...*

$$x_k = x_{k-1} + C(b - Ax_{k-1}). \tag{5}$$

Alternativ führen wir die Bezeichnungen $B := I - C \cdot A$ *und* $c := C \cdot b$ *ein. Dann gilt:*

$$x_k = Bx_{k-1} + c. \tag{6}$$

Wird nun die Gleichung aus 2.3.2 für eine Funktion $f(x) = Bx + c$ als Funktionsvorschrift eingesetzt, erhalten wir eine Fixpunktgleichung, deren Fixpunkt eine Lösung für unser LGS $Ax = b$ wäre. Das Verfahren konvergiert für den Fall, dass B die Voraussetzungen des banachschen Fixpunktsatz erfüllt. Für dünn besetzte Matrizen existieren besonders effiziente Verfahren. Da im vorliegenden Fall nicht von dünn besetzten Matrizen ausgegangen werden kann, werden diese Verfahren in dieser Arbeit außer Acht gelassen.
In den Publikationen [23], [20] ist ein Verfahren zur Lösung von Fixpunktgleichungen der Form $x = Ax + b$ aufgeführt, das auf der $*$ -Operation basiert.

2.3.1 *-Semiring $\mathcal{S}$

Ein Verfahren zur Lösung von LGS der Form $x = Ax + b$, wobei A eine $n \times n$ Matrix mit Einträgen $x \in \mathcal{S}$ und $b^n \in \mathcal{S}^n$ ist, kann in bestimmten Semiringen, den *-Semiringen angewandt werden.

Definition 2.3.3 (*-Semiring [9])**.** *Ein *-Semiring ist ein Semiring mit der zusätzlichen unären Operation* $*$*, für die gilt:*

- *Für Semiringe, die unendliche Summen ermöglichen:* $a^* = 1 + a^1 + a^2 ....$
- $a^* = 1 + aa^* = 1 + a^*a$

Für eine Gleichung $x = Ax + b$ kann die Lösung anhand von A^* ermittelt werden. Die Berechnung von A^* ist für Matrizen mit n=1 oder n=2 gegeben durch:

$$|a| = a \text{ und } \begin{vmatrix} a & b \\ c & d \end{vmatrix} = d + ca^*b \tag{7}$$

Für Matrizen mit $n \geq 3$ ergibt sich A^* durch:

$$\begin{vmatrix} a_{1,1} & \dots & a_{1,n} \\ \vdots & \dots & \vdots \\ a_{1,n} & \dots & a_{n,n} \end{vmatrix} = \begin{vmatrix} b_{1,1} & \dots & b_{1,n-1} \\ \vdots & \dots & \vdots \\ b_{1,n} & \dots & b_{n-1,n-1} \end{vmatrix} \text{ und } b_{i,j} = \begin{vmatrix} a_{1,1} & \dots & a_{1,j+1} \\ a_{i+1,1} & \dots & a_{i+1,j+1} \end{vmatrix} \tag{8}$$

Das LGS der Form $x = Ax + b$ wird durch die Berechnung von A^* gelöst, indem für die einzelnen Indizes des Lösungsvektors Matrizen konstruiert werden. Für jede Komponente x_i wird eine Matrix A_i durch Anhängen des Vektors b als Spalte $n+1$ konstruiert. Zusätzlich erhält die Matrix einen Einheitsvektor als Zeile $n+1$, der die Eins an der Position i beinhaltet. Die quadratische Matrix A_i wird anhand der aufgeführten Gleichungen (7) und (8) in einen Wert umgewandelt. Das folgende Beispiel 2 demonstriert die oben beschriebene Berechnung für den Körper der reellen Zahlen. [20],[24]

Beispiel 2. *[20] Ein LGS liegt in der Form $Ax = b$ vor und wird in $x = (I-A)\cdot x+b$ umgeformt.*

$$\begin{Bmatrix} 5 & -3 \\ 3 & 1 \end{Bmatrix} \begin{Bmatrix} x_0 \\ x_1 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 9 \end{Bmatrix} \Leftrightarrow \begin{Bmatrix} x_0 \\ x_1 \end{Bmatrix} = \begin{Bmatrix} -4 & 3 \\ -3 & 0 \end{Bmatrix} + \begin{Bmatrix} 1 \\ 9 \end{Bmatrix}$$

Anschließend wird die Berechnung von A_i^* durchgeführt. An dieser Stelle eine Beispielrechnung für x_0:

$$x_0 = \begin{vmatrix} -4 & 3 & 1 \\ -3 & 0 & 9 \\ 1 & 0 & 0 \end{vmatrix} = \begin{Vmatrix} \begin{vmatrix} -4 & 3 \\ -3 & 0 \end{vmatrix} & \begin{vmatrix} -4 & 1 \\ -3 & 9 \end{vmatrix} \\ \begin{vmatrix} -4 & 3 \\ 1 & 0 \end{vmatrix} & \begin{vmatrix} -4 & 1 \\ 1 & 0 \end{vmatrix} \end{Vmatrix} =$$

$$\begin{vmatrix} 0 + (-3)\cdot(-4)^*\cdot 3 & 9 + (-3)\cdot(-4)^*\cdot 1 \\ 0 + 1\cdot(-4)^*\cdot 3 & 0 + 1\cdot(-4)^*\cdot 1 \end{vmatrix} = 2$$

Die Berechnung von $x_1 = 3$ erfolgt analog und somit ist eine Lösung für das LGS gefunden.

Die Implementierung des Lösungsverfahrens und ein Generator für Semiringe sind veröffentlicht [23]. Die Umformungen von aus dem Partitionsverfeinerungs-Algorithmus resultierenden Gleichungssystemen der Form $Ax = b$ in eine Fixpunktgleichung setzen additive Inverse voraus. An dieser Stelle stellt sich daher die Frage, ob für alle *-Semiring additive Inverse vorhanden sind. Für distributive Verbände (vgl. 2.3.3) kann im Allgemeinen keine Umformung garantiert und somit auch der Algorithmus nicht auf jedes LGS der Form $Ax = b$ angewandt werden.

Beispiel 3. *Für die reellen Zahlen ist das additive Inverse als Umkehrung der Addition $a + b = c$ durch $c - b = a$ mit $a, b, c \in \mathbb{R}$ definiert.*

Ein Beispiel für einen distributiven Verband ohne die Möglichkeit, die Addition umzukehren, ist $([0,1], max, \cdot, 0, 1)$ *mit* $a^* = 1$ *und* $a \in [0,1]$. *Das Maximum von* 0.03 *oder* 0.4 *und* 0.5 *ist* 0.5. *Die Umkehrung lässt sich aufgrund der Mehrdeutigkeit nicht durchführen.*

Zusätzlich muss die $*$-Operation definiert sein und die oben beschriebenen Einschränkungen erfüllen.

2.3.2 Quotientenkörper und Körpererweiterungen

Die einzelnen Lösungsverfahren für LGS ergeben sich aus den verschiedenen Eigenschaften und Einschränkungen bestimmter Semiringe. Als Körper werden dabei alle Mengen M bezeichnet, bezüglich deren inneren binären Verknüpfungen $\oplus$ und $\otimes$ für alle $a \in M$, ausgenommen für die Null bei $\otimes$, sowohl das additive als auch das multiplikative Inverse existiert. In Anbetracht der Tatsache, dass mehrere effiziente Umsetzungen von Lösungsverfahren für LGS über Körpern existieren, ist es Ziel der Recherche, Körper weitestgehend automatisch generieren zu lassen. Für zwei Fälle von bestimmten Körper-Konstruktionen werden im Rahmen der Implementierung zwei vollautomatische Generierungs-Routinen umgesetzt. In diesem Abschnitt werden die Grundlagen zu den Konstruktionen erläutert.
Für sämtliche Körper ermöglicht das Gauß-Jordan-Verfahren das Lösen von linearen Gleichungssystemen. Besonders effiziente Umsetzungen erfolgen auf Basis der GPU, da dort die Parallelität genutzt werden kann [25]. Die Anwendung von Algorithmus B über Körpern kann somit vollständig gewährleistet werden. Einen Fokus dieser Arbeit bilden jedoch bestimmte Körper, die aufgrund ihrer Eigenschaften vollautomatisiert generiert werden können.

Definition 2.3.4 (Quotientenkörper [26])**.** *Zu jedem nullteilerfreien kommutativen Ring mit einem Einselement e, auch bezeichnet als Integritätsbereich* $(R, +, \cdot)$, *existiert ein (kommutativer) Körper mit folgenden Eigenschaften:*

> $(K, +, \bullet)$ *enthält einen zu* $(R, +, \cdot)$ *isomorphen Unterring.*
>
> *Alle Elemente von K lassen sich als Quotienten* $ab^{-1}(a, b \in R; b \neq 0)$ *darstellen.*
>
> *Addition*: $\frac{a}{b} + \frac{c}{d} = \frac{ad+cb}{bd}$
>
> *Multiplikation*: $\frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd}$

Die Division von zwei Elementen x, y kann auf die Multiplikation von $x \cdot y^{-1}$ zurückgeführt werden. Im Fall der Quotientenkörper lässt sich das Inverse eines Elementes

$\frac{a}{b}$ einfach durch Vertauschen von Dividend und Divisor $\frac{b}{a}$ konstruieren. Für alle euklidischen Ringe ist es darüber hinaus auch möglich den erweiterten euklidischen Algorithmus zur Bestimmung des größten gemeinsamen Teilers zu verwenden (ggT). Aufgrund der Beschränkung des Wertebereichs von Zahlen in Rechnern und der Rechenregeln für die Addition und Multiplikation ist das Kürzen nach der Operation notwendig, um einen unnötigen Überlauf zu vermeiden.

Definition 2.3.5 (Euklidischer Ring [27])**.** *Ein nullteilerfreier Ring R heißt euklidisch, wenn es eine Abbildung (euklidische Normfunktion) $g\colon R \setminus 0$ gibt, sodass es zu $a, b \in R$ mit $b \neq 0$ stets $q, r \in R$ mit*

$$a = qb + r \text{ mit } r = 0 \text{ oder } g(r) \leq g(b)$$

gibt.

Beispiel 4. *Ein Beispiel sind die ganzen Zahlen $\mathbb{Z}$. Eine Normfunktion ist durch den Absolutbetrag gegeben.*

Ein weiteres Beispiel für euklidische Ringe bilden die Polynome $\mathbb{K}[X]$ über einem Körper $\mathbb{K}$. Als Normfunktion ergibt sich $v(f) = grad(f) + 1$. Die Division mit dem Rest von zwei Polynomen $a, b \in \mathbb{K}[X]$ erfolgt durch die Subtraktion eines Vielfachen des Polynoms b von a so lange, bis der Rest $r \in \mathbb{K}[X]$ $grad(r) < grad(b)$ erfüllt. Algorithmen zur Berechnung des ggT von Polynomen über einem Körper sind somit analog zu den natürlichen Zahlen umsetzbar [28].
Eine weitere Möglichkeit, neue Körper zu konstruieren, ist, bereits vorhandene Körper um Elemente aus einem ihrer Oberkörper zu erweitern.

Definition 2.3.6 (Ringhomomorphismus [29])**.** *Gegeben seien zwei Ringe $(R, +, \cdot)$ und $(S, \oplus, \otimes)$. Eine Funktion $\phi\colon R \to S$ heißt Ringhomomorphismus, wenn für alle Elemente $a, b \in R$ gilt:*
$\phi(a + b) = \phi(a) \oplus \phi(b)$ und $\phi(a \cdot b) = \phi(a) \otimes \phi(b)$.

Definition 2.3.7 (Körpererweiterung [30])**.** *Seien K,L zwei Körper mit $K \subset L$ und sei $i\colon K \hookrightarrow L$ ein Ringhomomorphismus mit $i(1) = 1$. Dann heißt $K \overset{i}{\hookrightarrow} L$ Körpererweiterung (L:K).*
L heißt Erweiterungskörper oder Oberkörper von K und K ist der Teilkörper oder Unterkörper von L. Sei $K \overset{i}{\hookrightarrow} L$ eine Körpererweiterung, dann ist L ein K-Vektorraum. Wir bezeichnen $(L : K) = dim_K(L)$ als Grad der Körpererweiterung.

Ein Beispiel für eine Körpererweiterung vom Grad 2 ist $\mathbb{C}\colon \mathbb{R}$, denn der Körper der komplexen Zahlen ist ein Oberkörper des Körpers der reellen Zahlen. Komplexe Zahlen lassen sich in der Form $a + bi$ mit $a, b \in \mathbb{R}$ darstellen. Dabei ist i eine imaginäre Zahl und b steht für den imaginären Anteil der komplexen Zahl. Die Grundrechenarten sind folgendermaßen definiert:

Addition: $(a + bi) + (c + di) = (a + c) + (b + d)i$

Multiplikation: $(a + bi) \cdot (c + di) = (ac - bd) + (ad + bc)i$

Division: $\frac{(a+bi)}{(c+di)} = \frac{(a+bi)(c-di)}{(c+di)(c-di)} = \frac{(ac+bd)}{(c^2+d^2)} + \frac{(bc-ad)}{(c^2+d^2)}i$

Die Subtraktion ist analog zur Addition definiert und die Reell- und Imaginäranteile sind nach jeder Operation einfach zu ermitteln. Weitere Körpererweiterungen werden durch die Aufnahme einer irrationalen Zahl zu der Menge der rationalen Zahlen ermöglicht. Analog zu der Erweiterung der reellen zu den komplexen Zahlen liegen auch hier die Anteile von Unter- und Oberkörper nach den binären Operationen vor. Dies kann für eine automatisierte Generierung von Semiring-Klassen genutzt werden.

Beispiel 5. *Der Körper $\mathbb{Q}$ kann um Elemente aus seinem Oberkörper $\mathbb{R}$ erweitert werden. Eine Körpererweiterung $\mathbb{Q}(\sqrt{2}) = \{a+b\sqrt{2}|a, b \in \mathbb{Q}\}$ vom Grad 2 ermöglicht z.B. nach Anwendung der Multiplikation $ac+(ad+cb)\sqrt{2}+(ad+cb)2$ eine einfache Trennung zwischen den Anteilen von Unter- und Oberkörper.*

Körpererweiterungen auf Basis der rationalen Zahlen umfassen nicht nur Erweiterungen vom Grad 2. Wird anstatt der 2. Wurzel z.B. die 3. Wurzel gewählt, wird eine Körpererweiterung vom 3. Grad gebildet.

Beispiel 6. *Eine Körpererweiterung $\mathbb{Q}(\sqrt[3]{2}) = \{a + b\sqrt[3]{2} + c(\sqrt[3]{2})^2|a, b, c \in \mathbb{Q}\}$ ist vom Grad 3. Die Multiplikation von zwei Elementen $(a+b\sqrt[3]{2}+c(\sqrt[3]{2})^2), (d+e\sqrt[3]{2}+f(\sqrt[3]{2})^2) \in \mathbb{Q}(\sqrt[3]{2})$ ergibt $(ad + 2bf + 2ce) + (ae + bd + 2cf)\sqrt[3]{2}) + (ac + cd)(\sqrt[3]{2})^2 \in \mathbb{Q}(\sqrt[3]{2})$.*

Körpererweiterungen des Körpers $\mathbb{Q}$ bieten zahlreiche Möglichkeiten, neue Körper zu konstruieren. Im Implementierungskapitel wird für eine einfache Körpererweiterung vom Grad 2 eine Vorlage zur Generierung von neuen Körpern vorgestellt.
Neben den Körpern gibt es noch zahlreiche weitere Semiringe, welche sich anhand ihrer Eigenschaften und Einschränkungen zusammenfassen lassen. Im nächsten Abschnitt werden distributive beschränkte Verbände und ein Verfahren zum Lösen von LGS vorgestellt. Das Verfahren benötigt kein additives Inverses.

2.3.3 I-Monoide

Eine weitere mathematische Struktur mit den Eigenschaften eines Semiringes bilden die vollständigen Heyting-Algebren [31].

Definition 2.3.8 (Verband [32])**.** *Es sei P eine geordnete nicht leere Menge. Als Supremum bezeichnet man die kleinste obere Schranke $a \in P$ von zwei Elementen*

$x, y \in P$ ($a = x \vee y$). Das Infimum ist die größte untere Schranke ($a = x \wedge y$).
Wenn das Supremum und Infimum für alle Elemente $x, y \in P$ existiert bezeichnen wir P als Verband.
Existiert für P sowohl eine obere ($\top = \bigvee P$) als auch untere ($\bot = \bigwedge P$) Schranke, so bezeichnen wir P als beschränkten Verband.

Die Eigenschaft von bestimmten Verbänden eine Art der Division zu enthalten, ermöglicht es, LGS über diesen Verbänden zu lösen.

Definition 2.3.9 (Pseudokomplement [32])**.** *Es sei $(L, \leq, \vee, \wedge)$ ein Verband. Sofern für a,b $\in L$ das größte x, welches $a \wedge x \leq b$ erfüllt, existiert, bezeichnen wir x als das Pseudokomplement von a zu b und schreiben $x = a \to b$. Existiert dieses für alle Elementpaare $(a, b) \in L$ so bezeichnet man L auch als Heyting-Algebra.*

Definition 2.3.10 (Vollständige Heyting -Algebra [31])**.** *Existiert für jede Teilmenge S eines Verbandes L sowohl $\bigwedge S$ als auch $\bigvee S$, nennen wir L einen vollständigen Verband. Ist L eine Heyting- Algebra, so nennen wir L eine vollständige Heyting-Algebra.*

Eine vollständige Heyting-Algebra erfüllt für beliebige $x \in L$ und $S \subseteq L$ ebenfalls das unendliche Distributivgesetz $x \wedge \bigvee\{y : y \in S\} = \bigvee\{x \wedge y : y \in S\}$. Jede vollständige Heyting Algebra ist laut Definition ebenfalls beschränkt. Jeder beschränkte und distributive Verband ist ein Semiring.

Definition 2.3.11 (l-Monoid)**.** *Sei $(P, \sqcup, \sqcap)$ ein Verband und $(P, \cdot, e)$ ein Monoid. Wenn $\cdot$ distributiv über $\sqcup$ ist, d.h., $x \cdot (y \sqcup z) = (x \cdot y) \sqcup (x \cdot y)$ und $(x \sqcup y) \cdot z = (x \cdot z) \sqcup (y \cdot z)$ für alle $x, y, z \in P$, bezeichnen wir $(P, \sqcup, \cdot)$ als l-Monoid.*
Als beschränkt bezeichnen wir $(P, \sqcup, \cdot, 0, e)$, wenn es ein $\bot = 0 \in P$ gibt, das $x \cdot 0 = 0 = 0 \cdot x$ für alle $x \in P$ erfüllt. Jeder beschränkte l-Monid ist ein Semiring $(L, \sqcup, \cdot, 0, e)$.

Ein Algorithmus zum Lösen von LGS über Heyting Algebren, welche zu den l-Monoiden gehören, [31] nutzt das Pseudokomplement [15]. Aufgrund der Distributivität von $\cdot$ über $\sqcup$ kann das Pseudokomplement für beliebige $a, b \in P$ in l-Monoiden vorausgesetzt werden. Eine mögliche Lösung für das LGS $Ax = b$ kann mittels des Pseudokomplements berechnet werden.

Satz 1. *Es sei $M = \{1, ..., m\}$ und $N = \{1, ..., n\}$. Weiterhin sei $A \in P^{m \times n}$. Der Lösungsvektor $\tilde{x} \in P^m$ wird mit $j \in M$ definiert durch: $\tilde{x_j} = \sqcap_{i \in N}(a_{ij} \to b_{ij})$*

Das Ergebnis der Berechnung ist entweder die größte Lösung oder ungültig. Dies bedeutet, folgende Aussagen sind äquivalent [15]:

$A\tilde{x} \leq b$

Für alle $x \in P^n$, die $Ax = b$ erfüllen, gilt $x \leq \tilde{x}$

Die Gleichung $Ax = b$ ist lösbar, nur wenn $A\tilde{x} = b$ gilt.

Algorithmus B ist somit auf alle beschränkten und distributiven Verbände anwendbar. Jeder endliche distributive Verband ist beschränkt und somit ein Semiring. Diese Art von Verbänden ist unter dem Aspekt der Semiring-Generierung besonders interessant. Sämtliche Elemente eines solchen Verbandes lassen sich anhand join-irreduzibler Elemente als endliche Mengen darstellen [32].

Definition 2.3.12 (Join-irreduzible Elemente [32])**.** *Sei P ein Verband. Ein Element $x \in P$ heißt join-irreduzible, falls*

$x \neq 0$ *(Für den Fall das P 0 enthält.) und*

$x = a \vee b$ *impliziert $x = a$ oder $x = b$ für alle $a, b \in P$ gelten.*
Die Menge der join-irreduziblen Elemente in P bezeichnen wir mit $\mathcal{J}(P)$.

Die Darstellung von endlichen distributiven Verbänden anhand von Mengen ist von Birkhoff bewiesen worden und bezieht sich auf *down-sets*.

Definition 2.3.13 (Down-sets [32])**.** *Q ist ein down-set, falls für alle $x \in Q, y \in P$, für die $y \leq x$ gilt, auch $y \in Q$ impliziert wird.*
Wir bezeichnen die geordnete Menge aller down-sets eines Verbandes P mit $\mathcal{O}(\mathcal{J}(P))$

Satz 2 ([32])**.** *Sei P ein endlicher distributiver Verband. Die Abbildung $\eta\colon P \to \mathcal{O}(\mathcal{J}(P))$, $\eta(a) = \{x \in \mathcal{J}(P) | x \leq a\}$, ist ein Isomorphismus von P zu $\mathcal{O}(\mathcal{J}(P))$.*

Anhand Birkhoffs Satzes zur Darstellung von endlichen distributiven Verbänden kann unter Angabe der join-irreduziblen Elemente der entsprechende Semiring generiert werden. Die Addition und Multiplikation sind dann durch Vereinigung und Schnitt der Mengen gegeben [32].
Körper und Heyting-Algebren ermöglichen eine voll automatisierte Anwendung des Algorithmus B. Beide Algorithmen nutzen dabei eine Variante der Division. Ein Beispiel für die Anwendung von Algorithmus B über Semiringen, ohne die Möglichkeit für alle Paare $a, b \in S$ die Division gewährleisten zu können, sind endliche Ringe.

2.3.4 Restklassenringe

Semiringe beinhalten nicht zwangsläufig ein additives Inverses. Ist dies jedoch der Fall, werden sie als Ringe bezeichnet. Endliche Ringe ergeben sich durch die Restklassen bei der Division von ganzen Zahlen durch eine positive ganze Zahl. Eine Zahl

$a > 0 \in \mathbb{N}$ teilt eine zweite Zahl $b > 0 \in \mathbb{N}$, falls eine dritte Zahl $c > 0 \in \mathbb{N}$ existiert, sodass $a \cdot c = b$ gilt. Ist dies nicht der Fall, so existiert eine Zahl $0 < r < a \in \mathbb{N}$ und es gilt $a \cdot c + r = b$.

Definition 2.3.14 (Restklassenring [33]). *Ist $n \geq 2$ eine natürliche Zahl, dann werden ganze Zahlen mit gleichem Rest bei Division durch n zu sogenannten Restklassen modulo n zusammengefasst. Zwei ganze Zahlen sind in derselben Restklasse, wenn ihre Differenz durch n teilbar ist. Die Restklassen bilden zusammen mit der unten erklärten Addition und Multiplikation den Restklassenring, der mit $\mathbb{Z}/n$ bezeichnet wird (sprich Z modulo n).*

Sei n eine Primzahl, so ist der Restklassenring ein endlicher Körper und es kann das Gauß-Verfahren angewandt werden. Ansonsten erhalten wir einen endlichen Ring ohne multiplikatives Inverses, welches für das Gauß-Verfahren erforderlich ist. Das Gauß-Verfahren basiert auf der Umformung der Ausgangsmatrix in eine Matrix in zeilenreduzierter Form.

Definition 2.3.15 (Zeilenreduzierte Form [34]). *Eine Matrix $A \in K^{m \times n}$ heißt zeilenreduziert, wenn gilt:*

> *Keine Nullzeile steht oberhalb einer Zeile $\neq 0$, d.h. die ersten r Zeilen seien die Zeilen $\neq 0$.*
>
> *Wenn der erste Eintrag $\neq 0$ in Zeile i in der Spalte j_i auftritt, so gilt $j_1 < j_2 < ..j_r$*

Lemma 3 (Äquivalenzumformungen beim Gauß-Verfahren [10]). *Ist A eine $m \times n$ Matrix und $b \in K_m$, dann bleibt die Lösungsmenge des LGS $Ax = b$ invariant unter folgenden Operationen:*

> *1.Multiplikation einer Gleichung mit einem Skalar $\lambda \in K \setminus \{0\}$.*
>
> *2.Addition der i. Gleichung $\alpha_{i,1}x_i + \ldots + \alpha_{i,n}x_n = \beta_i$*
> *zur j. Gleichung $\alpha_{j,1}x_i + \ldots + \alpha_{j,n}x_n = \beta_j$ für $i, j \in \{1 \ldots m\}$ mit $i \neq j$.*
>
> *3.Vertauschung zweier Gleichungen.*

Durch Anpassung des Gauß-Verfahrens erhält man einen Algorithmus, um lineare Gleichungssysteme $Ax = b$ über einem Ring $\mathbb{Z}_q$ zu lösen. Die oben aufgeführten Umformungen sind auf Restklassenringe anwendbar, jedoch mit einem wesentlichen Unterschied. Dieser besteht in der Einschränkung von $c \neq 0$ bei der Umformung 1 in Lemma 3, denn wenn c ein Nulleiler ist, kommen Lösungen hinzu, welche nicht in der Lösungsmenge des ursprünglichen LGS liegen.

Lemma 4 (Umformungen bei LGS über Restklassenringen)**.** *Bei der Multiplikation mit einer von 0 verschiedenen Zahl ist eine Lösung von* $a_1x_1 + a_2x_2 + ... + a_nx_n = a$ *auch eine Lösung von* $ca_1x_1 + ca_2x_2 + ... + ca_nx_n = ca$*, da* $c \cdot (a_1x_1 + a_2x_2 + ... + a_nx_n) = (c \cdot a) \Longleftrightarrow (c \cdot a) = (c \cdot a)$*. Jedoch ist die Lösung der modifizierten Gleichung mit c, nicht zwangsläufig eine Lösung der ursprünglichen Gleichung, sofern* $c = 0$ *oder ein Nullteiler ist.*

Einen Beweis dafür, dass die Lösungsmenge von $ca_1x_1 + ca_2x_2 + ... + ca_nx_n = ca$ nicht zwangsläufig eine Lösungsmenge für $a_1x_1 + a_2x_2 + ... + a_nx_n = a$ ist, erhalten wir durch ein Gegenbeispiel:

Beispiel 7. $3x = 5$ *besitzt die eindeutige Lösung* $x = 7$ *in* $\mathbb{Z}_8$ *, da* $3 * 7 = 21$ *und* $21 \ mod \ 8 = 5$*.*
Durch Multiplikation der Gleichungen mit dem Nullteiler 2 erhalten wir $2 \cdot 3x = 2 \cdot 5$*. Die umgeformte Gleichung besitzt in* $\mathbb{Z}_8$ *folgende Lösungsmenge* $\{3, 7\}$*. Jedoch ist* $3 * 3 = 1$ *und somit keine Lösung der ursprünglichen Gleichung.*

In dieser Arbeit wird u.a. das Lösen von LGS über endlichen Ringen $\mathbb{Z}_q$ durch die oben beschriebene Anpassung des Gauß-Verfahrens eigenständig in einer Implementierung umgesetzt, um die Anwendung von Algorithmus B zu erweitern. In der Literatur sind Verfahren wie das Hensel-Lifting oder die Verwendung der Smith-Normalform zu finden.
Nach dem chinesischen Restsatz besitzt das LGS $Ax = b \, mod \, q$, wobei A eine Matrix über $\mathbb{Z}_q$, b ein gegebener Vektor über $\mathbb{Z}_q$ und $q = \prod_{i=0}^{k} p_i^{e_i}$ die Primfaktorzerlegung von q ist, eine Lösung $x \in \mathbb{Z}_q$, genau dann, wenn $Ax = b \, (mod \, p_i^{e_i})$ lösbar für alle $1 \leq i \leq k$ ist. Diese Erkenntnis ermöglicht es, den Lösungsweg in Teilprobleme aufzuteilen. Die Anzahl der Teilprobleme ergibt sich durch die Primfaktorzerlegung $q = \prod_{i=0}^{k} p_i^{e_i}$ [35]. Das Hensel-Lifting ermöglicht die Berechnungen der Lösungsvektoren der einzelnen Teilprobleme. Ein LGS $Ax = b \, (mod \, p_i^{e_i})$ baut auf der Lösung von $Ax = b \, (mod \, p_i)$ auf.

Verfahren 1 (Hensel-Lifting [36])**.** *Die Lösung eines LGS* $Ax = b \, (mod \, p^i)$ *wobei* $p, i \in \mathbb{N}$ *und p eine Primzahl ist, lässt sich auf einer Lösung* x_0 *aus der Lösungsmenge des LGS* $Ax = b \, (mod \, p)$ *aufbauen. Angenommen uns liegen alle ganzzahligen Vektoren* $(x_0, ..., x_{i-1})$ *vor , sodass*

$$A(x_0 + x_1p + ...x_{i-1}p^{i-1}) \equiv b \ (mod \ p^i) \tag{9}$$

gilt. Als Nächstes wollen wir einen Vektor x_i *berechnen, der*

$$A(x_0 + x_1p + ... + x_{i-1}p^{i-1} + x_ip^i) \equiv b \ (mod \ p^{i+1}) \tag{10}$$

löst. Betrachten wir die Gleichung (9) im Raum der ganzen Zahlen, wissen wir, dass Folgendes gilt:

$$A(x_0 + x_1 p + ...x_{i-1} p^{i-1}) = b - p^i y_i \tag{11}$$

Es muss eine ganze Zahl y_i existieren, die Gleichung (11) erfüllt. Durch Einsetzen von $b - p^i \cdot y_i$ aus (11) in (10) erhalten wir durch Kürzen mit p^i daher im Raum der ganzen Zahlen:

$$b - p^i y_i + A x_i p^i = b - p^{i+1} y_{i+1} \Leftrightarrow -y_i + A x_i = p y_{i+1} \tag{12}$$

und somit $Ax_i \equiv y_i \ mod\ p$.

Ein Nachteil des Hensel-Lifting ist, dass die Primfaktorzerlegung $\prod p_i^{e_i}$ von q vorliegen muss. Bis heute sind keine effizienten Algorithmen für die Primfaktorzerlegung einer natürlichen Zahl bekannt, deren Primfaktoren sehr groß sind. Des Weiteren ist beim Lösen der Teilprobleme unklar, falls mehrere Lösungen für einen Lifting-Schritt vorliegen, welche zum Endergebnis führen. Analog zu dem adaptierten Gauß-Verfahren kann erst nach dem Durchlauf jedes möglichen Lösungspfades oder bis eine Lösung ermittelt worden ist eine endgültige Aussage über die Lösbarkeit des LGS getroffen werden.
Die Lösbarkeit eines LGS $Ax = b \bmod q$ wird durch die Lösbarkeit von $Ax = b\,(mod\, p^e)$ bestimmt. Dieses Problem lässt sich auf das Lösen von linearen diophantischen Gleichungen reduzieren. Besitzt ein Gleichungssystem der Form $Ax + p^e y = b$ eine Lösung $x', y' \in \mathbb{Z}$, so gilt $Ax' = b \bmod p^e$ [35]. Die Lösung von linearen diophantischen Gleichungen in Polynomialzeit ist anhand des erweiterten euklidischen Algorithmus möglich [37]. Die Form $Ax + p^e y = b$ eines LGS ist die Ausgangslage für eine Komplexitätsanalyse eines Verfahrens, das die Smith-Normalform verwendet. Die Berechnung der Smith-Normalform basiert dabei auf einem zufallsbedingten Verfahren, das zu der Klasse der Monte-Carlo-Algorithmen gehört [38]. Monte-Carlo-Algorithmen akzeptieren mit einer nach oben beschränkten Wahrscheinlichkeit auch falsche Ergebnisse (vgl. [39]). Die Ergebnisse der Analyse zeigen, dass das Lösen von LGS über $\mathbb{Z}_q$ in randomized NC^2 liegt.

Definition 2.3.16 (Nick's Class [40])**.** *NC^i ist die Klasse von Problemen, welche mittels Schaltkreisen polynomialer Größe und einem fan-in von 2 in $O(log^i\, n)$ gelöst werden können. Ein fan-in ist die Anzahl der Eingänge, die ein Gatter verarbeiten kann.*

Die Komplexitätsanalyse auf Basis des Hensel-Liftings und der Primfaktorzerlegung von q zeigt, dass das Lösen von LGS über $\mathbb{Z}_q$ in NC^3 liegt.[35]
Neben der Umsetzung von zwei Algorithmen zur Lösung von LGS über $\mathbb{Z}_q$ beinhaltet

diese Arbeit statistische Erhebungen der Laufzeiten von Partitionsverfeinerungs-Algorithmus B für verschiedene Restklassenringe.

2.3.5 Direktes Produkt

Neue Semiringe lassen sich u.a. durch das direkte Produkt erzeugen. Die Operationen werden dabei komponentenweise definiert und lassen sich somit automatisiert konstruieren.

Definition 2.3.17 (Direktes Produkt Semiringe)**.** *Wir bezeichnen die Menge* $\{(r_1, ..., r_n) | r_1 \in R_1, ..., r_n \in R_n\}$ *als direktes Produkt* $R = R_1 \times ... \times R_n$*.*

Addition: $(r_1, ..., r_n) + (r_1^{'}, ..., r_n^{'}) = (r_1 + r_1^{'}, ..., r_n + r_n^{'})$

Multiplikation: $(r_1, ..., r_n) \cdot (r_1^{'}, ..., r_n^{'}) = (r_1 \cdot r_1^{'}, ..., r_n \cdot r_n^{'})$

Die komponentenweise erzeugte mathematische Struktur ist ebenfalls ein Semiring [41].

Für jedes direkte Produkt kann eine Funktion zur Lösung eines LGS komponentenweise aus den Verfahren der einzelnen Semiringe zusammengesetzt werden, falls für jeden Semiring ein solches Verfahren bekannt ist. Das direkte Produkt ermöglicht somit eine vollautomatisierte Generierung eines neuen Semirings. Ein gewichteter Automat über einem solchen Semiring kann daher unter Anwendung von Algorithmus B auf äquivalente Zustände untersucht werden.
In Anbetracht der Semiring-Vielfalt beinhaltet diese Arbeit eine Ausarbeitung zur Generierung von Semiring-Klassen in C#. Die Implementierungen der einzelnen Semiringe für diese Arbeit erfüllen die Rahmenbedingungen für das Faktorisieren innerhalb von Algorithmus B. Ein Anwender bekommt zusätzlich die Möglichkeit, einen beliebigen Semiring u.a. durch Angabe von Addition und Multiplikation in das bestehende Programm einzufügen.

3 Implementierung

Die Anwendung des Partitionsverfeinerungs-Algorithmus für gewichtete Automaten setzt voraus, dass für den Semiring des Automaten ein Lösungsverfahren für LGS vorhanden ist. Die implementierten Semiringe des Programms enthalten alle eine Methode, um ein gegebenes LGS, das mindestens eine Lösung hat, zu lösen. Ein Anwender soll für den Fall, dass er einen Automaten auf äquivalente Zustände überprüfen möchte, dessen Semiring nicht im Programm vorhanden ist, die Möglichkeit haben, diesen erstellen zu lassen. Ziel ist es daher, das Programm so weit wie möglich automatisiert um weitere Semiring-Klassen erweitern zu können. Ein Grundgerüst für die Struktur und das Verhalten einer C#-Klasse, die einen Semiring repräsentiert, ist durch Definition 2.1.3 gegeben. Konstante Funktionsdefinitionen für Addition und Multiplikation sind somit feste Bestandteile jeder Semiring-Klasse. Die Programmcode-Generierung ermöglicht es, gleichbleibende Codefragmente, wie Funktionsdefinitionen oder String-basierte Konstruktoren automatisiert zu erstellen und somit einen erheblichen Arbeitsaufwand einzusparen.
Im ersten Unterkapitel wird eine Einführung in die Grundlagen der Code-Generierung gegeben. Im Anschluss daran wird auf die Transformation bestimmter Compiler eingegangen. Einige Compiler nutzen abstrakte Syntaxbäume, um die Übersetzung von Programmcode in Hochsprache in Maschinencode abzuwickeln. Das Konzept von Transformation anhand von Baumstrukturen wird ebenfalls innerhalb von CodeDOM, dem für diese Arbeit verwendeten Framework, verwendet. Der Hauptteil der Implementierung besteht darin, die Eingaben des Anwenders in eine Baumstruktur zu überführen, um den C#-Generator von CodeDOM zur Code-Generierung nutzen zu können. Das Erstellen einer Baumstruktur wird in den anschließenden Unterkapiteln ausführlich vorgestellt. Zum Schluss wird die Anwendung der grafischen Schnittstelle des Semiring-Generators an Beispielen erklärt.

3.1 Grundlagen der Code-Generierung

Das automatisierte Erstellen von spezifischem Quellcode auf der Grundlage von Modellen, Vorlagen oder Eingabeparametern wird als generative Programmierung bezeichnet. Generative Programmierung ist ein Paradigma für die Modellierung von Software-System-Familien, sodass unter wenigen Angaben Programmcode aus einzelnen wiederverwendbaren implementierten Komponenten zu einem Produkt verknüpft wird. Anstatt einzelne Programme eines bestimmten Problembereiches von null an zu erstellen, können einzelne Instanzen aus diesem Bereich anhand eines gemeinsamen generativen Modells erstellt werden. Dieses Modell beinhaltet die Beschreibung des Problemraums, die einzelnen Bausteine und das Wissen, wie einzelne

Komponenten verknüpft werden. Der Begriff „Generator“ bedeutet zunächst, dass es sich um ein Programm handelt, welches als Eingabe eine abstrahierte Spezifikation erhält und daraus Programmcode generiert. Generatoren unterliegen dem Paradigma der generativen Programmierung. Die Eingabe enthält nur die notwendigsten Informationen für die Generierung. Ein Generator ist in der Lage, einen Bogen zwischen der abstrahierten Spezifikation und der Implementierung zu spannen. Dieser Bogen ist eine Abbildung zwischen der Menge aller möglichen Eingabekombinationen und den zulässigen Kombinationen aus den Komponenten. Eine geringfügige Änderung in den Eingaben kann eine vollkommen andere Implementierung ergeben. Der Vorteil von Generatoren ist dabei die Tatsache, dass Programmierer diese Abbildung nicht manuell vornehmen müssen. (vgl. [K. 9, S. 332-333] [42])
Die Wiederverwendbarkeit von Software ist eine weitere Motivation der generativen Programmierung, welche in dieser Arbeit für die teils semiautomatisierte Generierung von Semiringen verwendet wird.

3.1.1 Programmcode-Generierung

Automatisierte Generierung von Programmcode in Abhängigkeit von Nutzereingaben ist eine Umwandlung der Eingabe in den spezifischen Quellcode als Ausgabe. Code-Generierung arbeitet größtenteils auf der Grundlage zweier Techniken. Zum einen auf der Basis von Komponenten und zum anderen mittels Transformation. Komponenten-gestützte Umsetzungen werden z.B. in GUI-Buildern eingesetzt [43]. „Transformation ist eine automatisierte und semantisch korrekte Modifikation eines Programms “[43].
Transformationen werden dabei nicht auf das Programm in Textform angewandt, sondern auf Modelle. Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) umfasst sämtliche Techniken, welche aus formalen Modellen lauffähige Software erzeugen können. „Formal“ bedeutet in diesem Kontext, dass das Modell die Software so weit abstrahiert, dass es die wesentlichen Aspekte abdeckt. Gängige Repräsentanten sind z.B. UML-Modelle oder Syntaxbäume. Ein Generator erhält das Modell als Eingabe und übersetzt die Inhalte in eine Zielsprache. [44]
Transformationen finden sich in vielen Bereichen der Informatik wieder. Graphersetzungssysteme sind ein geeignetes Beispiel für ein Werkzeug, um Systemzustände dynamisch zu modellieren [45]. Produktionsregeln sind ein Beispiel, wie innerhalb der Chomskysprachen Variablen auf der linken Seite in Variablen oder Terminalsymbole der rechten Seite überführt werden [6]. Das zweite Beispiel spielt für Generatoren insofern eine Rolle, da die Syntax eines Programms mittels einer kontextfreien Grammatik beschrieben werden kann.
Allgemein läuft eine Transformation zwischen zwei Modellen nach bestimmten Re-

geln ab. Programme zur Umwandlung von verschiedenen Modellen beinhalten eine Definition, die eine Anleitung zur Umwandlung eines Modells ist. Die Anleitung selbst setzt sich aus Transformationsregeln zusammen [46],[47].

Begriffserklärung 3.1.1 (Transformationsregel [46],[47]). *Eine Transformationsregel ist eine Beschreibung, wie eines oder mehrere Konstrukte in der Quellsprache in eines oder mehrere Konstrukte in der Zielsprache transformiert werden können.*

Begriffserklärung 3.1.2 (Transformationsdefinition [46],[47]). *Eine Transformationsdefinition ist eine Menge von Transformationsregeln. Zusammen beschreiben sie, wie ein Modell in der Quellsprache in ein Modell der Zielsprache transformiert werden kann.*

Begriffserklärung 3.1.3 (Transformation [46],[47]). *Eine Transformation ist das automatische Generieren eines Zielmodells aus einem Quellmodell entsprechend einer Transformationsdefinition mit Erhalt der Semantik, sofern die Sprache dies zulässt.*

Das gewählte Namespace des „Code Document Object Model "(CodeDOM) nutzt das Konzept der Baumstruktur zur Darstellung von Programmcode. Der Generator des .Net CodeDOM transformiert das Modell in den Programmcode einer angegebenen Zielsprache. Das Prinzip von Transformation zwischen formalen Sprachen anhand von abstrakten Syntaxbäumen wird ebenfalls in einigen Compilern genutzt.

3.1.2 Einführung Transformation in Compiler

Transformation ist ein wesentlicher Bestandteil von Compilern. Ein in Hochsprache erstelltes Programm wird bei Frontend-Backend-Compilern mittels Parsern in einen AST (AST: abstract syntax tree) umgewandelt. Die Übersetzung des AST in Maschinencode wird als Synthese bezeichnet. Baumstrukturen ermöglichen es, Programmbestandteile geeignet darzustellen, da z.B. Klassendefinitionen, Anweisungen oder While-Schleifen hierarchisch aufgebaut sind. Unnötige Informationen aus der für den Menschen leserlichen Hochsprache wie das Semikolon entfallen. Diese Informationen werden jedoch für die Umwandlung der Hochsprache in den AST, welche vom Parser durchgeführt wird, zum Erkennen der Struktur benötigt. (vgl. [48])
Programmiersprachen gehören zu den formalen Sprachen. Grammatiken ermöglichen eine endliche Beschreibung von unendlichen formalen Sprachen.

Definition 3.1.1 (Kontextfreie Grammatiken [6]). *Eine Grammatik ist ein 4-Tupel* $G = (V, \Sigma, P, S)$. *V ist eine endliche Menge von Variablen, Σ ist eine endliche Menge von Terminalsymbolen. Dabei muss $V \cap \Sigma = \emptyset$ sein. P ist die endliche Menge*

der Regeln oder Produktionsregeln. Formal ist P eine endliche Teilmenge von $(V \cup \Sigma)^+ \times (V \cup \Sigma)^*$. $S \in V$ ist die Startvariable.
Seien $u, v \in (V \cup \Sigma)^*$. Wir definieren die Relation $u \Rightarrow_G v$ (in Worten: u geht unter G unmittelbar über in v), falls u und v die Form haben

$$u = xyz \tag{13}$$

$$v = xy^{'}z \text{ mit } x, y \in (V \cup \Sigma)^*. \tag{14}$$

und $y \rightarrow y^{'}$ eine Regel in P ist. Für kontextfreie Grammatiken gelten zusätzlich die Einschränkungen, dass auf der linken Seite nur eine Variable stehen darf und die Anzahl der Symbole kleiner oder gleich der Anzahl der Symbole auf der rechten Seite sein muss.

Die Sprache ergibt sich durch alle ableitbaren Worte einer Grammatik. Worte setzen sich dabei nur aus Terminalsymbolen zusammen. Die Ableitung eines Wortes lässt sich anhand eines AST darstellen. Das Wort ergibt sich aus den Blättern, gelesen in der Reihenfolge von links nach rechts. Ein Ausdruck zur Addition von zwei Variablen ist z.B. ein mögliches Wort einer formalen Sprache.

$$\begin{pmatrix} Id & \rightarrow & Var\{cons(\text{"}Var\text{"})\} \\ Var & \rightarrow & Exp \\ IntConst & \rightarrow & Exp\{cons(\text{"}Int\text{"})\} \\ \text{"}-\text{"}Exp & \rightarrow & Exp\{cons(\text{"}Uminus\text{"})\} \\ Exp\text{"}*\text{"}Exp & \rightarrow & Exp\{cons(\text{"}Times\text{"})\} \\ Exp\text{"}+\text{"}Exp & \rightarrow & Exp\{cons(\text{"}Plus\text{"})\} \\ Exp\text{"}-\text{"}Exp & \rightarrow & Exp\{cons(\text{"}Minus\text{"})\} \\ Exp\text{"}=\text{"}Exp & \rightarrow & Exp\{cons(\text{"}Eq\text{"})\} \\ Exp\text{"}>\text{"}Exp & \rightarrow & Exp\{cons(\text{"}Gt\text{"})\} \\ \text{"}(\text{"}Exp\text{"})\text{"} & \rightarrow & Exp \end{pmatrix}$$

Abbildung 7: Produktionsregeln einer kontextfreien Sprache. Quelle: [49]

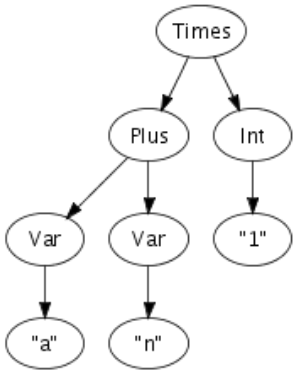


Abbildung 8: Abstrakter Syntaxbaum für $(a+n) \cdot 1$. Quelle: [49]

Die Syntax einer Programmiersprache lässt sich nicht vollständig von kontextfreien Grammatiken beschreiben. Kontextabhängige Anforderungen werden daher als Zusatz formuliert. Die Umsetzung mittels kontextabhängiger Grammatiken lässt sich aufgrund der wachsenden Komplexität in der Praxis nicht vornehmen. Die Erkennung von einzelnen Terminalsymbolen wird als lexikografische Analyse bezeichnet und basiert auf regulären Ausdrücken. Diese ergibt die Eingabe für die Syntaxanalyse. (vgl. [48])
Zur Konstruktion des Ableitungsbaumes existieren zwei Verfahren. Das Top-Down-

Verfahren ist eine Konstruktion von der Wurzel zu den Blättern, während beim Bottom-Up-Verfahren von den Blättern aus gestartet wird. Bei beiden Verfahren können Mehrdeutigkeiten auftreten. Ein Ausdruck wie *2*3+4* ergibt zwei verschiedene Ableitungsbäume, wenn nicht eindeutig festgelegt wird, welche Ableitung Vorrang hat.

Zur Vermeidung von Mehrdeutigkeiten werden Präzedenz- und Assoziativitätsregeln

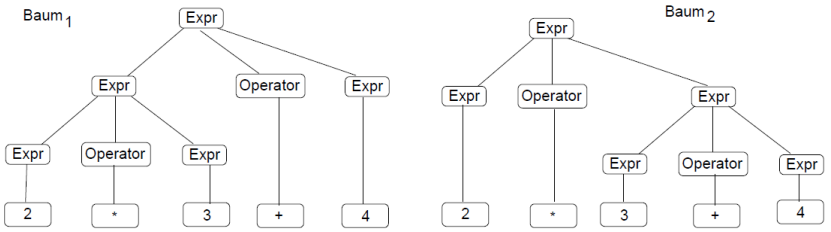


Abbildung 9: Die unterste Baumebene enthält einzelne Terminalsymbole. Mögliche Ableitungen nach oben hin mittels kontextfreier Grammatiken. Quelle: [48]

festgelegt. Aufgrund der mathematischen Regel „Multiplikation geht vor Addition“ hat der Operator $*$ eine höhere Präzedenz als der Additionsoperator $+$. Die Präzedenzen von Operatoren werden entweder als zusätzliche Information angegeben, um die Grammatik nicht weiter aufzublähen, oder in die Grammatik selbst integriert. Für den Fall, dass drei Operatoren aus der gleichen Präzedenzgruppe aufeinander treffen, d.h. $a \bullet b \circ c$, werden die Ausdrücke oder Terminalsymbole z.B. links-assoziativ angeordnet, d.h. $a \bullet b \circ c = (a \bullet b) \circ c$ (vgl. [48]). Das hier aufgeführte Beispiel entspricht einer abstrahierten Form des AST. Bei Compilern, welche als Zwischendarstellung AST verwenden, enthalten die Knoten noch weitere Attribute, welche z.B. Informationen zu Typen und Werten enthalten (vgl. [48]).

Das Prinzip der Weiterverarbeitung eines AST in Maschinencode lässt sich anhand einer Stack-Maschine veranschaulichen. Eine Stack-Maschine arbeitet, wie dem Namen entnommen werden kann, mit einem Stack für die Speicherung von Zwischenergebnissen. Jede Rechnerarchitektur enthält einen Befehlssatz, der eine Menge von Maschinenbefehlen ist. Für den primitiven Ausdruck „5“ erhalten wir im Stack den Befehl *PUSH 5*. Dabei steht *PUSH w* für das Befüllen des Stacks von oben mit *w*. Der Befehl *ADD* entfernt die beiden oberen Werte und fügt dann ihre Summe hinzu. Durch Transformationsregeln wird festgelegt, welche Konstrukte des einen Modells auf Konstrukte des anderen Modells abgebildet werden. Maschinencode-Generierung ist abhängig von der Zielplattform. Das hier aufgeführte Stack-Maschinen-Prinzip wird bei der Code-Generierung für die virtuelle RISC-Maschine verwendet. (vgl.

[48])
Einige Compiler wandeln eine Hochsprache in eine Zwischendarstellung, z.B. eine Baumstruktur um und transformieren dieses Modell in Maschinencode. Im Fall der generativen Programmierung anhand von Modellen liegt jedoch im Voraus kein Quellcode in Hochsprache vor, sondern das Modell wird genutzt, um Quellcode in einer Hochsprache zu generieren. CodeDOM nutzt die Umkehrbarkeit des Vorgangs, der vom Compiler beim Parsen durchgeführt wird. Anhand von linearer Transformation wird ein Objektdiagramm in Quellcode einer Hochsprache umgewandelt [50].

3.2 CodeDOM des .net Frameworks

Das CodeDOM (Code Document Object Model) ermöglicht es unabhängig von einer Programmiersprache ein Quellcodemodell zu erstellen. Das Modell setzt sich dabei aus einzelnen CodeDOM-Elementen, die für bestimmte Quellcode-Elemente stehen, zusammen. Das Modell kann anhand der vorhandenen Generatoren, die erweitert werden können, in die Sprachen des Visual-Studio transformiert werden.
Anders als bei den T4-Templates-Textvorlagen des Visual-Studio, welche den zu generierenden Quellcode direkt in Textform beinhalten, arbeitet CodeDOM mit Objekten. T4-Templates generieren Programmcode durch die Angabe von Textblöcken, Kontrollblöcken und Direktiven. Die Textblöcke enthalten den zu generierenden Code. Direktive fassen die Angaben zusammen, wie der Code generiert werden soll. Kontrollblöcke sind Programmcode-Fragmente, welche z.B. Inhalte von Variablen oder Wiederholungen innerhalb der Textblöcke steuern.
In dem folgenden Unterkapitel wird das Konzept von CodeDOM vorgestellt und ein Beispiel in Bezug auf den AST im Compiler-Abschnitt genommen.

3.2.1 CodeDOM-Konzept

Analog zu der Zwischendarstellung bei Compilern arbeitet CodeDOM mit einzelnen Knotenpunkten, welche im Gesamten einen Objektgraphen bilden. Das Objekt-Modell stellt den zu generierenden Programmcode in einer hierarchischen Baumstruktur dar. Für die einzelnen Programmbausteine werden Klassen des CodeDOM Namespace genutzt. CodeDOM verfolgt die Motivation, Sprachkonstrukte eines Programms in eine Eins-zu-Eins-Beziehung mit der objektorientierten Darstellung zu bringen. Sprachunabhängige Programmkonstrukte, z.B. eine Bedingung (*ConditionStatement*), können abstrahiert werden und sind generisch. Anders verhält es sich mit sprachlichen Elementen, welche nur in einer bestimmten Sprache auftauchen. CodeDOM bietet in solchen Fällen einen Umweg, der nicht mehr als generisch bezeichnet werden kann. Die Funktionsweise von CodeDOM basiert auf einem umfas-

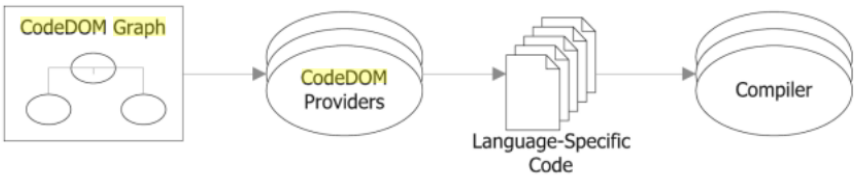


Abbildung 10: Abstraktionsebenen zur Generierung von Quellcode mittels CodeDOM. Quelle: [51]

senden Angebot an Klassen, um Programmkonstrukte in einer Baumstruktur anzuordnen. Den obersten Container des Programms bildet dabei die *CodeCompileUnit*. Der Verweis auf ein CodeDOM-Programmdiagramm findet sich in einer *CodeCompileUnit* und enthält das Programm als Modell. Die *CodeCompileUnit* beinhaltet neben den Verweisen auf CodeDOM-Programmdiagramme Angaben zu den benötigten Assemblies. Ein Assembly ist unter .Net ein Paket, welches sämtliche DLL- und EXE-Dateien verwaltet. Die in einer Assembly enthaltenen Dateien werden als Module bezeichnet. Die Metadaten des Assembly werden im Manifest abgelegt und setzten sich aus drei Komponenten (Bezeichner-Versionsnummer, enthaltene Module, verwendete Assemblies) zusammen.
Die Wurzeln einzelner Klassen werden mittels *CodeTypeDeclaration* angegeben. In den einzelnen Klassen werden Knotenpunkte entweder unter *Members* oder *Statements* hinzugefügt. Das Objektmodell kann mittels *Provider* in Quellcode umgewandelt werden. Neben dem Objektmodell benötigt der sprachspezifische *Provider* einen *TextWriter*, um eine Sequenz von Zeichen in einen Stream einer bestimmten Codierung zu schreiben. CodeDOM unterstützt auch Assembly-Generierung aus Quellcode in Textform.
Die Modelle des CodeDOM Frameworks können anhand der *Provider-Klassen* in Quellcode umgewandelt werden. Die automatisierte Generierung des Modells in Abhängigkeit der Anwendereingaben erfordert Vorlagen in CodeDOM-Namespace-Objekten, welche Objektmodelle für die verschiedenen Bestandteile einer Semiring-Klasse erzeugen. Eine CodeDOM basierte-Vorlage kann z.B. für den konkreten Ausdruck *2*3+4* definiert werden und entspricht Baum1 aus Abbildung 9.

```
CodeBinaryOperatorExpression expr1 =
new CodeBinaryOperatorExpression(
       // Left operand.
                new CodeVariableReferenceExpression("2"),
       // CodeBinaryOperatorType enumeration value of Multiplication.
                CodeBinaryOperatorType.Multiply,
       // Right operand.
```

```
                new CodeVariableReferenceExpression("3"));
CodeBinaryOperatorExpression expr2 =
new CodeBinaryOperatorExpression(expr1,
CodeBinaryOperatorType.Add,
new CodeVariableReferenceExpression("4"))
```

Die Umsetzung von CodeDOM-Vorlagen erfordert beim Programmieren eine Umstellung vom konventionellen Programmieren in einer Hochsprache in die Abstraktion von Programmcode in hierarchisch angeordnete Objekte.

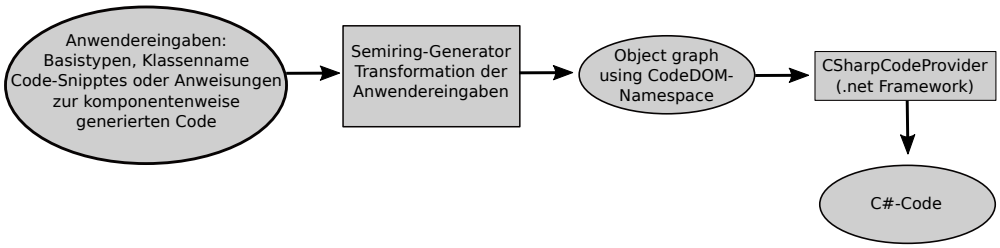


Abbildung 11: Beteiligte Komponenten und Ablauf einer Semiring-Klassen Generierung.

Das Hauptproblem ist es, die Eingaben des Anwenders in ein CodeDOM zu überführen. Im Rahmen dieser Arbeit bildet CodeDOM die Grundlage zur Implementierung des Semiring-Generators. Die Eingaben werden in Bezug auf den Problemraum in Baumstrukturen umgewandelt, die anhand des CodeDOM Namespaces erzeugt werden. Eine Übersicht über die Komponenten des Semiring-Generators sei an dieser Stelle in Abb. 11 im Bezug zum verwendeten Framework gegeben. Design und Implementierung von CodeDOM-Vorlagen für Semiringe werden im nächsten Kapitel ausführlich beschrieben. [52]

Eine weitere Technik, Quellcode effizient wiederzuverwenden ist die Verwendung von generischen Typparametern, die von der formalen Sprache C# unterstützt werden.

3.3 Generika und Reflexion des .net Frameworks

Die Umsetzung des Semiring-Generators nutzt zusätzlich zu dem CodeDOM Namespace des .net Frameworks noch *System.Generics* und *System.Reflection*.

3.3.1 Generika

Generische Programmierung verfolgt ähnliche Ziele wie die generative Programmierung. Eines dieser Ziele ist die Absicht, Codeabschnitte nach Aufgaben zu gliedern. Gemeint ist damit, eine Codestruktur anzustreben, deren Codesegmente einzelnen Aufgaben klar zuzuordnen sind. Zusammen bilden diese Codesegmente eine Komponente des Programms. Generische Programmierung ermöglicht durch Parametrisierung die Realisierung von Komponenten-Familien. Mehrere Komponenten werden von verschiedenen Typen in der gleichen Weise verwendet. (vgl. [43])
Generika (Generics) steht für die Umsetzung des Typparameter Konzepts innerhalb des .Net Framework. Generika bietet zahlreiche generische Auflistungstypen der .net Framework-Klassenbibliothek an (System.Collections.Generic). Zusätzlich ermöglicht Generika die Implementierung von eigenen generischen Klassen und Methoden in C#. Ein generischer Platzhalter für einen beliebigen Semiring wird anhand des generischen Typparameters T von Generika umgesetzt. Die Definition von gewichteten Automaten als *Matrix<T>* ermöglicht eine generische Anwendung von Partitionsverfeinerungs-Algorithmus B. Alle Einträge der Matrix stammen aus dem gleichen Semiring und die Ausführung von Algorithmus B basiert auf den binären Operationen „Multiplikation“ und „Addition“ (Quelle: Programm zu [3]). Eine Instanziierung einer Matrix erzwingt die konkrete Angabe des gewünschten Semirings. Dieses Konzept wird in vielen Programmiersprachen für generische Speicherstrukturen, z.B. *GenericList<T>*, verwendet. Dabei entspricht *GenericList* nicht direkt einem Typen, sondern eher einer Kopie. Unter Angabe eines Typs, der dem Compiler bekannt sein muss, wird bei der Instanziierung einer Variablen ein Typ konstruiert. Ein Semiring-Generator ist im Rahmen eines Programms, welches generische Algorithmen zur Lösung von Fixpunktgleichungen unterstützt, auf der Basis von `C++` Vorlagen umgesetzt worden [23]. C#-Generika ersetzt im Unterschied zu den `C++`-Vorlagen den generischen Typ erst zur Laufzeit. Einige weitere Unterschiede gegenüber `C++`-Vorlagen ergeben sich durch die Einschränkungen, dass ein generischer Typparameter nicht selbst generisch sein und der Typparameter nicht als Basisklasse für den generischen Typ verwendet werden kann. Der Aufruf von arithmetischen Operatoren innerhalb einer generischen C#-Klasse ist ebenfalls untersagt. C#-Generika verbindet somit Wiederverwendbarkeit und Typsicherheit. [53]

3.3.2 Reflexion des .net Frameworks

Die Möglichkeit Programmen, ihre oder die Struktur eines anderen Programms bekannt zu machen oder zu modifizieren, wird in der Informatik als Reflexion bezeichnet. Für den Semiring-Generator ist es wichtig, die anderen Semiring-Klassen zu kennen, falls er eine Klasse konstruiert, die auf anderen Semiringen aufbaut.
Das .net Framework bietet die *System.Reflection*-Bibliothek für die Analyse von Programmstrukturen an. Sämtliche Programme, welche im Rahmen des .net Frameworks erstellt worden sind, liegen als Assembly-Dateien vor (vgl. Abschnitt 3.2.1). In den Daten der Manifestdatei sind alle Module angegeben, die innerhalb der Assembly-Datei zur Verfügung stehen. Reflexion ermöglicht den Zugriff auf Informationen innerhalb der Assembly-Dateien. [54]
Der Aufwand, die Klasseninformationen separat zu verwalten und zu aktualisieren, um den Generator immer mit gültigen Informationen zu versorgen, kann durch Reflexion ersetzt werden. Anhand des *System.Type* und der Reflexion lassen sich Informationen bezüglich eines Typen abrufen. Der Typ selbst kann anhand der Assembly-Datei und des Typnamens geladen werden, da Typen in Modulen enthalten sind. Die Eigenschaften, Methoden und Felder sind innerhalb des Typs definiert und können über die Methoden der *System.Type* nach dem Laden zur Laufzeit abgefragt werden.

Die Einbindung von C#-Generika lässt sich für einen speziellen Fall von Polynomen umsetzen. Die Möglichkeit, Programmstrukturen eines zweiten Programms während der Laufzeit dynamisch abzurufen, erleichtert die Konzipierung eines Generators, der Eingaben in ein CodeDOM überführt und dabei Konstruktoren automatisch erstellt. Das nächste Kapitel beschreibt Design, Implementierung und die semiautomatisierte Generierung von Semiring-Klassen.

3.4 Semiring-Generator

Ziel der Implementierung dieser Arbeit ist es, das Anwendungsfeld des Partitionsverfeinerungs-Algorithmus B um beliebige Semiringe erweitern zu können. Der Anwender soll die Möglichkeit erhalten, einen Semiring durch bestimmte Angaben zu definieren und in das bestehende Programm einzufügen. Der Semiring-Generator setzt die Anwendereingaben in eine Baumstruktur um, die anhand des CodeDOM C#-Generators in C#-Code abgebildet wird. Die Angaben des Anwenders beziehen sich dabei auf das Verhalten und die Eigenschaften des Semirings, die von einer generierbaren C#-Klasse repräsentiert werden müssen.
Automatisiert generierte Klassen setzen voraus, dass genügend Gemeinsamkeiten als Grundlage für eine Vorlage vorhanden sind. Die Bestandteile (Struktur, Verhalten) einer Semiring-Klasse sind durch die Definition 2.1.3 gegeben. Eine weitere Methode zur Lösung von LGS kommt für eine vollautomatisierte Nutzung von Algorithmus B hinzu. Eine Auflistung an dieser Stelle schafft eine Übersicht über die notwendigen Methoden einer C# Semiring-Klasse.

binäre Addition (operator +)

binäre Multiplikation (operator *)

binärer Vergleichsoperator (operator ==)

neutrales Element Addition (Zero)

neutrales Element Multiplikation (One)

Lösen LGS (FindLinearCombination)

In Anbetracht der Vielfalt binärer Operationen lassen sich Semiringe nicht in einer Generierungsvorlage zusammenfassen. In den folgenden Unterkapiteln werden die Umsetzung und der Gebrauch zweier Programme, die zusammen das Ziel verfolgen, die Anwendbarkeit von Algorithmus B zu erweitern, vorgestellt.
Im Designabschnitt werden ein Überblick über die übergeordneten Programmkomponenten und eine feinere Übersicht anhand von UML-Klassendiagrammen gegeben. Anschließend werden allgemeine Komponenten des Generators im Detail erläutert und das Konzept von CodeDOM wird an einem Beispiel demonstriert. In dem darauffolgenden Unterkapitel werden basierend auf den mathematischen Grundlagen aus 2.3 für spezielle Semiringe Implementierungsdetails vorgestellt. Im Anschluss daran werden die grafischen Schnittstellen für den Partitionverfeinerungs-Algorithmus B und den Generator vorgestellt. Die Anwendung des Generators wird an drei Beispielen demonstriert.

3.4.1 Übersicht über Design und Umsetzung

Zunächst wird die Entwicklung vom bestehenden Programm zum Endprogramm vorgestellt. Darauf folgen Erklärungen zum Design und zu den Entscheidungen hinter den gewählten Konzepten. Im Anschluss darauf wird ein Einblick in die CodeDOM basierte Umsetzung des Generators gegeben.

Ausgangs- und Endlage der Implementierung Dieser Arbeit lag bereits zu Beginn eine Konsolenanwendung von Algorithmus B vor. Darin sind die generische Matrix-Klasse und einige Semiringe vollständig implementiert. Außerdem beinhalten die Semiringe bereits eine Umsetzungen des Gauß-Verfahrens und des Lösungsverfahrens für die Heyting-Algebren. Parallel zu der Einarbeitung in die mathematischen Grundlagen wird die Konsolenanwendung u.a. in eine Anwendung mit einer benutzerfreundlichen Oberfläche überführt. Die Grafiken 12 und 13 verschaffen einen Überblick über den Unterschied zwischen Anfangs- und Ausgangssituation der Anwendung. Die Eingabe von Algorithmus B ist ein gewichteter Automat bzw.

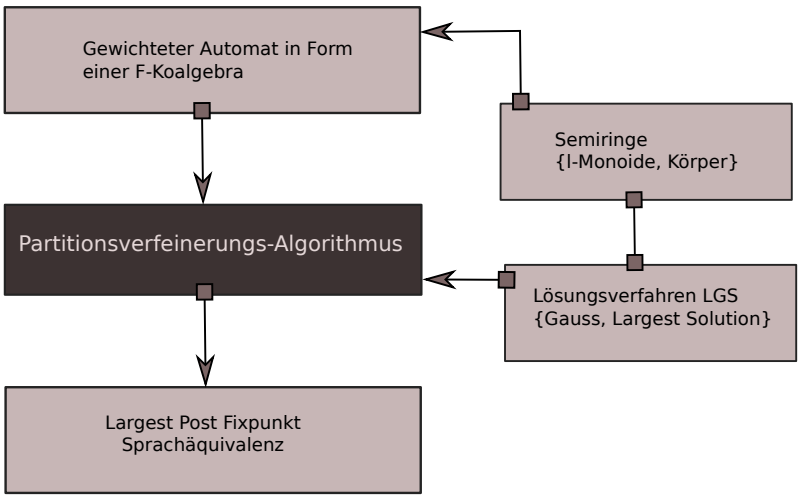


Abbildung 12: Ausgangspunkt für die Entwicklung eines Semiring-Generators zur Erweiterung des Anwendungsfeldes von Algorithmus B.

eine $(A \times X + 1) \times X$ Matrix mit Einträgen aus einem Semiring $\mathcal{S}$. Wie bereits in Abschnitt 3.3 erklärt, ist die Matrix-Klasse generisch aufgebaut. Der Nutzen ist ersichtlich, da der Ablauf von Algorithmus B für jeden Semiring gleich erfolgt. Eine Implementierung der Lösungsverfahren liegt für Körper und l-Monoide bereits zu beginn der Arbeit vor (vgl. Abb. 12). Unter wiederholtem Aufruf des Funktors und

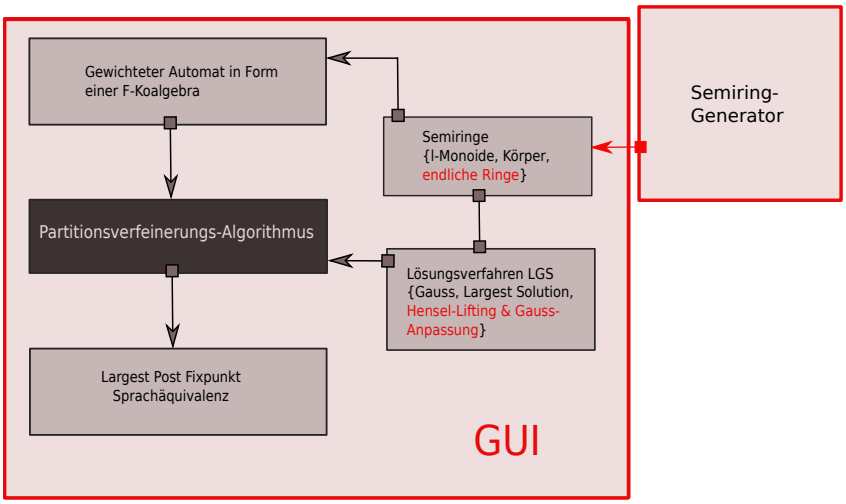


Abbildung 13: Ergebnis der Ausarbeitung des Anwendungsfeld von Algorithmus B.

der Suche nach Linearkombinationen terminiert der Algorithmus letztendlich, wenn die Terminierungseigenschaften, erläutert in 3, erfüllt sind (vgl. Abb. 6).
Die grafische Schnittstelle (GUI) für den Partitionsverfeinerungs-Algorithmus B ist ebenfalls generisch implementiert. Die Modell-View-Controller-Architektur (MVC) ermöglicht den Austausch von einzelnen Komponenten. In der Umsetzung wird dies genutzt, um generische Datenmodelle innerhalb der MVC-GUI bereitzustellen. Die Instanziierung einer generischen Matrix erfolgt somit im korrespondierenden Datenmodell und wird auch direkt dort verwaltet (vgl. links Abb. 13).
Zu der Anwendung von Algorithmus B kommt ein Programm zur Generierung von Semiringen hinzu, wie in Abb. 13 oben rechts dargestellt. Die Entscheidung, den Programmcode-Generator auszulagern, wird im Abschnitt 3.4.1 erläutert. Die Erweiterung um die endlichen Ringe (vgl. Abb. 13) ist auf die Ergebnisse der Recherche von LGS-Lösungsverfahren zurückzuführen. In den nächsten beiden Abschnitten werden die einzelnen Komponenten der zwei Programme und deren Zusammenhänge ausführlich vorgestellt.

Programm Design Die Erweiterung des Anwendungsfeldes des Partitionsverfeinerungs-Algorithmus B unterteilt sich in zwei Programme (vgl. 14). Das erste Programm ist die Ergänzung der bereits vorhandenen Konsolenanwendung um eine grafische Schnittstelle. Beim Starten des Programms wird für jede vorhandene Semiring-Klasse ein eigenes *Datamodell* angelegt. Innerhalb der GUI kann der Anwender einen dieser Semiringe auswählen und einen gewichteten Automaten als Matrix eingeben.

Die Auswahl des Semirings bestimmt letztendlich den Typen der Matrix. Diese Matrix wird von dem entsprechenden *DataModell* erstellt und verwaltet. Jedes *DataModell* benachrichtigt die *View*, sobald eine neue Matrix instantiiert worden ist. Der Anwender sieht in einer Auflistung alle Matrizen unter Angabe des Semirings. Zusätzlich wird dem Anwender sein aktuell erzeugter oder ausgewählter Automat unter Zuhilfenahme der *GraphViz-lib* angezeigt.

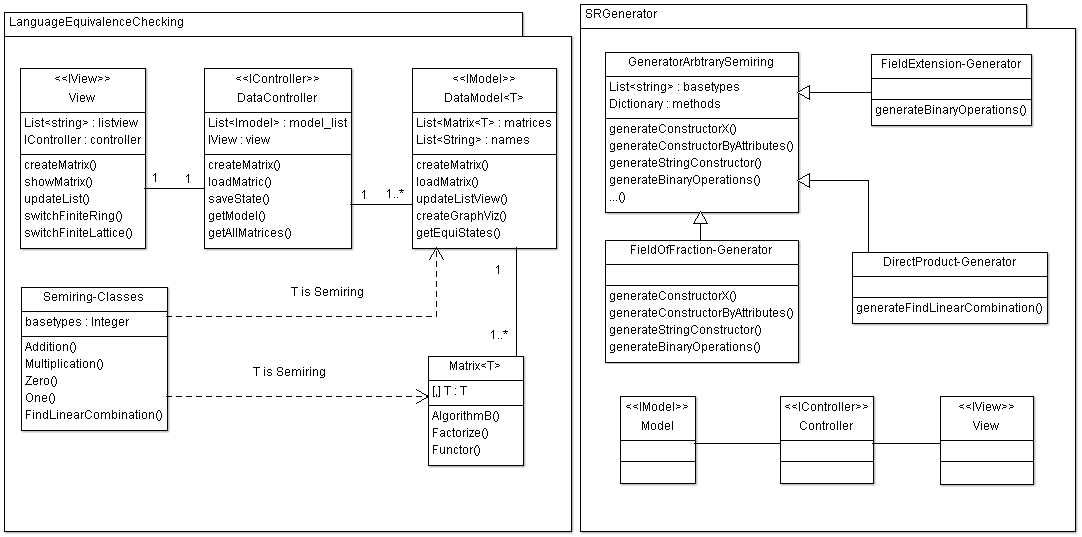


Abbildung 14: Reduziertes UML-Klassendiagramm von Partitionsverfeinerungs-Algorithm-B und SRGenerator

Verschiedene Restklassenringe und endliche distributive Verbände werden durch den Austausch einer statischen Variablen zugänglich gemacht. Im Fall der Restklassenringe wird die *static int modulo* mit dem angegebenen Wert des Anwenders instantiiert. Diese Information wird für Matrizen über Restklassenringen in einer zusätzlichen Information im Namen der Matrix verwaltet. Für endliche distributive Verbände wird diese Metainformation innerhalb des Datenmodells für *FiniteLattices* analog gespeichert. Die vom Anwender generierten endlichen Verbände werden von einer gesonderten Klasse *PreorderModel* verwaltet.
SRGenerator erzeugt, im Unterschied zu der Bereitstellung eines neuen endlichen Verbandes durch Instanziierung einer statischen Variablen, neue C#- Klassen. Innerhalb der GUI von *SRGenerator* gibt es vier verschiedene Eingabemasken. Hinter jeder Eingabemaske steht eine eigene Generatorklasse. Die Vaterklasse ist *GeneratorArbitrarySemiring* (vgl. rechtes UML-Klassendiagramm 14). Diese Klasse beinhaltet verschiedene Methoden, die in semantische und syntaktische Hilfsmethoden unterteilt werden. Semantische Methoden beziehen sich auf die Generierung des Ver-

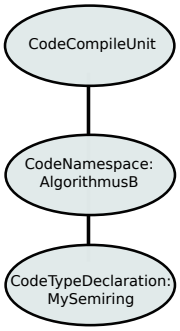
haltens einer Semiring-Klasse, wie z.B. die Methode *generateAdditionComponentwise*. Diese Methode erstellt in Abhängigkeit der ausgewählten Basistypen (vgl. rechtes UML-Klassendiagramm 14) eine komponentenweise Addition. Syntaktische Hilfsmethoden ermöglichen die Generierung von allgemeinen Programmbausteinen, z.B. einer Iterationsschleife. Durch Angabe des Zählers, Startwertes, Grenzwertes und der Anweisung innerhalb der Iterationsschleife wird die komplette Schleife als Teilprogrammdiagramm des CodeDOM erstellt. Die Rückgabe der Methode kann an der erforderlichen Position innerhalb des CodeDOM eingefügt werden. Sämtliche Hilfsmethoden werden an die Kindlassen vererbt und jede Kindklasse besitzt zum Teil ihre eigenen semantischen Generator-Methoden (vgl. rechtes UML-Klassendiagramm 14). Die Konstruktor-Methoden der Vaterklasse werden von *LMonoidGenerator* und *DirectProductGenerator* genutzt. *FieldOFFractionGenerator* benötigt eine eigene Anleitung zur Zusammenstellung der Konstruktoren.
Sämtliche Eingaben des Anwenders werden ebenfalls anhand von vier Datenmodellen verwaltet. Die Interaktion zwischen Anwender und Generator basiert auch auf der MVC- Architektur.

Konzept: Generate-Compile-Build Der Anwender wechselt durch einen Button in *SRGenerator*. Sämtliche Verbände und Matrizen, die vom Anwender in seiner Sitzung erstellt worden sind, werden in einer Autosave-Datei gespeichert. Das Schließen des Programms ist nötig, da eine neue Klasse zunächst kompiliert und durch den *build-Vorgang* dem Programm bekannt gemacht werden muss.
Die Erweiterung der Algorithmus-B-Anwendung erfolgt in vier Schritten. Im ersten Schritt tätig der Anwender seine Eingaben bezüglich des neuen Semirings. Dies wird im Anwendungsabschnitt erläutert. Der zweite Schritt erfolgt nach der Betätigung des *GenerateSemiring*-Buttons und startet die Instanz einer der vier Generator-Klassen, welche die Eingaben in ein CodeDOM überführt. Der dritte Schritt startet die Instanz eines C#- *CodeDOMProvider*. Anhand des Provider Objekts wird das CodeDOM in Programmcode transformiert. Ein *StreamWriter* schreibt dabei den Code in eine .cs-Datei mit dem Namen der Semiring-Klasse. Zum Schluss wird der erzeugte Code testweise kompiliert. Treten keine Compilerfehler auf, wird die neue Semiring-Klasse in die Projektdatei der Algorithmus-B- Datei aufgenommen und durch *MSDN Build* die komplette Anwendung neugebaut.
Eine Alternative bietet *System.Reflection* und das Laden neuer Klassen zur Laufzeit. Eine neue Klasse wird in Form einer Klassenbibliothek generiert (DLL) und durch Anforderung der Assembly anhand des Pfads zur Laufzeit ins Programm geladen. Die Anzahl der DLL-Dateien würde mit jedem Semiring steigen. Das wird durch eine Trennung der Programme vermieden. Außerdem handelt es sich bei der

ersten Anwendung um die Untersuchung der Sprachäquivalenz von Zuständen eines gewichteten Automaten und deren Verwaltung. *SRGenerator* erzeugt dagegen neue C#-Klassen und bietet in einigen Fällen deren vollautomatisierte Generierung. Die Anwendungen müssen daher nicht zwangsläufig parallel laufen. Der Wechsel zwischen den beiden Programmen wird jeweils durch einen Button realisiert und beide Programme kümmern sich automatisch um die Sicherung der Anwendereingaben der aktuellen Sitzung. Das Ziel ist es, den Wechsel für den Anwender möglichst einfach zu gestalten. Für den Anwender macht es nach außen hin keinen Unterschied, zumal ein separater Anzeigebereich bei einer Umsetzung innerhalb der gleichen Ausführungsdatei ebenfalls nötig wäre. Die Trennung begründet sich somit zum einen durch semantische Aspekte und zum anderen aufgrund einer strukturierten Verwaltung der DLL-Dateien.
Im nächsten Abschnitt wird die Umwandlung der Anwendereingaben in ein CodeDOM an einem Beispiel demonstriert.

Einführung in das Programmieren einer AST-Vorlage Dieser Abschnitt beinhaltet eine Einführung in die Generierung eines CodeDOM aus den Anwendereingaben. Zunächst wird eine kurze Beschreibung der Anwendereingaben gegeben und anschließend die Verarbeitung allgemein erklärt. Im Anschluss daran folgt eine detailliertere Veranschaulichung anhand der Generierung einer Additionsoperation.
Ein Anwender hat im voraus u.a. den Basistyp *Double* und für die Addition die Option *Componentwise* ausgewählt. Die Eingaben werden an eine Instanz der vier Generator-Klassen übergeben, welche die Klasse zunächst als leeren Knoten instantiiert. Der Generator ruft anschließend eine Methode zum Generieren der Basistypen in Abhängigkeit von der Anwenderauswahl auf. Nachdem alle Felder erstellt und hinzugefügt worden sind, ruft der Generator die Konstruktor-Methoden auf. Im Anschluss daran werden sämtliche Eingaben des Anwenders bezüglich der binären Operationen abgearbeitet. Die einzelnen Methoden für die Methoden-Generierung folgen dabei immer dem selben Prinzip. Zunächst werden die Anweisungen des Funktionsrumpfes erstellt. Der Anwender hat zwei Optionen die Anweisungen einer Operation zu bestimmen.
In vielen Fällen kann die Addition komponentenweise durchgeführt werden. Dies bedeutet der Anwender wählt die Option *Componentwise* aus. Registriert der Generator diese Angabe beim Abarbeiten der Eingaben, ruft er die entsprechende Hilfsmethode auf. Diese muss zwischen den verschiedenen Basistypen unterscheiden und erzeugt abhängig davon die einzelnen Anweisungen. Die komponentenweise erzeugten Operationen für die einzelnen Basistypen werden zum Schluss als Eingabe für die Rückgabe-Anweisung verwendet. Diese erfolgt unter Verwendung des Basistypen-

Konstruktors.
Die zweite Option ermöglicht die Angabe des Funktionsrumpfes, indem die Programmzeilen vom Anwender selbst eingegeben werden. In dem Fall ruft der Generator eine andere Hilfsmethode auf, die alle Programmzeilen als Block einrückt und anhand des *CodeSnippetStatements* als einzelnes Blatt in den Teilbaum anhängt.
Die erstellten Anweisungen werden dann an eine syntaktische Hilfsmethode weitergereicht, welche die Funktionsdefinition erstellt und die Anweisungen in den Funktionsrumpf einfügt. Zum Schluss wird die Methode, welche in Form eines AST vorliegt, von der Klasse als Teilbaum aufgenommen.
Die Implementierung basiert auf den Klassen des CodeDOM und den Verarbeitungsroutinen der Anwendereingaben innerhalb der einzelnen Generator-Klassen. Der oben beschriebene Ablauf startet mit der Instanziierung einer Generator-Klasse. Dabei werden die Eingaben an den Konstruktor der Generator-Kasse übergeben. Diese enthält eine *CodeCompileUnit targetUnit* und eine *CodeTypeDeclaration targetClass*. Eine *CodeCompileUnit* ist dabei der Container des neuen Programms, welches als Programmdiagramm dargestellt wird. Für die Generierung von Semiringen wird der *Namespace*, der innerhalb der Anwendung von Algorithmus-B verwendet wird, benötigt. Der Generator definiert eine *CodeNamespace*-Instanz mit dem selben Namen. Unter diesem Objekt fügt der Generator die *targetClass* mit dem Namen des neuen Semirings hinzu.




```
namespace AlgorithmB
{
    class MySemiring
    {
    }
}
```

```
namespace AlgorithmB
{
    class MySemiring
    {
     public double mem0;
    }
}
```

Abbildung 15: Links: Wurzel, Namespace-Knoten und Klassen-Knoten. Mitte: Quellcode zu links. Rechts: Generierter C#-Code nach Hinzufügen eines Feldes.

Im Beispiel von Abbildung 15 wählt der Anwender ein Attribut mit dem Basistyp *System.Double* aus. Der Generator verwendet eine *CodeMemberField-Instanz*, um das Feld zu erzeugen. CodeDOM bietet über die Eigenschaften *Attributes*, *Name* und *Type* die Möglichkeit, die Eingaben des Anwenders zu verarbeiten. Dabei werden die Namen automatisch gesetzt und sind fest. Der *targetClass* werden Klassenmember, wie z.B. Basistypen, durch den Ausdruck *targetClass.Members.Add(...)* hinzugefügt.

Vor einem Beispiel der Generierung von einzelnen Anweisungen innerhalb einer Operation am Beispiel der Addition, erfolgt zunächst eine Veranschaulichung wie die Funktionsdefinition erstellt und in den Baum eingefügt wird. Im Fall der Addition wird anfangs ein Objekt *method* vom Typ *CodeMemberMethod* deklariert. Das *CodeMemberMethod*-Objekt enthält die Eigenschaft *Attributes*, welche für die Addition mit *public* und *static* belegt wird. Zusätzlich wird der Name und Rückgabetyp des Operators zugewiesen. Zum Schluss werden die vorab generierten Anweisungen anhand von *method.Statements.Add(...)* in den Teilbaum angehängt.
Für die Generierung der einzelnen Anweisungen ruft der Generator im Fall von *Componentwise* die *generateAdditionComponentwise(List<string> basetypes)* auf. Diese Methode gibt ein Array von *CodeStatements* zurück, das wiederum in den Teilbaum von *method* angehängt wird. Im vorliegenden Beispiel existiert nur ein Basistyp, daher ergibt *return new MyClass(a.mem0 + b.mem0)* die Rückgabeanweisung der Methode und auch die einzige Anweisung.
Für den Fall, dass ein Basistyp als Array deklariert wird, erzeugt *SRGenerator* Addition und Multiplikation angelehnt an die Interpretation von Polynomen als Arrays. Für die Division und Subtraktion gibt es zurzeit keine Automatisierung für Arrays. Die Generierung von bestimmten Polynomen inklusive ihrer Division ist jedoch über eine generische Klasse abgedeckt. Das wird in Abschnitt 3.4.3 genauer erläutert.
Für die Addition von Arrays erstellt die Hilfsmethode Codefragmente, die zunächst überprüfen, welcher Parameter für diesen Basistypen das größere Array besitzt. Dafür wird eine *If-Else*-Anweisung gebaut, die jeweils für jeden Block die entsprechende Iterationsschleife erhält. Innerhalb der Schleifen werden dem größeren Array die Werte des kleineren Arrays an die entsprechenden Positionen im Array hinzuaddiert. Das verrechnete Array repräsentiert dann die Instanz des Basistypen für den Rückgabetypen, der wiederum eine neue Instanz der Klasse selbst ist. Wählt der Anwender mehrere Basistypen aus, werden diese einzeln zusammengerechnet und bilden die Eingabe des Basistyp-Konstruktors. Bei der komponentenweise erstellten Addition kann davon ausgegangen werden, dass die Addition für jeden Basistypen definiert ist. *SRGenerator* erlaubt keine Generierung von Semiring-Klassen ohne eine Definition der Addition.
Der Generator bildet die Eingaben des Anwenders auf die dynamisch erzeugten Konstrukte aus den Objekten des CodeDOM Namespace ab und erzeugt somit einen Objektgraphen, der die gewünschten Eigenschaften und Verhaltensmuster des Semirings repräsentiert.

3.4.2 Allgemeine Implementierungsdetails und Probleme

Im ersten Abschnitt wird das allgemeine Klassendiagramm vorgestellt, damit Struktur und Verhalten einer Semiring-Klasse gegeben sind. Aufbauend darauf werden in den weiteren Abschnitten Probleme und Details behandelt, die bei der Umsetzung aller Semiring-Generator-Klassen auftreten.

Grundgerüst Struktur und Verhalten Jede Semiring-Klasse beinhaltet die notwendigen binären Operationen, um ihr Verhalten zu spezifizieren (vgl rechts Abb. 16). Außerdem enthält jede Klasse die neutralen Elemente bezüglich der Addition und Multiplikation. Zusätzlich benötigt jede Klasse drei Konstruktoren. Zunächst bedarf es eines Konstruktors mit dem Parametertyp *String*, da der Anwender seine Eingaben über eine *String*-basierte GUI-Komponente tätigt. Zur Überprüfung der Eingabe wird zunächst eine separate Methode (*allowedStrings*) aufgerufen, bevor die Instanz eines Matrixelementes erstellt wird. In der Regel kann diese Überprüfung auch direkt im String-Konstruktor geschehen, um eine fehlerhafte Eingabe abfangen zu können. Innerhalb der Generator-Klasse werden diese Implementierungen jedoch getrennt behandelt, um den Quellcode der Generierungs-Vorlagen am Anfang übersichtlicher zu gestalten. Im Abschnitt über die String-Konstruktoren wird auf diesen Aspekt näher eingegangen.

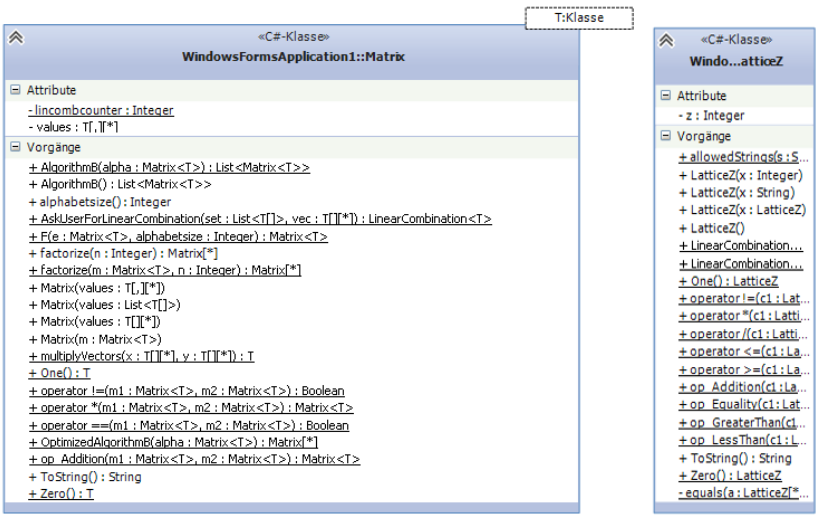


Abbildung 16: Übersicht über die generische Matrix-Klasse und Grundgerüst eines Semirings am Beispiel: Verband der ganzen Zahlen.

Die einzelnen binären Operationen im Ausgangs-Quellcode rufen je nach Umsetzung entweder den Basistypen-Konstruktor oder den Konstruktor mit dem Semiring-Typ selbst als Parameter auf. Dies hängt von der Operation und wie sie umgesetzt wird

ab. Um beide Fälle abzudecken, werden im Standard-Modus alle drei Konstruktoren komponentenweise erstellt.
Bei der Generierung einer Semiring-Klasse erwartet das Programm immer die Angabe von Addition, Multiplikation, Gleichheitsoperator und den neutralen Elementen. Der Anwender kann den Funktionsrumpf selbst angeben oder komponentenweise erstellen lassen. Sofern es sich bei der Generierung z.B. um einen l-Monoid handelt, wird der Anwender zusätzlich aufgefordert, einen Größenvergleichsoperator anzugeben. Der Umfang der erwarteten Eingaben hängt von der Art des Semirings ab. Die Unterschiede zwischen Semiringen sind im Unterkapitel 2.3 aufgeführt.
In dieser Arbeit werden lediglich Semiringe umgesetzt, für die eine Methode zur Lösung von LGS vorhanden ist. Es steht dem Anwender frei, einen Semiring zu definieren, der keine LGS-Methode enthält. In dem Fall wird der Anwender bei der Abarbeitung von Algorithmus B aufgefordert, die Frage, ob eine Linearkombinationen möglich ist, selbst zu beantworten und, falls vorhanden, anzugeben. Die Methode *LinearCombination* wird innerhalb der generischen Matrix ((vgl. links Abb. 16)) abhängig vom Semiring in der *Factorize*-Methode mittels Reflexion aufgerufen.
In den folgenden Abschnitten werden allgemeine Probleme und Lösungen der Generator-Vorlagen vorgestellt.

Redundante CodeDOM-Ausdrücke Viele Referenzausdrücke, z.B. die Parameternamen *a* und *b* der binären Operationen, tauchen in der Erstentwicklung an zahlreichen Stellen auf. Innerhalb der Klasse *Constants* werden neben den für die GUI-Interaktion benötigten konstanten Bezeichnern zur Verwaltung der Eingaben, redundante CodeDOM-Ausdrücke aufgelistet und über das Schlüsselwort *static* für alle Generator-Klassen zugänglich gemacht.

SRGenerator kennt alle Semiring-Klassen Der Semiring-Generator benötigt zur Konstruktion eines Semirings, der auf anderen Semiringen basiert, die Typ-Informationen bezüglich der einzelnen Semiring-Klassen. Diese Informationen ändern sich mit jedem Semiring, der generiert oder verändert wird. Der Aufwand bei der Implementierung, sich um die Verwaltung solcher Informationen separat zu kümmern, kann anhand der *System.Reflection* eingespart werden. Reflexion bietet die Möglichkeit, Assembly-Dateien während der Laufzeit zu laden und Instanzen von Typen zu erstellen. Anhand einer solchen Instanz können Informationen über die entsprechende Semiring-Klasse abgerufen werden.
SRGenerator benötigt Reflexion an zwei Stellen. Zum einen erstellt der Generator eine Liste mit den Typen der Basistypen, die der Anwender auswählt, um die Felder seiner neuen Semiring-Klasse festzulegen. Die Anwenderauswahl liegt aufgrund der

GUI erstmal in Zeichenketten vor. *SRGenerator* kennt den Pfad der Algorithmus-B-Assembly-Datei. Anhand des Namens und der Assembly-Datei kann der Generator die ausgewählten Typen erstellen. Diese Typenliste wird an zahlreichen Stellen im Generator benötigt, um Typen auf Gleichheit zu überprüfen oder an CodeDOM-Konstrukte weiterzureichen.
Ein zweiter Grund ergibt sich durch die Generierung der String-Konstruktoren und einer Eingabeinformations-Methode, die innerhalb der ersten GUI genutzt wird. Der Anwender benötigt bei zusammengesetzten Semiringen eine Übersicht über die primitiven Datentypen, die vom String-Konstruktor erwartet werden. Die Generierung dieser Übersicht wird anhand von Reflexion ermöglicht. Für den String-Konstruktor, der innerhalb dieser Arbeit entwickelt worden ist, kommt SRGenerator ohne eine Übersicht der primitiven Datentypen oder Datentypen, deren String-Konstruktor nur eine zusammenhängende Zeichenkette erwartet, nicht aus. Für diesen und den Abschnitt über die Generierung der String-Konstruktoren werden solche Datentypen als endgültige Basistypen bezeichnet. Für eine Auswahl von Basistypen erstellt *SR-Generator* anhand einer rekursiven Methode eine geordnete Auflistung der zugrunde liegenden endgültigen Basistypen. Die Rekursion startet mit dem ersten ausgewählten Typen und ruft anhand Reflexion dessen Felder ab.
Die Methode *GetField(String, BindingFlags)* der Klasse *Type* gibt ein Objekt zurück, das die Felder enthält, die den angegebenen Anforderungen, den sogenannten *BindingFlags*, entsprechen. Im vorliegenden Fall sind die Felder, die nur mit *private* gekennzeichnet sind und weder konstant noch statisch sind, von Bedeutung. Die privaten Felder einer Semiring-Klasse spezifizieren den Zahlbereich des Semirings. Sowohl die automatisch als auch die manuell erstellten Klassen definieren den Zahlbereich des Semirings durch private Felder, während andere Eigenschaften ebenfalls privat sein können, aber zusätzlich als statisch gekennzeichnet werden. Eine Semiring-Klasse benötigt entweder nur private Felder, um den Bereich der Zahlen einzugrenzen, oder zusätzliche Informationen, die für alle Instanzen einer Klasse gelten und daher mit statisch gekennzeichnet werden. Die Ordnung eines Verbandes innerhalb der Klasse *FiniteLattice* ist statisch und spielt für den String-Konstruktor keine Rolle.
Sobald ein endgültiger Basistyp erreicht wird, fügt die Rekursion diesen der Auflistung hinzu, somit bleiben die endgültigen Typ-Informationen korrekt sortiert.
Die Informationen innerhalb einer zur Laufzeit geladenen Assembly-Datei sind durch das Konzept Generate-Compile-Build immer aktuell. Der nächste Abschnitt erläutert den Nutzen der Auflistung innerhalb der Generierung von String-Konstruktoren und führt ein einfaches Beispiel auf.

Generierte String-Konstruktoren Die Eingabe eines gewichteten Automaten innerhalb der GUI von Algorithmus B erfolgt über Zeichenketten. Jede Semiring-Klasse benötigt daher einen String-Konstruktor. Lässt der Anwender einen Semiring generieren, steht es ihm frei, ob er die Konstruktoren eigenständig implementieren möchte. Die Aufgabe eines Semiring-String-Konstruktors besteht in erster Linie darin, die Eingabezeichenkette nach den Basistypen aufzuteilen. Der Vorgang der Aufteilung folgt einem Muster und kann automatisiert werden.
Zunächst wird allgemein und an einem konkreten Beispiel die Anforderung an den String-Konstruktor erklärt. Im Anschluss daran wird der erste Teilschritt der Verarbeitung innerhalb eines String-Konstruktors beschrieben. Dieser Vorgang wird zusätzlich an einem Beispiel veranschaulicht. In Bezug auf den zweiten Schritt der Verarbeitung wird der Nutzen des ersten Schrittes erläutert.
Die Anforderung an die automatisierte Generierung von String-Konstruktoren beinhaltet die Möglichkeit, sämtliche Kombinationen aus den Semiringen und einigen primitiven Datentypen (double, int, uint) als Basistypen zu verwenden. Zusätzlich können auch Arrays mit Semiringen als Datentyp gebildet werden. Ein generierter String-Konstruktor muss in der Lage sein, die Eingabezeichenkette derart aufzusplitten, dass jedem Basistyp die richtige Teilzeichenkette zugewiesen wird. Zur Veranschaulichung der auftretenden Problematik wird hier das Beispiel der rationalen Zahlen angeführt. Die Eingabe des Anwenders besteht für eine rationale Zahl aus zwei durch ein Kommata getrennten ganzen Zahlen, die den Basistypen mit den Bezeichnern „mem0 “und „mem1 “zugewiesen werden.

```
public class Rational
{
 int mem0;
 int mem1;
 public Rationale (String input)
 {
 // Trennzeichen: ,
 string [] split = input.Split(',');
 mem0 = split[0];
 mem1 = split[1];
 }
 ...
}
```

Für den Fall, dass der Anwender den Semiring der rationalen Zahlen als Basistyp für ein Array verwenden möchte und zusätzlich einen weiteren Basistypen wählt, reicht

die C#-Methode zum Aufsplitten von Zeichenketten nicht aus. Für jeden String-Konstruktor eines Semirings wird daher eine Information in Abhängigkeit von den Basistypen erstellt. Diese Information enthält eine Auflistung der endgültigen Basistypen. Ein endgültiger Datentyp erwartet entweder eine Zeichenkette, die kein Trennzeichen enthält, z.B. Double, Int32 und der Semiring der komplexen Zahlen oder Zeichenketten für Arrays, die mit eckigen Klammern eingeschlossen werden. Diese Information wird innerhalb des String-Konstruktors an die *Split-Methode* der Basisklasse *Parser* weitergereicht. Sämtliche von der Klasse unabhängige Methoden bezüglich des Aufsplittens sind in einer Basisklasse implementiert. Jeder generierte Semiring erbt von dieser Basisklasse, um die Generierung von redundanten Codefragmenten zu vermeiden. Die *Split-Methode* trennt die Eingabezeichenkette nach ihren eigenen Basistypen auf und ermöglicht das Weiterreichen der einzelnen Teilzeichenketten. Im Fall eines Semirings wird die Teilzeichenkette an den Konstruktor übergeben und im Falle eines primitiven Datentyps wird die Durchführung einer Zuweisung über das entsprechende *Convert.To* umgesetzt.
Die Generierung der spezifischen Eingaben für die *Split-Methode* hat das Ziel, eine Rückgabe zu erstellen, die ein nach Basistypen geordnetes String-Array zurückgibt. Dieses Array enthält die Zeichenketten derart aufgesplittet, dass die Teilzeichenketten für jeden endgültigen Datentypen in der richtigen Reihenfolge sortiert sind. Die Erkennung von Eingaben für Arrays werden mittels Zählen von offenen und schließenden eckigen Klammern durchgeführt. Zusätzlich kennt die Split-Methode die Position jedes Array-Basistyps und der einzelnen endgültigen Basistypen. Die *Split-Methode* erkennt durch Klammern eingeschlossene Eingaben und zieht diese aus der Zeichenkette heraus. Anschließend weist sie diese Teilzeichenketten dem Rückgabe-Array an der entsprechenden Position zu. Sobald alle von eckigen Klammern eingeschlossenen Zeichenketten extrahiert und dem Rückgabe-Array zugewiesen worden sind, wird der Rest des Eingabe-Strings anhand der Kommata aufgesplittet und die Teilstrings werde an die richtigen Positionen des Rückgabe-Arrays für die endgültigen Basistypen eingefügt.
In Abb. 17 ist ein Beispiel für die Kombination aus dem Semiring der rationalen Zahlen und einem Array darüber gegeben. Die Split-Methode kennt die endgültigen Basistypen Int, Int und ein Array vom Typ Rational. Zunächst wird die Zeichenkette $[3, 4, 1, 2]$ extrahiert und an die Position 2 des Rückgabe-Arrays eingefügt. Da keine weiteren Zeichenketten mit eckigen Klammern vorhanden sind, teilt die Split-Methode die restlichen Zeichen $3, 4$ anhand des Komma-Trennzeichens in 3 und 4 auf. Diese Aufteilungen werden in das Rückgabe-Array einsortiert. Die Weiterverarbeitung erfolgt anschließend durch Weitergabe der entsprechenden Array-Einträge an die Konstruktoren oder endgültigen Basistypen. Die Verarbeitung innerhalb von

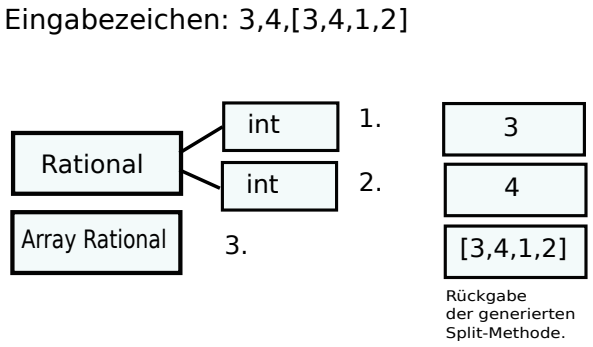


Abbildung 17: Verarbeitung der generierten Split-Methoden in Abhängigkeit der Basistypen.

Arrays erfolgt durch eine separate Methode, welche die Aufteilung der Eingabezeichen innerhalb von Arrays durchführt.
Die Information über die Reihenfolge der endgültigen Basistypen dient dem generierten String-Konstruktor dazu, die Einträge in der richtigen Reihenfolge an die entsprechenden Konstruktoren weiterzugeben. Der String-Konstruktor wird mit dem Wissen darüber, welcher Basistyp wie viele Einträge weitergereicht bekommt, konstruiert. Die ersten beiden Felder der Rückgabe im Beispiel von Abb. 17 werden somit an den Konstruktor der rationalen Zahlen weitergereicht.
Der Anwender muss zwar sämtliche Array-Eingaben mit eckigen Klammern umschließen, jedoch entfällt die Kennzeichnung von zusammengesetzten Eingaben, z.B. für die rationalen Zahlen. Durch das Wissen über die Reihenfolge und die Anzahl der endgültigen Basistypen kann die Verarbeitung innerhalb des String-Konstruktors einer Semiring-Klasse automatisiert erstellt werden.
Zusätzlich zu der Generierung des String-Konstruktors wird eine Methode zur Überprüfung der Gültigkeit einer eingegebenen Zeichenkette erstellt. Die Trennung erfolgt aus Gründen der besseren Lesbarkeit innerhalb der aktuellen Generator-Klasse. Die Codefragmente innerhalb der Generierungs-Methoden sind zum Teil redundant, jedoch bläht die Generierung von Codefragmenten zum Abfangen von unzulässigen Eingaben die Methode für die Generierung des String-Konstruktors auf. Eine Überarbeitung der Umsetzung zur Generierung von Ausnahmebehandlungen innerhalb des String-Konstruktors gehört zu den zukünftigen Optimierungsmöglichkeiten. Entschließt sich ein Anwender dazu, den String-Konstruktor eigenständig zu implementieren, wird die Methode zur Überprüfung nicht erstellt. In dem Fall ist es die Aufgabe des Anwenders, sich um die Überprüfung der Eingaben anhand von Aus-

nahmebehandlungen zu kümmern. Die Verarbeitung einer Zeichenkette, die manuell von einem Anwender implementiert wird, ist für den Semiring-Generator unbekannt. Aus Sicherheitsgründen können solche Klassen nicht zur Konstruktion von anderen Klassen verwendet werden, da nicht sichergestellt werden kann, dass der Konstruktor tatsächlich nach Basistypen aufteilt. Die Methode zur Instanziierung einer Matrix und der einzelnen Matrixelemente innerhalb des Programms zur Ausführung des Partitionsverfeinerungs-Algorithmus ist sowohl für einen automatisiert erstellten als auch einen manuell implementierten Konstruktor entworfen.

In den nächsten beiden Abschnitten wird auf die Probleme der Verwaltung von Semiring-Klassen eingegangen.

Verwaltung neuer Semiringe innerhalb des Generators Im Unterkapitel 2.3 werden einzelne Semiringe nach ihren Eigenschaften gegliedert vorgestellt. Quotientenkörper erlauben es z.B. nur, Polynome zu verwenden, welche über Körpern definiert sind, um das Kürzen zu ermöglichen. Das direkte Produkt gestattet es dem Anwender beliebige Semiringe aus den vorhandenen auszuwählen und einen neuen zu bauen. Die Auswahl von Semiringen ist eine sich ständig ändernde Information. Der Anwender kann Semiringe hinzufügen oder auch löschen. Auswahl und Einschränkungen müssen für den Generator zugänglich sein. Dies geschieht durch Verwaltungsdateien, welche direkt im Anwendungsordner selbst liegen und sich mit jeder Generierung eines neuen Semiringes ändern. Erstellt der Anwender z.B. über die Anwendermaske für l-Monoide einen neuen distributiven Verband unter dem Namen *Mylattice*, wird nur die Verwaltungsdatei für l-Monoide aktualisiert. Anhand dieser Datei steht der neue Semiring nur dem *DirectProduct-Generator* und dem *ArbitrarySemring-Generator* als Basistyp zur Verfügung. Anders verhält es sich, wenn der Anwender über die *ArbitrarySemirings-GUI* -Maske einen Semiring als Körper markiert. Der Name dieses Semirings erscheint in der Verwaltungsdatei für Körper und wird automatisch zusätzlich zu den vorher genannten Generatoren dem *FieldOfFraction-Generator* als Erweiterung für die Polynome zugänglich gemacht. Restklassenringe können vom *DirectProduct-Generator* unter Angabe des Modulo genutzt werden. Alle endlichen distributiven Verbände werden ebenfalls in einer Verwaltungsdatei abgelegt und sind dem Generator bekannt.

Neue Semiringe für Algorithmus B Sämtliche Verwaltungsdateien werden nach jedem *BUILD* des Algorithm-B-Programms ebenfalls in dessen Anwendungsordner kopiert, um alle neuen Semiringe innerhalb der GUI bereitzustellen. Die Verwaltungsdateien werden beim Start des Programms ausgelesen und die Objekttypen

werden über die Methoden der Klassen *Type* und *Activator* erstellt. Für jede Klasse wird ein generisches Datenmodell anhand der Typ-Informationen konstruiert. Zum Schluss werden die gesammelten Datenmodelle zusammen mit der *View* bei der Instanziierung des Controllers überreicht. Alle drei Komponenten bilden gemeinsam die MVC-Architektur.
Die Eigenschaften und Methoden der verschiedenen Semiringe unterscheiden sich auch in der Umsetzung der Generierungs-Routinen. Spezifische Details und Probleme werden im nächsten Unterkapitel basierend auf den mathematischen Grundlagen aus 2.3 beschrieben.

3.4.3 Spezifische Implementierungsdetails

Wie bereits im Grundlagenkapitel zur Verhaltensäquivalenz aufgeführt, hängt die Anwendung von Algorithmus B von der Lösbarkeit linearer Gleichungssysteme ab. Das Gauß-Jordan-Verfahren wird für Körper angewandt und die Heyting-Algebren sind mit dem *Largest-Solution*-Konzept abgedeckt [15]. Beide Verfahren arbeiten mittels Addition und Multiplikation. Die Angabe von Addition, Multiplikation und neutralen Elementen reicht jedoch nicht immer zur Generierung von Körpern, Heyting-Algebren oder anderen Semiringen aus. Die weiteren Abschnitte befassen sich mit einzelnen Varianten von Semiringen, welche teils vollautomatisiert generiert werden können. Die letzten beiden Fälle benötigen keine Code-Generierung und ermöglichen durch Austauschen einer statischen Variablen die Nutzbarkeit von verschiedenen Semiringen.

Das direkte Produkt Ein Element des direkten Produktes $R_1 \times ... \times R_n$ ist ein Tupel mit Elementen aus den einzelnen Semiringen (vgl. 2.3.5). Die binären Operationen des direkten Produkts sind komponentenweise definiert. Das bedeutet, sämtliche komponentenweise definierten Verarbeitungsroutinen (vgl. 3.4.2 und 3.4.1) der Generator-Basisklasse benötigen keine Überladung.
Die Methode einer Semiring-Klasse zur Überprüfung, ob eine Linearkombination für gegebene Vektoren existiert, untersucht, ob ein Vektor b sich gegebenenfalls als Linearkombination von einer gegebenen Menge A von Vektoren darstellen lässt. Der *DirektProductGenerator* beinhaltet eine eigene Methode, um die Lösung von LGS über direkten Produkten zu ermöglichen.
Für das direkte Produkt werden innerhalb der *FindLinearCombination* Programmzeilen generiert, die aus der gegebenen Menge von Vektoren für jeden Semiring des direkten Produkts eine Matrix erstellen, indem die korrespondierenden Einträge aus den Tupeln von A extrahiert werden. Analog werden die einzelnen Vektoren aus den Tupeln des Vektors b gebildet. Der nächste Schritt in der Generierung ist es, den Aufruf der *FindLinearCombination* für jedes spezifische LGS $A_i x_i = b_i$ des Semirings $R_i \in (R_1 \times ... \times R_n)$ einzufügen. Die Methode wird so konstruiert, dass sie eine Linearkombination zurückgibt, wenn alle aufgerufenen Methoden eine gültige Linearkombination zurückgeben.

Quotientenkörper und Körpererweiterungen Die Schwierigkeit, Körper unter Angabe von Addition und Multiplikation vollautomatisiert generieren zu lassen, ergibt sich allgemein durch ein fehlendes Verfahren zur Ermittlung des multiplikativen Inversen. Für die Konstruktion von Quotientenkörpern ist dies durch das Vertauschen von Divisor und Dividend möglich.

Diese Arbeit beinhaltet die Entwicklung eines vollautomatisierten Code-Generators für Quotientenkörper auf der Grundlage von euklidischen Ringen. Erstellt der Anwender innerhalb der Eingabemaske *ArbitrarySemiring* einen Semiring und kennzeichnet diesen als euklidischen Ring, besteht zusätzlich die Möglichkeit, einen Quotientenkörper aus diesem zu bilden.
Zu den euklidischen Ringen gehören sämtliche Polynome über Körpern (vgl. 2.3.2). Für diese steht eine generische Klasse zur Konstruktion von neuen Semiringen zur Verfügung. Sobald vom Anwender eine als Körper gekennzeichnete Semiring-Klasse erstellt wird, steht für ihn zusätzlich der konstruierte Datentyp aus Polynom und neuem Körper zur Weiterverarbeitung bereit. Die binären Operationen für Polynome über Körpern lassen sich einheitlich definieren. Innerhalb einer binären Operation eines Polynoms werden die Operatoren des generischen Typparameters mit *dynamic* gekennzeichnet.
Zu den vier Grundrechenarten enthält ein euklidischer Ring eine Division mit Rest (%). In der Implementierung unterteilt sich die Division in zwei binäre Operationen. Bei der Standard-Division von zwei Elementen wird der ganzzahlige Anteil und bei der Division mit Rest der Rest zurückgegeben. Diese Operationen werden dazu benötigt, den GGT zu ermitteln, der aufgrund der Beschränkung von Zahlenbereichen innerhalb von Rechnern und der Beschaffenheit der Addition bei Quotientenkörpern zum Kürzen benötigt wird.

Eine weitere Möglichkeit, neue Körper zu generieren, ist es, bereits vorhandene Körper um Elemente aus deren Oberkörpern zu erweitern. Generierung von Körpererweiterungen verschiedenen Grades (vgl. 2.3.2) werden innerhalb der Implementierung dieser Arbeit nicht umgesetzt. Die Generator-Klasse für Körpererweiterungen unterstützt alle Körpererweiterungen mit dem Unterkörper der rationalen Zahlen durch Erweiterung um eine irrationalen Zahl $\sqrt{x}$ mit $x \in \mathbb{N}$. Der Anwender erhält einen neuen Körper durch Angabe einer irrationalen Zahl. Die Generierung hätte ebenfalls durch den Austausch einer statischen Variablen umgesetzt werden können. Der Ausblick, die Automatisierung von Körpererweiterungen höheren Grades oder um mehrere Elemente zu erarbeiten, begründet die Entscheidung die Generierung von Körpererweiterungen vom *SRGenerator* aus umzusetzen.

Endliche distributive Verbände Endliche distributive Verbände sind Semiringe, die ein Spezialfall der l-Monoide sind (vgl. 2.3.3). Im Gegensatz zur Generierung von direkten Produkten oder Quotientenkörpern als C#-Klasse müssen für verschiedene endliche distributive Verbände keine separaten Klassen erstellt werden. Wie der Satz von Birkhoff zeigt, können die Elemente eines Verbandes als Mengen von join-irreduziblen Elementen des Verbandes dargestellt werden (vgl. 2.3.3). Die binären Operationen entsprechen der Vereinigung und dem Schnitt. Die neutralen Elemente sind die leere Menge und die Menge aller join-irreduzibler Elemente eines Verbandes.
Die C#-Klasse *FiniteLattice* enthält ein öffentliches und statisches Feld, das den Verband repräsentiert. Der Austausch der Variablen ermöglicht es dem Anwender, verschiedene Verbände zu nutzen. Die Einträge einer Matrix erhalten Elemente aus nur einem Verband, daher wird die Information bezüglich des Verbandes im Namen der Matrix angehängt. Jedes Mal, wenn eine Matrix mit Einträgen aus einem endlichen distributiven Verband geladen wird, setzt das Programm die statische Variable auf den entsprechenden Verband. Analog generiert *SRGenerator* Lade-Programmzeilen innerhalb der Konstruktoren und Operationen, falls endliche distributive Verbände vom Anwender ausgewählt werden, um die statische Variable vor der Ausführung einer binären Operation mit der Auswahl zu synchronisieren.
Der erste Ansatz der Implementierung besteht aus drei Teilen. Der erste Teil ist die oben beschriebene Klasse *FiniteLattice*, die den Verband als statische Variable, die Operationen, neutralen Elemente und das Lösungsverfahren für Heyting-Algebren (vgl. 2.3.3) enthält. Der zweite Teil ist eine Klasse *Set*, die für eine Menge von Zeichenketten steht. Die join-irreduziblen Elemente werden als Zeichenketten eingegeben. Zusätzlich gibt der Anwender die nächstuntergeordneten join-irreduziblen Elemente an. Eine gültige Menge von irreduziblen Elementen repräsentiert ein Element des Verbandes. Eine gültige Menge von Zeichenketten bedeutet, dass für ein gegebenes join-irreduzibles Element das komplette Down-Set in der Menge vorliegt. Als Beispiel dient der Verband aus Abb. 18, der auf zwei verschiedene Weisen vorliegt. Die Elemente d, g, b und c lassen sich nicht durch andere Elemente des Verbandes zusammensetzen und gehören somit zu der Menge der join-irreduziblen Elemente $\{d, g, b, c\}$.
Der Verband, der rechts in Abb. 18 als Mengenverband dargestellt wird, besteht insgesamt aus 8 Elementen. Ausgehend vom Mengenverband ist es ungültig, ein Element $\{g\}$ anzugeben, da $\{g\}$ kein Down-Set ist. Im vorliegenden Beispiel ist $e \leq g$ aber $e \notin \{g\}$.
Im ersten Ansatz der Implementierung von endlichen distributiven Verbänden werden nach Angabe der join-irreduziblen Elemente und der zugehörigen Ordnung alle

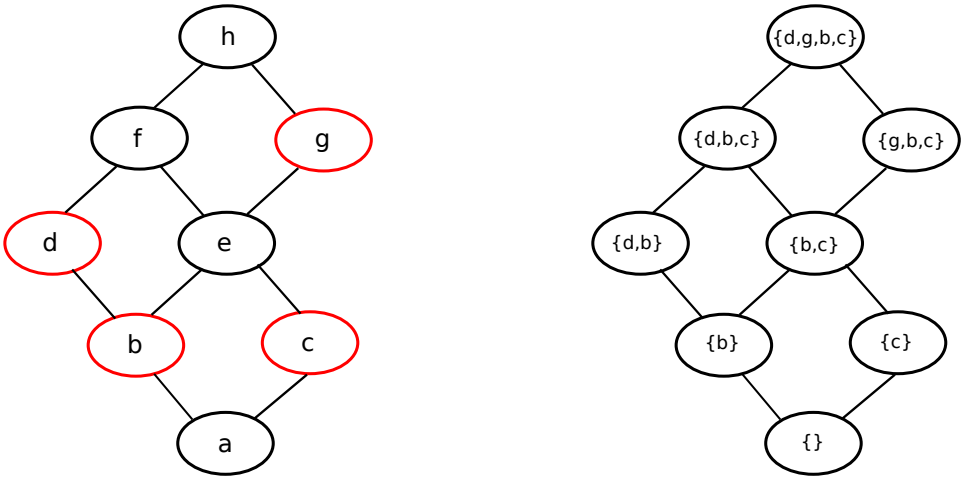


Abbildung 18: Links: Distributiver endlicher Verband mit 8 Elementen und 4 join-irreduziblen Elementen. Rechts: Der gleiche distributive endliche Verband dargestellt als Mengenverband.

Elemente des Verbandes erstellt. Jedes Element ist dabei eine Instanz der Klasse *Set* von Zeichenketten. Die Instanzen der Klasse *Set* werden in einer Datenstruktur, einer generischen Liste, abgelegt und an die Verwaltungsklasse für Verbände als Liste übergeben. Die Liste repräsentiert den Verband, den der Anwender unter Angabe der Ordnung und eines Namens eingibt. Die Generierung der einzelnen Verbandselemente erfolgt durch Vereinigungen. Im ersten Schritt werden die join-irreduziblen Elemente, die aus nur einer zusammenhängenden Zeichenkette bestehen, erstellt. Aus nur einer Zeichenkette bestehende Elemente sind die join-irreduziblen Elemente, die keine untergeordneten join-irreduziblen Elemente besitzen. Im zweiten Schritt werden die Elemente erstellt, die sich aus einem join-irreduziblen Element und dessen Down-Set zusammensetzen. Im nächsten Schritt werden die Vereinigungen der zuletzt erstellten Mengen vorgenommen, sofern eine Menge nicht zum Down-Set einer anderen Menge gehört. Die Mengen, die noch nicht vorhanden sind, werden in den Verband aufgenommen. Dieser Vorgang terminiert, wenn die Vereinigung aller Elemente erreicht wird und alle anderen notwendigen Kombinationen von Vereinigungen durchgeführt worden sind. Im schlimmsten Fall gibt der Anwender einen Verband an, dessen join-irreduzible Elemente minimal sind. Die Anzahl der Verbandselemente wächst für eine derartige Angabe exponentiell mit der Menge der join-irreduziblen Elemente. Die Zahl der Vereinigungen steigt somit rapide in Abhängigkeit von den join-irreduziblen Elementen an. Zusätzlich benötigen die einzelnen Elemente des Verbandes Speicherplatz, da der Verband nur einmal erstellt werden soll. Die Umsetzung hat noch einen weiteren Nachteil, denn die Vereinigung und der Schnitt arbeiten auf Zeichenketten und verwenden zur Überprüfung, ob eine

Zeichenkette bereits vorhanden ist, die Methode *Contains* mit linear wachsendem Suchaufwand bezüglich der Anzahl der Elemente in der Liste. Diese Methode wird ebenfalls für den Vergleichsoperator verwendet.
Die Division von zwei Elementen a, b eines Verbandes wird anhand des Pseudokomplements von b zu a bestimmt. Die Methode für die Berechnung des Pseudokomplements (vgl. 2.3.3) erfolgt anhand der Suche nach bestimmten Elementen. Zunächst wird überprüft, ob der Divisor b gleich dem Dividenden a oder untergeordnet ist. Falls dies der Fall ist, gibt die Division das größte Element des Verbands zurück. Andernfalls wird überprüft, ob der Dividend gleich oder dem Divisor untergeordnet ist. Falls dies zutrifft und der Divisor nicht dem kleinsten Element entspricht, wird der Divisor selbst zurückgegeben. Für den Fall, dass keine der beiden beschriebenen Bedingungen vorliegt, berechnet sich das Pseudokomplement durch Iteration. Alle Elemente x_i des Verbandes, die $b \sqcap (x_i \sqcup a) \leq a$ erfüllen, werden mit a vereinigt. Die Vereinigung all dieser Elemente mit a ergibt das Pseudokomplement, welches das maximale Element ist, das $b \sqcap x_i \leq a$ erfüllt.[55]
Im Rahmen einer weiteren wissenschaftlichen Arbeit von S. Küpper wird unabhängig von dieser Arbeit parallel ein alternatives Konzept implementiert. Die Umsetzung löst den ersten Ansatz aufgrund zweier Vorteile ab. Zunächst wird der Verband nicht in Form von einzelnen Instanzen für jedes Verbandselement erstellt. Somit entfallen Kosten für den Speicherplatz und die Generierung aller Verbandselemente. Weiterhin entfallen die Kosten für Schnitt, Vereinigung und Überprüfung auf Gleichheit bedingt durch die lineare Suche innerhalb von Listen.
Analog zu der ersten Idee existiert eine eigene Klasse *FiniteLattice*, um einzelne Verbandselemente zu repräsentieren. Die Klasse enthält ebenfalls ein statisches Feld, um verschiedene Verbände nutzen zu können. Das Feld ist vom Typ *Preorder* und definiert die Verbandsordnung. Die Klasse *Preorder* besteht aus drei Feldern: ein Feld für die Anzahl der join-irreduziblen Elemente und einem zweiten Feld für deren Bezeichner. Zusätzlich wird für jedes join-irreduzible Element eine Liste mit den untergeordneten join-irreduziblen Elementen angelegt und innerhalb eines Arrays, dem dritten Feld, verwaltet. Dieses Array enthält die Verbandsordnung. Die Indizes der Namensliste und des Arrays stehen für die join-irreduziblen Elemente. Der Bezeichner eines solchen Elements wird bei der Eingabe des Verbandes zunächst festgelegt und einem festen Index zugeordnet.
Die Klasse *FiniteLattice* enthält neben der Verbandsordnung ein Feld für ein Element des Mengenverbands. Ein Element wird durch ein *BitArray* dargestellt. Ein auf eins gesetztes Bit entspricht dabei dem booleschen Wert *true*. Das *BitArray* enthält die Dimension, die innerhalb der aktuellen Verbandsordnung als Anzahl der join-irreduziblen Elemente festgelegt ist. Der Index wird mit einem join-irreduziblen

Element assoziiert. Die Indizes des *BitArrays* stimmen mit denen innerhalb der Verbandsordnung überein. Ein gültiges Element des Mengenverbands liegt dann als *BitArray* vor, wenn die gesetzten Bits ein Down-Set charakterisieren. Die Verwendung des *BitArrays* ermöglicht somit die Überprüfung der Gültigkeit einer Anwendereingabe, ohne dabei alle Verbandselemente zu generieren. Es werden lediglich die Zustände aller Bits und ob die gesetzten Bits ein Down-Set darstellen kontrolliert.
Ein weiterer Vorteil des *BitArrays* ist die Ersetzung von Schnitt und Vereinigung unter Verwendung der *Contains-Methode*. Im Falle des *BitArray* lassen sich die Operationen anhand logischer Operatoren umsetzen. Die Addition, welche der Vereinigung entspricht, wird durch das logische Oder umgesetzt. Die Durchführung der Multiplikation erfolgt anhand des logischen Und.
Die Berechnung des Pseudokomplements durch den oben beschriebenen Suchalgorithmus wird durch logische Operatoren ersetzt. Zunächst wird der Divisor b negiert und anschließend mit dem Dividenden a über das logische Oder verknüpft. Somit werden alle Elemente x_i aus dem Verband berücksichtigt, die $b \sqcap (x_i \sqcup a) \leq a$ erfüllen. Das Ergebnis der Oder-Verknüpfung muss allerdings noch auf das nächst kleinere gültige Element reduziert werden, falls es keiner gültigen Menge entspricht.
Die Berechnung anhand von logischen Operatoren sind effizienter als Operationen, die auf linearer Suche aufbauen. Sowohl die Vorteile des Konzepts, Mengenverbände nicht durch die Generierung aller Elemente darzustellen, als auch die Vorteile logischer Operatoren begründen das Ersetzen der ersten Implementierung.

Restklassenringe Restklassenringe unterteilen sich in endliche Körper und Ringe. Sofern es sich bei der Modulo-Zahl um eine Primzahl handelt, erfüllt der endliche Ring die Körperaxiome. Andernfalls beinhaltet der Ring Nullteiler, was wiederum dazu führt, dass nicht jedes Element ein multiplikatives Inverses besitzt. Ohne die Garantie von multiplikativen Inversen werden alternative Verfahren zum Lösen von LGS verwendet (vgl. 2.3.14). Eine Anpassung des Gauß-Verfahrens ermöglicht die Berechnung von LGS der Form $Ax = b \bmod q$, auch wenn q keine Primzahl ist.
Zunächst wird das LGS in Stufenform gebracht. Für den Fall, dass keine Primzahl vorliegt, wird die Gauß-Anpassung aufgerufen. Ziel der Äquivalenzumformungen beim Gauß ist es, entlang der Diagonalen nur Einsen vorzufinden. Das Ergebnis liegt dann auf der rechten Seite des LGS vor. Die Umformungen von LGS über endlichen Ringen, die nicht auf einer Primzahl beruhen, garantieren keine Konstruktion einer solchen Diagonalen. Das LGS wird anhand des Koeffizienten der betreffenden Position in der Diagonalen umgeformt, sodass sich unterhalb der Diagonalen nur Nullen befinden. Nach Umformung in Stufenform berechnet ein auf Rekursion beruhendes Verfahren jeden möglichen Lösungsvektor für jede mögliche Lösung der aktuellen Gleichung. Mehrere Lösungen treten bei Koeffizienten auf, die Nullteiler von $\mathbb{Z}_q$ sind. Jede endgültige Lösung des umgeformten LGS wird auf Gültigkeit bezüglich der Ausgangslage überprüft. Falls das ursprüngliche LGS gelöst wird, gibt die Methode diese Lösung als Linearkombination zurück. Ansonsten wird die Lösung verworfen und der nächste Lösungspfad berechnet und überprüft. Das Verfahren terminiert, wenn eine Lösung gefunden wurde oder alle möglichen Lösungspfade überprüft worden sind und keine Lösung vorhanden ist. Handelt es sich bei den Koeffizienten um einen zu q teilerfremden Wert, gibt es nur eine eindeutige Lösung für die korrespondierende Gleichung. Die Anzahl der möglichen Lösungspfade ist endlich, da die Anzahl der Elemente in $\mathbb{Z}_n$ durch n beschränkt ist.
Für endliche Körper gibt es neben dem Gauß-Verfahren und anderen deterministischen Verfahren (vgl. 2.3.2) noch einen unveröffentlichten Zufallsalgorithmus von P. Raghavendra, der zunächst eine Zufallsmenge der Größe $N = \lceil 145q^2 \ln qn \rceil$ von Vektoren erzeugt, die potenzielle Lösungsvektoren für die unterste Gleichung des Systems sind. Alle Vektoren, die Lösung der aktuellen Gleichung sind, werden herausgefiltert und eine neue Menge der Größe N wird durch Kombination erstellt. Die Kombination basiert auf der Addition von $q+1$ zufälligen Vektoren aus den verbleibenden gültigen Vektoren. Die kombinierte Menge enthält somit immer noch gültige Lösungen für die vorherige Gleichung und erhöht die Wahrscheinlichkeit, ebenfalls Lösungen für die aktuelle Gleichung zu beinhalten. Eine hohe Wahrscheinlichkeit, eine Lösung für das LGS zu finden, ist für $N = \lceil 145q^2 \ln qn \rceil$ für Primkörper nachgewiesen. In dieser Arbeit wird jedoch das Ziel verfolgt, LGS über endlichen Ringen

lösen zu können. Außerdem gibt es bei der Umsetzung Speicherprobleme für zu hohe Modulo-Werte, daher wird das Verfahren nicht weiter untersucht.
In einer wissenschaftlichen Ausarbeitung [35] über die Komplexitätsanalyse von Lösungsverfahren linearer Gleichungssysteme über endlichen Ringen wird u.a. das Hensel-Lifting aufgeführt (vgl. Abschnitt 1). Das Hensel-Lifting setzt voraus, dass die Primfaktorzerlegung $q = \prod_{i=0}^{k} p_i^{e_i}$ bekannt ist.
Zusammengefasst besteht die implementierte Methode zunächst aus der Ermittlung der Primfaktorzerlegung für q (Zeile 2 in Algorithmus 3). Anschließend wird das Hensel-Lifting für jeden einzelnen Primfaktor p_i und dessen Exponenten e_i durchgeführt. Sobald das Hensel-Lifting eine ungültige Lösung zurückgibt, terminiert die Methode (Zeilen 3-12 in Algorithmus 3). Andernfalls wird eine Lösung für das LGS $Ax = b \bmod q$ anhand des chinesischen Restsatzes und den einzelnen Lösungen für die Primfaktoren zusammengesetzt (Zeile 13 in Algorithmus 3).

Algorithm 3: startHenselLifting(Matrix A, Vector b)

```
1  begin
2      factorization ⟵ getPrimeFactorization(modulo);
3      for x ∈ factorization do
4          LinearCombination solution;
5          Permutation per;
6          getSolutionHenselLifting(A,b,...);
7          if !solution.Exists then
8              return solution;
9          else
10             solutions.Add(solution);
11         end
12     end
13     return getSolutionByChineseRemainder(solutions,factorization);
14 end
```

Der erste Schritt in der Implementierung des Hensel-Liftings ist die Umsetzung der Primfaktorzerlegung. Zunächst wird hier ein naiver Algorithmus mit exponentieller Laufzeit eingesetzt. Die Zahl q wird zunächst so lange durch die Primzahl zwei geteilt, bis das Ergebnis der Division nicht mehr durch zwei teilbar ist. Anschließend wird der aktuelle Teiler so lange um eins inkrementiert, bis der nächste Primfaktor erreicht wird. Das Verfahren terminiert, sobald die Division eins ergibt. Der Algorithmus kann, um die Laufzeit zu optimieren, durch das quadratische Sieb oder andere bekannte optimierte Verfahren für größere Zahlen ausgetauscht werden. Die Laufzeiten sind ebenfalls exponentiell und verlaufen für die Verfahren je nach Anzahl der Dezimalstellen unterschiedlich. [56],[57]
Im zweiten Schritt wird das Hensel-Lifting (vgl. Algorithmus 5) für die Lösungen

der LGS $Ax = b \bmod p_i^{e_i}$ aufgerufen. Zunächst wird innerhalb des ersten Aufrufs der rekursiven Methode das LGS $Ax = b \bmod p_i$ in Stufenform gebracht (Zeile 6 in Algorithmus 5). Da es sich bei p_i um eine Primzahl handelt, werden hier die Äquivalenzumformungen des Gauß-Verfahrens (vgl. 3) verwendet. Falls nach der Umformung keine Nullzeilen oder ungültigen Zeilen vorliegen, wird im nächsten Schritt die Lösung anhand des Gauß-Verfahrens berechnet (Zeilen 7-13 in Algorithmus 5). Anschließend wird h um eins erhöht, um die nächsten Berechnungen durchzuführen. Die Lösung wird mit p^{h-1} multipliziert, dabei entspricht der aktuelle Modulo p^h, und auf die Rückgabe-Variable aufaddiert (Zeilen 14-22 in Algorithmus 5). Für den Fall, dass der zu p_i gehörende Exponent $e_i > 1$ ist und bis dahin eine Lösung existiert, wird zusätzlich ein neues LGS $Ax = b_{neu}$ mit $h + 1$ konstruiert (Zeile 23 in Algorithmus 5). Dieser Schritt wird als „Liften“ bezeichnet (vgl. Algorithmus 4) und verwendet, um eine Lösung für das LGS $Ax = b \bmod p_i^{h+1}$ zu ermitteln. Das

Algorithm 4: NextLiftingStep(Matrix A, Vector b,Matrix set, Vector vec,int h, LinearCombination lc)

```
1  begin
2  |    Modulo.modulo = (int)Math.Pow(p, h +1);
3  |    Modulo[] Ax = MultiplyCopyList(A, lc.Multiplicands);
4  |    Modulo[] xi = DeepArrayCopy(b);
5  |    SubtractArray(xi, Ax);
6  |    int mod = (int)Math.Pow(p, h);
7  |    Divide(xi,mod);
8  |    bool integer =checkIfIntegerAfterDivision;
9  |    if !integer then
10 |    |    lc.Exists=false;
11 |    else
12 |    |    vec=xi;
13 |    end
14 end
```

„Liften“ beginnt mit der Anpassung des Modulo mit p^{h+1}, um die Ausgangsmatrix A mit dem aktuellen Stand des Lösungsvektors zu multiplizieren (Zeilen 2,3 in Algorithmus 4). Anschließend wird der Ausgangsvektor b kopiert, Ax subtrahiert und durch p^h dividiert. Falls die Division einen ganzzahligen Lösungsvektor xi ergibt, wird für den nächsten Schritt des Hensel-Liftings der zu kombinierende Vektor auf das Ergebnis der Division gesetzt (Zeilen 4-13 in Algorithmus 4). Die Matrix A und der Vektor xi bilden zusammen das LGS für den nächsten Aufruf der rekursiven Methode *getSolutionHenselLifting* (Zeile 24 Algorithmus5).
Im Falle, dass nach einer Umformung in Stufenform eine oder mehrere Nullzeilen vorliegen, wird das Verfahren für alle möglichen Kombinationen für die frei wählbaren

Algorithm 5: getSolutionHenselLifting(Matrix A, Vector b, Matrix set,Vector vec,int n, int p, int exp, LinearCombination solution, int h, Permutation p)

```
1  begin
2      if h<exp then
3          if n==0 then
4              Modulo.mod=p;
5              Matrix setT = copy(set); Vector copy = copy(vec);
6              DownwardElimantion(setT,copy);
7              if notSolvable(setT,copy) then
8                  solution.Exists = false; return ;
9              else
10                 n = getNumbersofZeroLines(setT,copy);
11                 if n==0 then
12                     LinearCombination x = Solve(setT,copy);
13                     h++; int mod = (int)Math.Pow(p, h - 1);
14                     solution.Exists= x. Exists;
15                     if h-1>0 then
16                         Modulo.modulo = (int)Math.Pow(p, h);
17                         MultiplyArray(x.Multiplicands, new Modulo(mod));
18                         AddArray(solution.Multiplicands, x.Multiplicands);
19                     else
20                         solution.Multiplicands= x.Multiplicands;
21                     end
22                     if solution.Exists then
23                         NextLiftingStep(A,b,set,vec,h,solution);
24                         getSolutionHenselLifting(A,b,set,vec,0,p,exp,solution,
                            h,per);
25                     else
26                         return;
27                     end
28                 else
29                     getSolutionsHenselLifting(...n...);
30                 end
31             end
32         else
33             while i< number of possible combinations do
34                 combination = getNextCombination;
35                 SaveCurrentState;
36                 UpdateMatrix(set,vec,combination);
37                 getSolutionHenselLifting(A,b,set,vec,0,p,exp solution,h,per);
38                 if(soltion.Exists)return ;
39                 UpdatetoStateBefore;
40                 ++i;
41             end
42         end
43     else
44         return;
45     end
46 end
```

Positionen im Lösungsvektor durchgeführt (Zeile 30 in Algorithmus 5). Die Kombinationen werden ebenfalls rekursiv erstellt und rufen erneut die Hensel-Lifting-Methode auf. Erst wenn alle Kombinationen keine Lösung ergeben, kann ein LGS als nicht lösbar klassifiziert werden (Zeilen 34-41 in Algorithmus 5).
Der dritte und letzte Schritt in der Umsetzung einer Methode zur Lösung von LGS anhand des Hensel-Liftings ist die Implementierung eines Lösungsverfahrens für Systeme von linearen Kongruenzen mit teilerfremden Modulo-Werten (Zeile 13 in 3). Die Lösung linearer Kongruenzen beruht auf dem chinesischen Restsatz und wird anhand des erweiterten euklidischen Algorithmus durchgeführt. Für jeden Modulo-Wert $p_i^{e_i}$ wird das korrespondierende und teilerfremde $M_i = q/p_i^{e_i}$ berechnet. Die Lösung der diophantischen Gleichung $r_i \cdot p_i^{e_i} + M_i \cdot s_i = 1$ beinhaltet die Werte r_i und s_i. Eine Lösung x für das LGS $Ax = b \bmod q$ ergibt sich anschließend durch $\sum_{i=0}^{n} M_i \cdot s_i \cdot x_i$, wobei n die Anzahl von Primfaktoren und x_i ein Lösung für $Ax = b \bmod p_i^{e_i}$ ist.[58]

Innerhalb einer weiteren GUI-Maske (*Arbitrary Semirings*) hat der Anwender die Möglichkeit, unter Angabe sämtlicher Bestandteile einen beliebigen Semiring als Klasse generieren zu lassen. Dies wird im Anwendungsabschnitt 3.5 u.a. an einem Beispiel erläutert.

3.5 GUI und Anwendung

Die Anwendung der Programme unterteilt sich in die Generierung von neuen Semiringen und das Untersuchen von gewichteten Automaten auf Sprachäquivalenz. In den folgenden Abschnitten werden die grafischen Schnittstellen und einige Optionen erläutert. Die Erläuterungen zu den Anwendungen orientieren sich dabei an einem möglichen Szenario.

Beispielszenario Ein Anwender verfolgt das Ziel, drei Automaten auf Sprachäquivalenz zu untersuchen. Der erste Automat, bezeichnet als Automat A1, entnimmt seine Gewichte aus dem Verband der ganzen Zahlen. Der zweite Automat A2 dagegen enthält Gewichte aus dem Restklassenring $\mathbb{Z}_{16}$. Der dritte Automat A3 ist über dem Schiefkörper der Quaternionen definiert. Die Eingabe von Automat A1 als Matrix kann sofort nach dem Programmstart erfolgen, denn der Verband der ganzen Zahlen liegt in der Semiring-Auswahl vor. Die anderen beiden Semiringe sind in der Auswahl nicht vorhanden.

3.5.1 Partitionsverfeinerungs-Programm

Die GUI für die Anwendung des Partitionsverfeinerungs-Algorithmus ermöglicht die Eingabe von Matrizen, die gewichtete Automaten repräsentieren, das Speichern von Matrizen, das Generieren von endlichen Verbänden, die Verwendung von Restklassenringen und die Anzeige eines gewichteten Automaten anhand der *Graph-Viz-lib*. Der Anwender wählt *LatticeZ* aus der Menge der unterstützten Semiringe aus, um Automat A1 einzugeben. Über dem Eingabefeld für die Matrix erscheint eine Information, welche die Reihenfolge der primitiven Datentypen angibt. Diese Angabe benötigt der Anwender, um zu wissen, wie sich ein einzelnes Element aus dem Semiring zusammensetzt. Im Fall des Verbandes der ganzen Zahlen erscheint an dieser Stelle ausschließlich der Integer-Datentyp. Zusätzlich kann der Anwender das Alphabet des Automaten festlegen. Für den Fall, dass der Anwender diese Option auslässt, wird ein provisorisches Alphabet in Abhängigkeit von der Anzahl der Matrixzeilen und Spalten erstellt. Ein gewichteter Automat wird in einer Matrix der Form $(\Sigma \times X + 1) \times X$, wobei A das Alphabet und X die Menge der Zustände ist, eingegeben. Die Anzahl der Zeilen muss daher die Form $|\Sigma| \cdot |X| + 1$ haben. Der Automat A1 erhält über das oberste Eingabefeld der Matrixeingabefläche zusätzlich noch einen Namen, der für die interne Verwaltung im Programm benötigt wird. (vgl. Abb. 19)
Nachdem die Eingaben bezüglich Automat A1 vollständig sind, betätigt der Anwender die Schaltfläche mit der Aufschrift *generate Matrix*. Für den Fall, dass alle

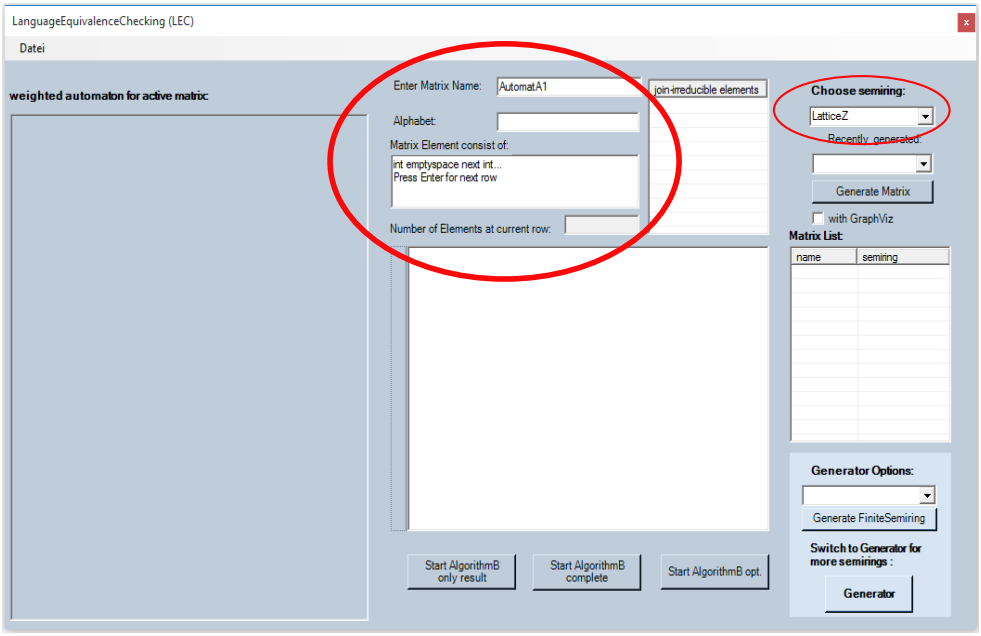


Abbildung 19: Erste Schritte bei der Eingabe eines gewichteten Automaten.

Eingaben korrekt sind, erscheint ein Eintrag in der Auflistung oben rechts mit dem Namen und Semiring des Automaten. Zusätzlich wird links der Automat visuell dargestellt. Der Anwender startet Algorithmus B und erhält nach kurzer Zeit eine Auskunft über die Sprachäquivalenz der Zustände.

Der Anwender fährt mit Automat A2 fort. Der Restklassenring $\mathbb{Z}_{16}$ ist in der Liste der unterstützten Semiringe nicht vorhanden. Die Auswahl bietet unter dem Namen *Modulo* die Möglichkeit, Restklassenringe zu verwenden. Im Standardmodus steht dem Anwender der endliche Ring $\mathbb{Z}_3$ direkt zur Verfügung. Um in einen anderen Restklassenring zu wechseln, klickt der Anwender auf die Schaltfläche *Generate finite semiring* innerhalb des Containers unten rechts. Es erscheint neben den Restklassenringen eine zweite Wahlmöglichkeit, die für die Generierung von endlichen Verbänden zur Verfügung steht (vgl. Abb. 20). Der Anwender wählt die Option *Modulo* aus und wird aufgefordert eine ganze positive Zahl einzugeben. Nachdem der Anwender die Zahl 16 eingegeben und bestätigt hat, steht nun der Restklassenring $\mathbb{Z}_{16}$ bereit. Der Automat A2 wird, wie bereits oben beschrieben, erstellt und der Auflistung oben rechts unter Angabe des Restklassenrings und eines Namens hinzugefügt. Automat A2 wird ausgewählt und Algorithmus B wird gestartet.

Als Letztes benötigt der Anwender die Auskunft bezüglich Automat A3. Der Schief-

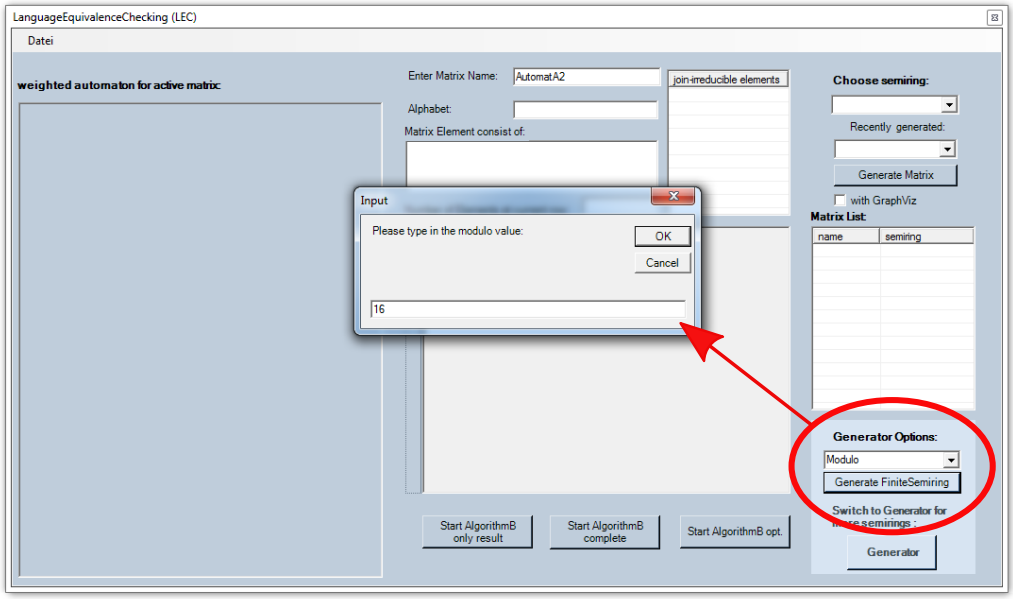


Abbildung 20: Festlegen auf den Restklassenring $\mathbb{Z}_{16}$ über den internen Generator für endliche Semiringe.

körper der Quaternionen ist jedoch nicht in der Auflistung zu finden. Weiterhin gehört der Schiefkörper weder zu den endlichen Verbänden noch zu den Restklassenringen. Der Anwender wechselt über die Schaltfläche *SRGenerator* in die zweite Anwendung. Sämtliche Eingaben der aktuellen Sitzung werden dabei automatisch gespeichert und beim nächsten Programmstart geladen.

3.5.2 SRGenerator

SRGenerator ermöglicht die Generierung von C#-Klassen, die jeweils Eigenschaften und Verhalten eines Semiringes repräsentieren. Die erzeugten C#-Klassen werden dem Anwender für die Untersuchung von gewichteten Automaten auf Sprachäquivalenz zugänglich gemacht, indem die Anwendung des Partitionsverfeinerungs-Algorithmus durch einen Erstellungsprozess um die neuen Klassen erweitert wird.
Nach dem Schließen der ersten Anwendung öffnet sich die GUI des Semiring-Klassen-Generators *SRGenerator*. Die erste Eingabe betrifft den Namen des Semirings. Der Anwender gibt oben links *Quaternionen* ein. Durch Bestätigung der Eingabe erscheint über dem Eingabefeld der aktuell verwendete Klassenname.
Dem Anwender stehen für das weitere Vorgehen fünf Eingabemasken zur Auswahl bereit. Der Semiring der Quaternionen gehört weder zu den direkten Produkten, den Quotientenköpern oder Körpererweiterungen noch zu den l-Monoiden. Der Anwender bleibt daher bei der Starteingabemaske für beliebige Semiringe.

Die Eingabemaske unterteilt sich in fünf Bereiche. Oben links bestimmt der Anwender die benötigten Basisdatentypen. Direkt unterhalb der Basistypen-Auswahl befinden sich die Eingabeoptionen für die Konstruktoren. Im Standardmodus stehen die Optionen der Konstruktoren auf *Componentwise*, das bedeutet, diese werden in Abhängigkeit von den festgelegten Basistypen automatisch erzeugt. Die Reihenfolge der Basistypen bestimmt die Initialisierung innerhalb der Konstruktoren. Der Container unten links ist farbig markiert, um die benötigten binären Operationen und Neutralelemente hervorzuheben. Die Angabe dieser ist die Voraussetzung für die Generierung eines Semirings. Hier stehen dem Anwender zwei Optionen zur Auswahl. Entweder die Operation oder Methode wird komponentenweise generiert oder der Anwender implementiert den Funktionsrumpf selbst.
Im Fall der Quaternionen legt der Anwender insgesamt vier Mal den primitiven Datentyp *double* an. In der Auflistung oben links erscheinen die entsprechenden Zeileneinträge. Eine Zeile besteht dabei aus dem Datentyp und dem Namen des Feldes innerhalb der C#-Klasse. Der Anwender benötigt den Namen, falls er beabsichtigt eine Methode selbst zu definieren. Die Namen werden automatisch generiert, indem der Zeichenkette *mem* der Index des Datentyps zugewiesen wird.
Die Konstruktoren benötigen keine gesonderten Informationen, daher belässt der Anwender es bei den Standardeinstellungen. Die Addition der Quaternionen ist komponentenweise definiert und kann automatisiert erstellt werden. Der Anwender wählt die Option *Componentwise* aus.
Die Implementierungen der Multiplikation und Division erfolgen unter der Option, den Programmcode innerhalb des Funktionsrumpfes eigenständig anzugeben. Hierfür wählt der Anwender zunächst für die Multiplikation die Option *Write Code* aus und die Eingabemaske wird blockiert. Parallel wird der rechte Bereich der GUI aktiviert, der aus einer Texteingabefläche und zwei Schaltflächen besteht. Der Anwender erhält oberhalb dieser Elemente Informationen über die Funktionsdefinition. In der Texteingabefläche gibt er die Programmzeilen ein und bestätigt mit einer der beiden Schaltflächen. Der rechte Bereich wird deaktiviert und die Eingabemaske kann verwendet werden. Analog verfährt der Anwender mit der Division. Für die Methode zur Lösung eines LGS kann der Anwender für Schiefkörper das Gauß-Verfahren auswählen.
Nachdem alle Eingaben vollständig sind, lässt der Anwender die Klasse generieren und wechselt durch das Klicken eines Buttons in die erste Anwendung. Auch hier werden die Eingaben der aktuellen Sitzung gespeichert und das Programm startet mit den Inhalten der letzten Sitzung.
Analog zu den ersten beiden Automaten steht der Semiring nach dem Erstellungspro-

zess zur Verfügung und der Automat A3 kann eingegeben und auf Sprachäquivalenz untersucht werden.

4 Evaluation Partitionsverfeinerung-Algorithmus B und SRGenerator

In der Evaluation werden beide Programme getestet und überarbeitet. Die hier vorgestellte Evaluation der Partitionsverfeinerung bezieht sich auf die Laufzeiten der Lösungsverfahren linearer Gleichungssysteme. Die Beschreibung der Testphasen des Semiring-Generators erläutern den Ablauf der Entwicklung und Überprüfung der generierten Semiringe.

4.1 Partitionsverfeinerung-Algorithmus B

Die Evaluation umfasst Laufzeiterhebungen für verschiedene Semiringe und Dimensionen von Matrizen. Zunächst werden die endlichen Ringe analysiert. In dem Fall werden zusätzlich zu den Variationen von Zustandsanzahl, die Einfluss auf die Dimensionen der Matrizen haben, der Einfluss der Modulo-Werte miteinbezogen. Anschließend folgt eine Erhebung der Laufzeiten von gleich großen Matrizen über dem Verband der ganzen Zahlen. Ziel der Laufzeiterhebung ist es die Abhängigkeit der Laufzeit des Partitionsverfeinerungs-Algorithmus von verschiedenen Lösungsverfahren linearer Gleichungssysteme zu untersuchen.
Die verwendeten Matrizen sind in jedem Fall im gleichen Umfang dünn besetzt und werden zufällig generiert. Ein Hinweis sei an dieser Stelle zu Laufzeitwerten gegeben, die aus der Reihe fallen. Ein Grund für fehlerhafte Messungen zu Beginn der Tabelle können Artefakte sein, die von Visual-Studio verursacht werden. Ungewöhnliche Werte, die innerhalb der Tabelle auftreten, können dadurch erklärt werden, dass eine zufällig erzeugte Matrix auch günstig konzipiert sein kann. Damit ist gemeint, dass die Berechnung des Erzeugendensystems nur wenige Zwischenschritte benötigt.
Die Unterschiede der Laufzeiten in den Tabellen 1 und 2 der Verfahren ergeben sich zum einen durch die Nullteiler und zum anderen durch die Anzahl der Kombinationsmöglichkeiten für auftretende Nullzeilen. Während beim Hensel-Lifting jeweils die Primfaktoren $q = \prod_{i=0}^{k} p_i^{e_i}$ die Anzahl der möglichen Kombinationen für ein LGS mit p_i^n bestimmen, wobei n die Anzahl der Nullzeilen beinhaltet und k die Anzahl der Primfaktoren, gilt bei der Gauß-Adaption für die Anzahl der möglichen Lösungspfade q^n. Erst wenn eine Lösung gefunden wurde oder alle Pfade ohne Erfolg durchprobiert worden sind, kann ein LGS bei beiden Verfahren als lösbar oder nicht lösbar klassifiziert werden. Auch für den Fall, dass keine Nullzeilen auftreten, erhöht sich die Anzahl der zu untersuchenden Lösungspfade bei der Gauß-Anpassung ausgehend von der i. Gleichung $a_{ii} \cdot x_i = b_i$, falls a_{ii} ein Nullteiler von q ist, um die Lösungsmenge der i. Gleichung.

Anzahl Zustände	Anzahl Schritte (lösen LGS)	Laufzeit(ms)
10	54	139
11	65	46
12	90	54
13	104	55
14	119	109
15	119	152
16	135	392
17	152	338
18	170	346
19	209	752
20	230	3738
100	5150	341296
101	5252	358940
102	5355	396202
103	5459	561102

Tabelle 1: Laufzeiten von Algorithmus B angewendet auf gewichtete Automaten über endlichen Ringen. Lösungsverfahren für LGS ist das rekursiv eigenständig umgesetzte Hensel-Lifting (vgl. Algorithmus 5). Größe des Alphabets ist 1 und der Modulo-Wert ist 100.

Anzahl Zustände	Anzahl Schritte (lösen LGS)	Laufzeit(ms)
10	54	70
11	77	5259
12	90	645
13	90	18828129

Tabelle 2: Laufzeiten von Algorithmus B angewendet auf gewichtete Automaten über endlichen Ringen. Lösungsverfahren für LGS ist die eigenständige Anpassung des Gauß-Verfahrens. Größe des Alphabets ist 1 und der Modulo-Wert ist 10.

Unabhängig von der Primfaktorzerlegung betrachtet, hängt die Laufzeit des Partitionsverfeinerungs-Algorithmus beim Hensel-Lifting von der Anzahl der auftretenden Nullzeilen, der Größe der Primfaktoren und der Anzahl der Zustände ab und ist effizienter als das adaptierte Gauß-Verfahren. Die Laufzeit des adaptierten Gauß-Verfahrens hängt zusätzlich noch von den Nullteilern ab. Beide Algorithmen verhalten sich im schlimmsten Fall exponentiell. Beim Hensel-Lifting wird zwar für das Lösen der einzelnen LGS innerhalb der Rekursion das Gauß-Verfahren mit Laufzeit $\mathcal{O}(|Z|^3)$ verwendet, aber in Abhängigkeit der möglichen Lösungswege beim Auftreten von Nullzeilen ist es vorab nicht bekannt, welche Kombination zum nächst gültigen Lifting-Schritt führt. Somit werden im schlimmsten Fall $\sum_{i=0}^{k}(p_i{}^{(|Z|-1)})^{e_i}$ LGS für

das Lösen eines LGS durchlaufen. Verglichen mit der Anzahl der möglichen Lösungspfade $q^{(|Z|-1)}$ beim adaptierten Gauß-Verfahren verhält sich das Hensel-Lifting effizienter.

Anzahl Zustände	Anzahl Schritte (lösen LGS)	Laufzeit(ms)
10	14	42
11	14	3
12	9	3
13	9	4
14	9	4
15	35	21
16	65	49
17	65	53
18	65	57
19	35	30
20	44	89
100	14	94

Tabelle 3: Laufzeiten von Algorithmus B angewendet auf gewichtete Automaten über dem Verband der ganzen Zahlen. Verfahren zur Lösung von LGS basierend auf dem Pseudokomplement (vgl. Satz 1). Größe des Alphabets:1.

In der Tabelle 3 sind die Laufzeiten über dem Verband der ganzen Zahlen aufgelistet und laut Tabelle hat die Anzahl der Zustände keinen enormen Einfluss auf die Laufzeit. Variationen der Alphabetgröße haben auf die Laufzeiten des adaptierten Gauß-Verfahren und Hensel-Liftings keinen großen Einfluss. In der folgenden Tabelle 4 wird das schnelle Wachstum der Laufzeiten des Heyting-Algebra-Verfahrens in Abhängigkeit der Alphabetgröße aufgeführt.

Alphabetgröße	Anzahl Schritte (lösen LGS)	Laufzeit(ms)
1	5	18
2	610	27439
3	...	...

Tabelle 4: Laufzeiten von Algorithmus B angewendet auf gewichtete Automaten über dem Verband der ganzen Zahlen ohne Einschränkung der verwendeten Elemente. Verfahren zur Lösung von LGS basierend auf dem Pseudokomplement (vgl. Satz 1). Anzahl Zustände:10.

In Tabelle 5 werden die Laufzeiten in Abhängigkeit von verschiedenen Alphabetsgrößen am Beispiel des beschränkten Verbands 18 in Abschnitt 3.4.3 aufgelistet. Auch hier ist ein Anstieg der Laufzeiten zu beobachten. In Anbetracht der Beobachtungen folgt anschließend eine Erhebung über dem Verband der ganzen Zahlen, der diesmal mit Einschränkungen in der Anzahl der verwendeten Elemente betrachtet wird.

Alphabetgröße	Anzahl Schritte (lösen LGS)	Laufzeit(ms)
1	20	5
2	20	5
3	779	111922

Tabelle 5: Laufzeiten von Algorithmus B angewendet auf gewichtete Automaten über dem Verband 18. Verfahren zur Lösung von LGS basierend auf dem Pseudokomplement (vgl. Abschnitt 1). Anzahl Zustände:10.

Alphabetgröße	Anzahl Schritte (lösen LGS)	Anzahl Elemente	Laufzeit(ms)
2	333	10	7719
3	1464	10	1003406
3	1893	11	1043663
3	1237	12	1029922

Tabelle 6: Laufzeiten von Algorithmus B angewendet auf gewichtete Automaten über dem Verband der ganzen Zahlen mit Einschränkung der verwendeten Elemente. Verfahren zur Lösung von LGS basierend auf dem Pseudokomplement (vgl. 1). Anzahl Zustände:10.

Mit steigender Anzahl der Alphabetsgröße a steigt auch sprunghaft die Anzahl der zu überprüfenden Vektoren auf lineare Abhängigkeiten, da diese von der Anzahl der Wörter abhängen. Die Alphabetgröße bestimmt die Anzahl der Wörter, denn diese ergibt sich durch a^l mit l = Länge der Wörter. Dies spielt bei Körpern und Ringen bei der Anzahl der Berechnungsschritte aufgrund der Verfahren keine tragende Rolle. Hinzu kommt der Aufwand der Berechnung eines Erzeugendensystems in Abhängigkeit von der Größe eines Subsemimoduls, das wiederum von den verwendeten Elementen und der Anzahl der Zustände abhängt.

4.2 Testen des Semiring-Generators

Der Semiring-Generator wird in der ersten Testphase während der Entwicklung mit starkem Fokus auf die Korrektheit der Generierung von String-Konstruktoren überprüft. Verschiedene Kombinationen von Basistypen dienen dazu die Funktionalität der generierten Konstruktoren zu untersuchen. In Anbetracht der möglichen Konstellationen von Basistypen, die auch als Array verwendet werden können, werden im Rahmen der Master-Arbeit acht verschiedene Konstellationen (wie *z.B. INT32, Lattice[], Double*) für die Tests verwendet. Die generierten Programme werden anhand des Programms zur Überprüfung der Sprachäquivalenz getestet und auftretende Fehler werden zunächst manuell beseitigt, um anschließend im Semiring-Generator angepasst werden zu können.

Die zweite Testphase umfasst die Generierung der Semiring-Klasse für rationale Zahlen, die dazu dient die Funktionalität des Generierungsprozesses für Quotientenköper zu überprüfen. In diesem Fall wird überprüft, ob die generierten Methoden das gewünschte Verhalten zeigen, indem einzelne Rechenschritte manuell gegengerechnet werden. Auftretende Fehler werden zunächst manuell entfernt. Die Änderungen in der Implementierung werden anschließend in den entsprechenden Generierungs-Methoden angepasst. Zusätzlich wird anhand der Klasse der rationalen Zahlen die Korrektheit der Körpererweiterung auf der Basis irrationaler Wurzeln ganzer Zahlen für einzelne Testfälle überprüft. Die zweite Testphase umfasst ebenfalls eine Überprüfung der generischen Polynom-Klasse, die anhand der komplexen Zahlen getestet und überarbeitet wird. Das Prinzip der Generierung, manueller Überprüfung und Anpassung innerhalb des Semiring-Generators erfolgt ebenfalls für die Generierung über die Maske für beliebige Semiringe stichprobenartig.
Zusammengefasst verfolgen die Tests zwei Ziele. Ein Ziel ist es während der Entwicklung die Syntax der generierten Programme zu überprüfen und die Generierung zu korrigieren. Das zweite Ziel der Testphasen ist es die Korrektheit des Verhaltens der generierten Programme zu analysieren und anzupassen. Der Aufwand der Testphasen ist in Anbetracht der Anzahl der Rechenoperationen wie z.B. bei der Polynomdivision oder komponentenweiser Division hoch und daher im Rahmen der Master-Arbeit nicht erschöpfend durchgeführt worden.

5 Ausblick und Fazit

In dieser Arbeit werden zwei unterschiedliche Themen untersucht. Zum einen wird der Partitionsverfeinerungs-Algorithmus evaluiert und dessen Anwendungsfeld um die endlichen Ringe erweitert und zum anderen wird ein Semiring-Generator entworfen und umgesetzt.

5.1 Fazit und Ausblick Partitionsverfeinerung

Die Implementierung einer grafischen Schnittstelle ermöglicht die Untersuchung von gewichteten Automaten auf Sprachäquivalenz unter der Verwendung von GUI- Standardleistungen wie Speichern, visueller Auflistung verschiedener Automaten und intuitiver Eingabemöglichkeiten. Während der Eingabe eines Automaten erhält der Anwender eine Information bezüglich des Aufbaus eines einzelnen Semiring-Elements. Das Alphabet eines gewichteten Automaten kann angegeben werden und zusätzlich kann der Automat als Graph visualisiert werden. Die grafische Darstellung des Automaten gibt dem Anwender bis zu einer bestimmten Größe zusätzlich eine intuitive Repräsentation von gewichteten Automaten. Der Wechsel zwischen den erstellten Automaten erfolgt über eine Auflistung, sodass der Anwender mit einem Klick den gewünschten Automaten auswählen kann.
Darüber hinaus lassen sich endliche Ringe und Verbände in wenigen Schritten erstellen und stehen direkt zur Verwendung bereit.
Der aktuelle Stand der GUI kann noch ausgebaut werden. Weitere anwendungsbezogene Optionen sind noch offen. Eine Erweiterung der Optionen für die Alphabetdefinition durch nachträgliche Änderung von einzelnen Symbolen würde die Eingabekorrektur für den Anwender vereinfachen. Auch die Umbenennung von Verbandselementen würde zusätzlich die Handhabung des Eingabevorgangs erleichtern.
Neben der Umsetzung einer GUI beinhaltet diese Arbeit die Ausarbeitung von Lösungsverfahren für endliche Ringe. Das Hensel-Lifting basiert auf der Primfaktorzerlegung. Es sind verschiedene Algorithmen für das Faktorisieren von natürlichen Zahlen bekannt. Die Untersuchung und das Bereitstellen von mehreren Verfahren können zu einer Optimierung der Laufzeit beitragen. Zusätzlich können Verfahren basierend auf der Smith-Normalform implementiert und bezüglich ihrer Laufzeiteffizienz untersucht werden.
Das Anwendungsfeld von Partititionsverfeinerungs-Algorithmus-B ist um die endlichen Ringe erweitert worden. Zusätzlich zu einem naiven Verfahren zur Lösung LGS ist ein effizienteres Verfahren implementiert worden. Allgemein lassen sich die aufgeführten Verfahren zur Lösung von LGS durch parallele Programmierung optimieren. Sowohl für das Gauß-Verfahren als auch das Hensel-Lifting sind Laufzeitanalysen auf

der Basis von paralleler Programmierung bekannt (vgl. [35], [25]).
Die Anwendung des Partitionsverfeinerungs-Algorithmus-B für gewichtete Automaten deckt somit Körper, Heyting-Algebren und endliche Ringe vollständig ab. Die Laufzeiten hängen direkt von den Verfahren ab und unterscheiden sich aufgrund der unterschiedlichen binären Operationen. Während das Gauß-Verfahren die Lösung durch Zeilenumformung ermittelt und somit weitere Rechenschritte benötigt, wenn die Anzahl der Zustände ansteigt, hängt bei den Heyting-Algebren die Laufzeit auch von der Größe des Alphabets ab. Die Anzahl der Rechenschritte für endliche Ringe steigt bei der aktuellen Implementierung zusätzlich mit der Anzahl der Nullzeilen eines LGS nach seiner Umformung, da im voraus nicht bekannt ist, welche der möglichen Lösungen zum nächsten gültigen Lifting-LGS führen. An den drei unterschiedlichen Verfahren wird deutlich, dass die Laufzeiten für verschiedene Dimensionen der Eingabematrix im direkten Bezug zu den Lösungsverfahren stehen.

5.2 Fazit und Ausblick SRGenerator

Die Anwendung des Partitionsverfeinerungs-Algorithmus beruht für gewichtete Automaten auf dem Lösen von linearen Gleichungssystemen. Möchte ein Anwender ihm bekannte Verfahren verwenden, steht es ihm frei, eigene C#-Klassen in das Programm zu integrieren. Während der Recherchen zur Generierung von Semiringen sticht eine wissenschaftliche Ausarbeitung [23] inklusive Implementierung eines Semiring-Generators heraus. Der Anwender hat dort u.a die Möglichkeit, einen Semiring durch die Angabe von Addition und Multiplikation erstellen zu lassen. Zusätzlich erwartet das Programm vom Anwender die Angabe eines Konstruktors. Die Besonderheit des *SRGenerators* ist die vollautomatisierte Erstellung von drei Konstruktoren: die Generierung eines Konstruktors, der als Parameter den Typ der Klasse selbst besitzt, einen Konstruktor mit den Typen der Klassenfelder als Parameter und den für die Eingabe unverzichtbaren Konstruktor, der eine Zeichenkette erwartet. Zusätzlich dazu wird *SRGenerator* zugeschnitten auf die Anforderungen des Algorithmus zur Untersuchung von Sprachäquivalenz entwickelt. Dies bedeutet, die grafischen Eingabemasken werden so konzipiert, dass der Anwender zwischen optionalen und notwendigen Eingaben unterscheiden kann. Anders als bei der Umsetzung von FPSolve [23] benötigen die Semiringe zusätzlich neutrale Elemente und Vergleichsoperatoren. Darüber hinaus kann der Anwender sein eigenes Lösungsverfahren implementieren und dies für denselben Semiring austauschen. Die generierten C#-Klassen können adaptiert oder verändert werden. Das Generate-Compile-Build-Konzept ermöglicht einen einfachen Austausch von spezifischen Implementierungen, ohne dass der Anwender sich mit dem gesamten Programm-Design auseinandersetzen muss.

Auch in der Umsetzung von *SRGenerator* können sowohl die Generierung als auch die GUI optimiert werden. Weitere Testphasen und anschließende Anpassungen im Semiring-Generator würden die Bedienbarkeit verbessern. Der aktuelle Stand der GUI beruht auf einer Anforderungsanalyse mit dem Fokus auf der Funktionalität des Programms. Die Benutzeroberfläche kann auf der Grundlage einer umfassenden Anwenderevaluation weiterentwickelt werden.
Im Grundlagenkapitel werden die Körpererweiterungen vorgestellt, die noch weitere Möglichkeiten zur vollautomatisierten Generierung von Körpern beinhalten, z.B. Körpererweiterungen um mehrere Elemente aus einem Oberkörper oder höheren Grades.
Interessant ist auch die Verwendung der $*$-Operation für die Lösung von LGS (vgl. Abschnitt 2.3.1). In Fällen, in denen das additive Inverse bekannt ist, können zukünftig Optionen zur Generierung von Semiring-Klassen zur Verfügung stehen, die LGS nach Umformung in die Fixpunktgleichung durch die $*$-Operation lösen (vgl. [20]).

Literaturverzeichnis

[1] Peter Gummm Thomas Ihringer. *Allgemeine Algebra mit Anhang: Universelle Algebra*. Heldermann Verlag, 2003.

[2] Mehryar Mohri, Fernando Pereira, and Michael Riley. Weighted automata in text and speech processing. In *IN ECAI-96 WORKSHOP*, pages 46–50. John Wiley and Sons, 1996.

[3] Barbara König and Sebastian Küpper. Generic partition refinement algorithms for coalgebras and an instantiation to weighted automata. In *Proc. of TCS '14*, IFIP AICT, pages 311–325. Springer, 2014. LNCS 8705.

[4] Manfred Droste and Dietrich Kuske. Weighted automata. *Automata: from Mathematics to Applications*, 2013. European Mathematical Society,To appear.

[5] H. Peter Gumm. State based systems are coalgebras. In *Cubo - Matematica Educacional 5*, pages 239–262, 2003.

[6] Uwe Schöning. *Theoretische Informatik - kurzgefasst*. Spektrum Akademischer Verlag, 4. a. (korrig. nachdruck 2003) edition, 2003.

[7] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA, 2nd edition, 2003.

[8] S. C. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, 1956.

[9] Manfred Droste, Werner Kuich, and Heiko Vogler. *Handbook of Weighted Automata*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[10] Prof. Dr. Xin-Long Zhou. Diskrete Mathematik für Informatiker. unpublished, WS 13-15. Online available `https://www.uni-due.de/mathematik/d_w13_dimainf1.shtml`; Last visit: 2015-12-4-11:21.

[11] Mehryar Mohri. *Handbook of Weighted Automata*, chapter Weighted Automata Algorithms. In [9], 1st edition, 2009.

[12] Marie-Pierre Béal, Sylvain Lombardy, and Jacques Sakarovitch. Conjugacy and equivalence of weighted automata and functional transducers. In Dima Grigoriev, John Harrison, and EdwardA. Hirsch, editors, *Computer Science – Theory and Applications*, volume 3967 of *Lecture Notes in Computer Science*, pages 58–69. Springer Berlin Heidelberg, 2006.

[13] Jiří Adámek, Filippo Bonchi, Barbara König, Mathias Hülsbusch, Stefan Milius, and Alexandra Silva. A coalgebraic perspective on minimization and determinization. In Lars Birkedal, editor, *Proc. Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 7213 of *Lecture Notes Comput. Sci.*, pages 58–73. Springer, 2012.

[14] Sebastian Küpper. Abschlusseigenschaften für graph-sprachen mit anwendungen auf terminierungsanalyse. Master's thesis, Universität Duisburg-Essen, 2012.

[15] Barbara König and Sebastian Küppero. A generic partition refinement algorithm, instantiated to language equivalence checking for weighted automata*. submitted, 2014.

[16] Jiří Adámek and Vera Trnková. Initial algebras and terminal coalgebras in many-sorted sets. *Mathematical Structures in Computer Science*, 21(2):481–509, 2011.

[17] Wikipedia-Grafik. Grafik distributiver endlicher Verband, 2015. Online available `https://de.wikipedia.org/wiki/Distributiver_Verband#/media/File:Hasse_diagram_of_powerset_of_3.svg`; Version: 2004 .7.24 , Last visit: 2015-12-4-11:24.

[18] Jiří Adámek and Václav Koubek. On the greatest fixed point of a set functor. *Theoretical Computer Science*, 150(1):57 – 75, 1995.

[19] Daniel Krob. The equality problem for rational series with multiplicities in the tropical semiring is undecidable. *International Journal of Algebra and Computation*, 04(03):405–425, 1994.

[20] with K.N. Balasubramanya Murthy and Srinivas Aluru. Solving linear systems on linear processor arrays using a *-semiring based algorithm! unpublished, 1999. Online available `http://geomete.com/abdali/publications.html`; Version:2009.5-16 , Last visit: 2015-12-4-11:33.

[21] Wikipedia. Fixpunkt Definition, 2015. Online available `https://de.wikipedia.org/wiki/Fixpunkt_%28Mathematik%29`; Version: 2015 .8.31, Last visit: 2015-12-4-11:24.

[22] Prof. Dr. Dr. h.c. Rolf Rannache. Vorlesung Einführung in die Numerik, 2012. Online available `http://numerik.uni-hd.de/~lehre/SS12/numerik0/`; Version: 2012.10.30, Last visit: 2015-12-4-11:30.

[23] Javier Esparza, Michael Luttenberger, and Maximilian Schlund. Fpsolve: A generic solver for fixpoint equations over semirings. In Markus Holzer and Martin Kutrib, editors, *Implementation and Application of Automata - 19th International Conference, CIAA 2014, Giessen, Germany, July 30 - August 2, 2014. Proceedings*, volume 8587 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2014.

[24] S. K. Abdali and B. D. Saunders. Transitive closure and related semiring properties via eliminants. *j-THEOR-COMP-SCI*, 40(2–3):257–274, 1985.

[25] Girish Sharma, Abhishek Agarwala, and Baidurya Bhattacharya. A fast parallel gauss jordan algorithm for matrix inversion using {CUDA}. *Computers & Structures*, 128:31 – 37, 2013.

[26] Peter Göthnerl. *Elemente der Algebra*. Vieweg+Teubner Verlag, 1997.

[27] Alexander Schmidt. *Einfürung in die algebraische Zahlentheorie*. Springer-Verlag Berlin Heidelberg, 2007.

[28] D. Hachenberger. *Mathematik für Informatiker*. Pearson Studium - IT. Pearson Studium, 2008.

[29] Gerd Fischer. *Lineare Algebra*. Grundkurs Mathematik. Vieweg Verlag, Wiesbaden, 14 edition, 2003. 14. Auflage (ISBN 3-528-03217-0) gefunden auf wikipedia.

[30] Daniel Scholz. Algebra, 2015. Online available `http://www.mehr-davon.de/content/algebra.pdf`; Version: 2014-3-27, Last visit: 2015-12-6-11:43.

[31] Zhou Jinglei and Li Qingguo. The largest solution of linear equation over the complete heyting algebra. *Acta Mathematica Scientia*, 30(3):810 – 818, 2010.

[32] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

[33] Wikipedia-Definition. Restklassenring, 2015. Online available `https://de.wikipedia.org/wiki/Restklassenring`; Version: 2015 .6.18-17:04 , Last visit: 2015-12-4-11:23.

[34] Prof. Dr. A. Pott. Lineare Gleichungssysteme - Teil 2 und Vektorräume - Teil 1, 2015. Online available `http://fma2.math.uni-magdeburg.de/~mathww/laag2010/skript_kap2_teil2.pdf`; No Version Information, Last visit: 2015-12-4-11:27.

[35] V. Arvind and T.C. Vijayaraghavan. The complexity of solving linear equations over a finite ring. In Volker Diekert and Bruno Durand, editors, *STACS 2005*, volume 3404 of *Lecture Notes in Computer Science*, pages 472–484. Springer Berlin Heidelberg, 2005.

[36] A. Das and C.E.V. Madhavan. *Public-key Cryptography: Theory and Practice*. Pearson Education, 2009.

[37] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[38] Mark Giesbrecht. Fast computation of the smith normal form of an integer matrix. In *In Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC '95*, pages 110–118. ACM Press, 1995.

[39] Prof. Dr. B.König. Formale Aspekte der Software-Sicherheit und Kryptographie, SS-2015. Online available `http://www.ti.inf.uni-due.de/fileadmin/public/teaching/ssk/slides/ss2015/folien.pdf`; Version: 2015 .8.10-10:21, Last visit: 2015-12-4-11:25.

[40] Warwick Leeuwen, Jan V., A. R. Meyer, and M. Nival. *Handbook of Theoretical Computer Science: Algorithms and Complexity*. MIT Press, Cambridge, MA, USA, 1990.

[41] J.S. Golan. *Semirings and their Applications*. Springer, 1999.

[42] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[43] Krzysztof Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. PhD thesis, Technical University of Ilmenau, October 1998.

[44] Thomas Stahl, Markus Völter, Sven Efftinge, and Arno Haase. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. Dpunkt.Verlag GmbH, second edition, 2007.

[45] H.J.Sander Bruggink. Towards process mining with graph transformation systems. In Holger Giese and Barbara König, editors, *Graph Transformation*, volume 8571 of *Lecture Notes in Computer Science*, pages 253–268. Springer International Publishing, 2014.

[46] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[47] Andreas Rentschler. Model to Text Transformations Languages, 2015. Online available `https://se2.informatik.uni-wuerzburg.de/mediawiki-se-public/images/5/5b/?C=M;O=D`; Version: 2014-12-4-10:33, Last visit: 2015-12-5.

[48] Michael Jaeger. Compilerbau – eine Einführung. Lecture script of Technische Hochschule Mittelhessen, SS15 2015.

[49] Startego/XT. Chapter 5. syntax definition and parsing, 2015. Online available `http://releases.strategoxt.org/strategoxt-manual/unstable/manual/chunk-chapter/tutorial-parsing.html`; No Version Information, Last visit: 2015-12-5-11:00.

[50] msdn technik-support Dominik Doerner. Grammars. unpublished, 24.09.2015. private email communication.

[51] Manish Jayaswal Vivek Shetty. *Practical .NET for Financial Markets*. Apress, 2006.

[52] Microsoft. Verwenden von CodeDOM, 2015. Online available `https://msdn.microsoft.com/de-de/library/y2k85ax6%28v=vs.110%29.aspx`; No Version Information, Last visit: 2015-12-4-11:20.

[53] Microsoft. Generics in the .net Framework, 2015. Online available `https://msdn.microsoft.com/en-us/library/ms172192%28v=vs.110%29.aspx`; Version: 4.6 and 4.5, Last visit: 2015-12-6-16:20.

[54] Microsoft. Reflektion in the .net Framework, 2015. Online available `https://msdn.microsoft.com/de-de/library/f7ykdhsy%28v=vs.110%29.aspx`; Version: 4.6 and 4.5, Last visit: 2015-12-6-16:20.

[55] Takashi Washio, Ken Satoh, Hideaki Takeda, and Akihiro Inokuchi, editors. *New Frontiers in Artificial Intelligence, JSAI 2006 Conference and Workshops, Tokyo, Japan, June 5-9 2006, Revised Selected Papers*, volume 4384 of *Lecture Notes in Computer Science*. Springer, 2007.

[56] L. Lovász, S. Giese, J. Pelikan, and K. Vesztergombi. *Diskrete Mathematik*. Springer-Lehrbuch. Springer Berlin Heidelberg, 2005.

[57] Peter Hartmann University Hamburg. Faktorisierungsalgorithmen natürlicher Zahlen, 2007.

[58] K.U. Witt. *Algebraische und zahlentheoretische Grundlagen für die Informatik: Gruppen, Ringe, Körper, Primzahltests, Verschlüsselung*. SpringerLink : Bücher. Springer Fachmedien Wiesbaden, 2014.

Eidesstattliche Erklärung

Eidesstattliche Erklärung zur <-Arbeit>

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

$Unterschrift:$ $Ort, Datum:$