Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

Parallelisierung von Graphersetzungssystemen

Diplomarbeit am
Institut für Programmstrukturen und Datenorganisation
Lehrstuhl Prof. Tichy
Prof. Dr. Walter F. Tichy
Fakultät für Informatik
Universität Karlsruhe (TH)

von

Jochen Schimmel

Betreuer:
Prof. Dr. Walter F. Tichy
Dipl.-Inform. Tom Gelhausen
Dipl.-Inform. Christoph Schaefer

Tag der Anmeldung: 5. Juni 2008
Tag der Abgabe: 3. November 2008

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 03.11.2008

Kurzfassung

Aktuelle Graphersetzungssysteme nutzen nur einen Ausführungsfaden. Das Leistungspotential von Mehrkernarchitekturen bleibt ihnen daher verschlossen. In dieser Arbeit werden sowohl das Parallelisierungspotential von Graphersetzungssystemen als auch Ansätze zur Implementierung aufgezeigt werden.

Als Parallelisierungsstrategie wird hierbei, neben weiteren Ansätzen, die Zerlegung von Graphen in Partitionen und die gleichzeitige Ausführung von Such- und Ersetzungsschritten beschrieben.

Zur Validierung wird eine Implementierung der vorgestellten Parallelisierungsansätze für das Graphersetzungssystem GrGen.NET durchgeführt.

Des Weiteren wird mit der Genetik eine Anwendungsdomäne der Graphersetzung vorgestellt, die durch ihren hohen Bedarf an Rechenleistung in besonderem Maße durch eine Parallelisierung profitiert.

Danksagung

An dieser Stelle möchte ich Herrn Professor Tichy für die Unterstützung bei meiner Diplomarbeit und die Zusammenarbeit der letzten Jahre danken. Außerdem gilt mein Dank meinen Betreuern Tom Gelhausen und Christoph Schaefer.

Ich danke Markus Klein für technische Hilfe und Unterstützung bei großen und kleinen Schwierigkeiten sowie Holger Kühner für die Korrektur.

Dem GrGen-Team danke ich für die Beantwortung technischer Fragen, was eine unverzichtbare Hilfe bei dieser Arbeit war. Hervorheben möchte ich hierbei Rubino Geiß, Moritz Kroll sowie Edgar Jakumeit. Für die Unterstützung bei Fragestellungen der Genetik danke ich Evelyn Tichy, Isabel Gehring und Alexander Knoll.

Mein besonderer Dank gilt meinen Eltern für ihre große Unterstützung während des gesamten Studiums. Des Weiteren danke ich Claudia und Bernd Dischinger, die mein Interesse an der Informatik seit meiner Schulzeit gefördert haben. Für Unterstützung, Motivation sowie endlose Geduld und Rücksicht bedanke ich mich bei meiner Lebensgefährtin Nadia Ittemann.

Inhaltsverzeichnis

1. Einleitung

Aktuelle Graphersetzungssysteme nutzen nur einen Ausführungsfaden. Das Leistungspotential von Mehrkernarchitekturen bleibt ihnen daher verschlossen. In dieser Arbeit sollen sowohl das Parallelisierungspotential von Graphersetzungssystemen als auch Ansätze zur Implementierung aufgezeigt werden. Somit lässt sich bei Anwendungen der Graphersetzung auf Mehrkernsystemen eine höhere Leistung erzielen.

Graphersetzung wird heutzutage im Compilerbau und zur Modelltransformation in der Softwareentwicklung genutzt. Aber auch in anderen Gebieten wie der Musik [Wank87] und der Biologie [McNi01] gibt es erste Ansätze zur Verwendung der Graphersetzung. Mit einer steigenden Anzahl an Anwendungsgebieten muss auch die Leistungsfähigkeit der vorhanden Graphersetzungssysteme steigen. Durch die aktuellen Entwicklungen in der Prozessortechnologie sind Graphersetzungswerkzeuge, wie auch viele andere Systeme, gezwungen, in Zukunft Gebrauch von mehreren Ausführungsfäden zu machen, um die volle Leistungsfähigkeit moderner Prozessoren auszuschöpfen. In dieser Arbeit wird aufgezeigt, welche Möglichkeiten zur Parallelisierung die Graphersetzung bietet und wie diese in ein vorhandenes System eingefügt werden kann. Hierzu wird exemplarisch das Graphersetzungssystem GrGen.NET [GrG] parallelisiert.

1.1 Zielsetzung der Arbeit

Um der Forderung nach mehr Rechenleistung nachzukommen, ist eine Unterstützung von Mehrkernarchitekturen bei Softwaresystemen mittlerweile unabdingbar. Bis etwa 2005 wurde eine Steigerung der Prozessorleistung noch hauptsächlich durch einen höheren Prozessortakt realisiert. Hierdurch konnten Anwendungen ohne Änderungen am Programmcode eine höhere Leistung erzielen. Dies hat sich seither geändert: Da eine höhere Taktrate nur durch einen unverhältnismäßig hohen Kühlungsaufwand erzielt werden könnte, gehen die Prozessorhersteller dazu über, mehrere Kerne auf einem Chip unterzubringen. Um diese neue Leistungsquelle zu nutzen, ist es jedoch nötig, dass der Anwendungsentwickler dies berücksichtigt und Code mit mehreren Ausführungsfäden erzeugt, so dass entsprechende Programmteile parallel ausgeführt werden können. Die Anforderungen an den Entwickler werden zusätzlich erhöht, da

in Zukunft mit einer weiterhin steigenden Anzahl von Kernen pro Chip zu rechnen ist, so dass der erzeugte Programmcode nicht nur mit mehreren Anwendungsfäden, sondern auch mit je nach Umgebung variabel vielen Fäden zurechtkommen muss.

Eine Unterstützung von Mehrkernarchitekturen ist nur in einfachen Fällen mit geringem Aufwand zu realisieren. Zunächst muss das Parallelisierungspotential der Anwendung erkannt werden, um anschließend eine geeignete Architektur entwickeln zu können. Muss die Parallelität zusätzlich in ein vorhandenes Software-System eingepflegt werden, erhöht sich der Aufwand zusätzlich.

Bei der Graphersetzung bildet der sogenannte *Arbeitsgraph* den Ausgangspunkt. Dieser Arbeitsgraph wird anhand eines definierten Regelwerks schrittweise verändert. Die Fragestellungen bei der Parallelisierung sind nun, unter welchen Bedingungen Suchvorgänge gleichzeitig durchgeführt werden können und wie Ersetzungsschritte innerhalb des Graphen mit mehreren Ausführungsfäden beschleunigt werden können. Die theoretischen Grundlagen hierzu wurden seit den siebziger Jahren erforscht. Diese liefern jedoch keine Antwort, wie die darin vorgestellten Erkenntnisse in ein Software-System überführt werden können.

Eine Implementierung eines parallelen Graphersetzungssystems wird in dieser Arbeit als Erweiterung des Graphersetzungssystems GrGen.NET geliefert. Um die Leistung der parallelisierten Version bewerten zu können, wurde darüber hinaus ein Benchmark entwickelt. Dieser simuliert die Genexpression, einen fundamentalen Prozess der Genetik.

1.2 Aufbau der Arbeit

In Kapitel 2 werden zunächst Grundlagen aus den Bereichen Graphentheorie, Graphersetzung sowie Mehrkernrechner erläutert.

Kapitel 3 beschreibt den aktuellen Stand der Forschung in Bezug auf die parallele Verarbeitung von Graphtransformationen. Des Weiteren werden Arbeiten aufgezeigt, in denen biologische Vorgänge durch (parallele) Graphsysteme abgebildet werden.

In Kapitel 4 wird beschrieben, welche Schritte nötig sind, um ein Graphersetzungssystem zu parallelisieren. Insbesondere wird hierbei aufgezeigt, inwiefern die in Kapitel 3 aufgezeigten Arbeiten hierzu von Nutzen sein können.

Kapitel 5 beschreibt, wie die in Kapitel 4 entwickelten Ansätze praktisch in das Graphersetzungssystem GrGen.NET eingebaut wurden. Insbesondere wird auf Schwierigkeiten beim Reengineering sowie auf Implementierungsalternativen hingewiesen, so dass dieses Kapitel zusammen mit Kapitel 4 auch als Leitfaden für zukünftige Parallelisierungen von anderen Graphersetzungssystemen herangezogen werden kann.

Kapitel 6 präsentiert einen im Rahmen dieser Arbeit entwickelten Benchmark für Graphersetzungssysteme. Dieser findet seine Anwendungsdomäne im Bereich der Genetik. Daher werden auch grundlegende Zusammenhänge aus diesem Fachbereich vorgestellt. Außerdem sind hier Messergebnisse der parallelisierten Version von GrGen.NET [GrG] zu finden.

Kapitel 7 schließlich fasst die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf mögliche zukünftige Entwicklungen im Bereich der parallelen Graphersetzung und deren Anwendungen.

2. Grundlagen

In diesem Kapitel werden zunächst grundlegende Begriffe der Graphentheorie sowie der Graphersetzung erläutert. Anschließend wird ein Überblick über Mehrkernarchitekturen sowie die Anwendungsentwicklung mit mehreren Ausführungsfäden gegeben. Danach werden einige für die Arbeit relevante Datenstrukturen aufgezeigt. Das Kapitel schließt mit einer Einführung in die Genetik. Diese ist nötig, da in Kapitel 6 ein im Rahmen dieser Arbeit entwickelter Benchmark beschrieben wird, der in diesem Anwendungsfeld angesiedelt ist.

Die Definitionen und Begriffserklärungen im ersten Abschnitt basieren auf [EEPT06] und [Goos00]. Weiterführende Informationen zu Mehrkernrechnern sind in [Tane01] zu finden. Eine genaue Beschreibung der Genetik findet man in [Knip06].

2.1 Graphentheorie

Ein *gerichteter Graph* $G = (E, K)$ besteht aus einer *Eckenmenge*[1] E und einer Relation $K = E \times E$, wobei K als *Kantenmenge* des Graphen bezeichnet wird. Die Elemente $(e, e') \in K$ werden als *Kanten* bezeichnet. Ist nicht nur $(e, e') \in K$, sondern auch (e', e), so wird die Kante als *ungerichtet* bezeichnet. Folgt $(e', e) \in K$ automatisch aus $(e, e') \in K$, so ist die Relation K *symmetrisch* und der Graph wird *ungerichtet* genannt.

Entsprechend der obigen Definition sind Graphen ein praktikables Mittel, um Relationen zwischen Objekten auszudrücken. Typische Beispiele sind Bekanntschaften zwischen Personen oder Straßenverbindungen zwischen Städten. Ungerichtete Kanten könnten in diesem Beispiel Straßen, die in beiden Richtungen befahren werden dürfen, repräsentieren, während gerichtete Kanten Einbahnstraßen symbolisieren. Ein Beispiel ist in Abbildung 2.1 zu sehen:

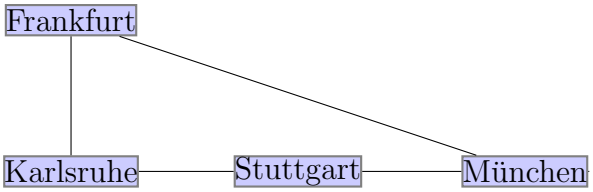

Abbildung 2.1: Einige Städte und Straßen als Graph dargestellt.

Ein Graph ist unabhängig von seiner visuellen Repräsentation: Entscheidend sind allein die Kantenmenge und die definierte Relation. Abbildung 2.2 zeigt daher 3mal denselben Graphen. Dies muss auch bei Graphersetzungssystemen berücksichtigt werden: Diese verwalten Graphen, also Relationen zwischen Objekten, aber keine visuelle Darstellung. Viele Werkzeuge enthalten jedoch Komponenten, die visuelle Darstellungen erzeugen können.

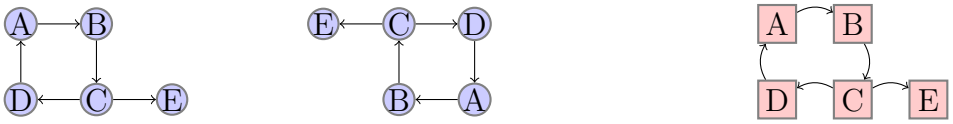

Abbildung 2.2: Drei verschiedene Darstellungen für denselben Graphen.

Unter einem *Teilgraphen* versteht man die Einschränkung der Ecken auf eine Teilmenge $E' \subseteq E$, wobei die Relation $\rho|_{E'} = \{(x, y)|x \in E' \wedge x \rho y\}$ verwendet wird. Dies entspricht den Ecken E' mit allen Kanten, die Ecken $e, e' \in E'$ verbinden.

Die oben gegebene Definition eines Graphen ist für die meisten Zwecke hinreichend. Um komplexere Graphformalismen wie Typen und Attribute zu definieren wird jedoch eine erweiterte Definition benötigt: Ein Graph $G = (E, K, s, t)$ besteht aus einer Eckenmenge E, einer Kantenmenge K, sowie zwei Funktionen $s, t : K \to E$, welche als *Quell-* und *Zielfunktion* bezeichnet werden. Die Quell- und Zielfunktionen ordnen jeder Ecke $e \in E$ ihre Quell- und Zielknoten $q, z \in K$ zu.

Unter Verwendung der zweiten Graphendefinition sind nun auch mehrere Kanten zwischen zwei Knoten möglich. Im Straßenbeispiel würde dies mehreren möglichen

[1] Ecken werden häufig auch als Knoten bezeichnet. Beide Begriffe werden in dieser Arbeit gleichermaßen verwendet.

Straßen zwischen zwei Orten entsprechen. Graphen, die mehrere Kanten zwischen zwei Knoten zulassen, werden auch als *Multigraphen* bezeichnet.

Definition 2.1.0.1 (Graphmorphismus) *Ein Graphmorphismus ist für die Graphen G_1 und G_2 mit $G_i = (E_i, K_i, s_i, t_i)$ für $i = 1, 2$ definiert als eine Funktion $f : G_1 \to G_2, f = (f_E, f_K)$ mit Funktionen $f_E : E_1 \to E_2$ und $f_K : K_1 \to K_2$. f erhält die Quell- und Zielfunktionen von G_i, so dass gilt:*

$$f_E \circ s_1 = s_2 \circ f_E \text{ und } f_K \circ t_1 = t_2 \circ f_K$$

Um die Elemente eines Graphen zu strukturieren kann man auf typisierte Graphen zurückgreifen: Hierbei wird jeder Ecke und jeder Kante ein Typ zugeordnet. Ein Vergleich zur objektorientierten Programmierung liegt nahe: Die Klassen entsprechen den Typen, während einzelne Knoten und Kanten eines Typs ihr Gegenüber in den Objekten finden. Um Typen zu verwenden, wird zunächst ein Typgraph $TG = (K_{tg}, E_{tg}, s_{tg}, v_{tg})$ definiert, der auch als Modell bezeichnet wird. Dabei ist K_{tg} die Knotentypmenge und E_{tg} die Kantentypmenge. Die Funktionen s_{tg} und v_{tg} geben den Typ der Quell- und Zielknoten einer Kante an. Das Tupel $(G, type)$, wobei G ein Graph und $type : G \to TG$ ein Graphmorphismus ist, ist dann ein typisierter Graph. Im Straßenbeispiel könnte man so den Ecken und Kanten Klassen wie beispielsweise Dorf, Stadt, Landstraße oder Autobahn zuordnen.

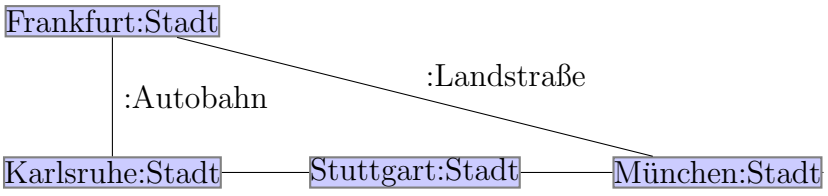

Abbildung 2.3: Ein typisierter Graph. Nach dem Doppelpunkt wird jeweils der Typ des Graphobjekts angegeben.

Bei einem attributierten Graphen werden den Ecken und Kanten ein oder mehrere Attribute zugeordnet. Beispielsweise könnte eine Kante zwischen zwei Städten ein Attribut „Entfernung" erhalten. Attribute werden hierbei jeweils einem bestimmten Typ von Graphelementen zugeordnet. Ein attributierter Graph wird folgendermaßen definiert:

$$G = (V_G, V_D, E_G, E_{NA}, E_{EA}, (s_j, t_j)_{j \in \{G,N,A,EA\}})$$

V_G entspricht den Graphknoten, V_D den Datenknoten. E_G, E_{NA} und E_{EA} sind die Mengen der Graph-, Knotenattribut- und Kantenattributkanten. Durch (s_j, t_j) wird jeder Kantenmenge eine Quell- und Zielfunktion zugeordnet. Auch hier werden die Graphkanten ihren jeweiligen Start- und Zielknoten zugeordnet: $s_G, t_G : E_G \to V_G$. Datenknoten werden den Graphknoten durch die Knotenattributkanten zugeordnet: $s_{NA} : E_{NA} \to V_G, t_{NA} : E_{NA} \to V_D$. Schließlich ordnen die Kantenattributkanten Datenknoten den Kanten des Graphen zu: $s_{EA} : E_{EA} \to E_G, t_{EA} : E_{EA} \to V_D$.

Bei einem typisierten Graphen kann auch eine Hierarchie entsprechend der Vererbung bei der objektorientierten Programmierung aufgebaut werden. So könnten beispielsweise Stadt und Dorf Typen sein, die von Ort abgeleitet sind. Attribute werden hierbei an Untertypen weitergegeben. Ein Untertyp kann seiner Attributmenge weitere Attribute hinzufügen.

2.2 Graphersetzung

Bei der Graphersetzung handelt es sich um die regelbasierte Veränderung eines Graphen. Der Graph, auf dem diese Veränderung stattfindet, wird als *Arbeitsgraph* bezeichnet. Regeln bestehen aus einer linken und einer rechten Seite, wobei die linke Seite ein Muster vorgibt, das im Arbeitsgraphen gefunden werden soll. Wurde die linke Seite im Arbeitsgraphen gefunden, werden entsprechend der rechten Seite Änderungen am Graphen vorgenommen. Ein Beispiel für eine Regel ist in Abbildung 2.4 zu finden:

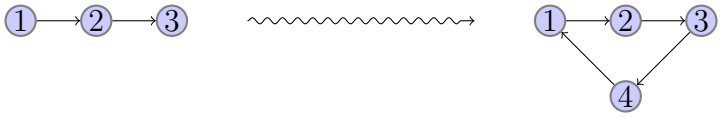

Abbildung 2.4: Ein Beispiel für eine Graphersetzungsregel: Ein neuer Knoten (4) wird an das Muster angefügt.

Im Kontext der Graphersetzung werden Regeln auch als *Produktionen* bezeichnet. Müssen bei einer Regel die Typen der Graphelemente im Muster und im Arbeitsgraphen übereinstimmen, so wird auch von typisierten Regeln (bzw. typisierten Produktionen) gesprochen. Wird ein Muster in einem Arbeitsgraphen gefunden, so wird die Menge der Graphelemente, die das Muster repräsentieren als *Musterinstanz* oder *Match* bezeichnet. Elemente, die Teil der linken Seite sind, aber nicht verändert werden, werden *Klebeelemente* genannt. Die Menge aller Klebeelemente einer Regel bildet deren *Klebegraphen*. Hierauf aufbauend sind folgende Definitionen gegeben:

Definition 2.2.0.2 *Ein* ***Graphersetzungssystem*** $GTS = (P)$ *besteht aus einer Menge von Produktionen* P*. Ein typisiertes Graphersetzungssystem* $GTS = (TG, P)$ *besteht aus einem Typgraphen* TG *und einer Menge typisierter Produktionen* P*. Eine typisierte Graphgrammatik* $GG = (GTS, S)$ *besteht aus einem typisierten Graphersetzungssystem sowie einem typisierten Startgraphen* S *(der Arbeitsgraph).*

Oft werden auch Softwaresysteme, welche die Graphersetzung implementieren, als Graphersetzungssysteme bezeichnet. Aus dem Kontext geht hervor, welche Art von Graphersetzungssystem gemeint ist.

Wird ein Arbeitsgraph A durch die aufeinander folgende (oder gleichzeitige) Anwendung mehrerer Produktionen P in einen Arbeitsgraphen A' überführt, so wird dies auch als *Ableitung* bezeichnet.

Bei den algebraischen Ansätzen zur Definition der Graphersetzung werden zwei Ansätze unterschieden: Der Single-Pushout Approach (SPO) [LöEh91] sowie der Double-Pushout Approach (DPO). In Abbildung 2.5 ist eine Regel, ein Arbeitsgraph sowie die Anwendung der Regel auf den Arbeitsgraphen nach DPO-Semantik zu sehen. Es ist offensichtlich, dass die verwaisten Kanten („dangling edges") entfernt werden müssen. Während der DPO-Ansatz die Anwendung von Regeln, bei denen verwaiste Kanten entstehen, verbietet, werden beim SPO-Ansatz verwaiste Kanten gelöscht, um wieder einen gültigen Graphen zu erhalten.

Bei typisierten Graphersetzungen ist zu beachten, dass das Auffinden von Übereinstimmungen mit der linken Seite einer Produktion sich entsprechend der Vererbung

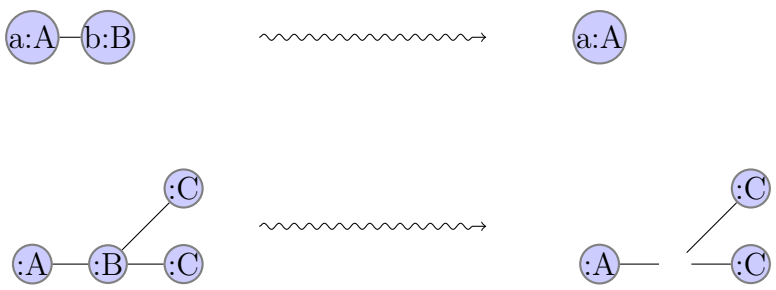

Abbildung 2.5: Oben ist eine Graphersetzungsregel zu sehen. Unten ein Arbeitsgraph, auf den die obige Regel angewendet wird.

verhält: Wird beispielsweise in einer Regel eine Ecke vom Typ Ort gesucht, sind auch Ecken vom Typ Dorf oder Stadt gültige Übereinstimmungen.

Bei der Mustersuche wird zwischen zwei Modi unterschieden: Monomorphe Passung und homomorphe Passung. In Abbildung 2.6 ist auf der linken Seite ein Suchmuster zu sehen, rechts ein Arbeitsgraph, in dem nach dem Muster gesucht wird. Bei einer monomorphen Passung bekommt jeder Knoten einen eindeutigen Bildknoten zugeordnet. In diesem Fall wären die beiden Knoten A_3 und A_4 zusammen mit der zwischen ihnen liegenden Kante eine gültige Instanz des Musters. Bei einer homomorphen Passung hingegen können mehrere Graphelemente des Musters auf dasselbe Element im Arbeitsgraphen abgebildet werden. Somit wäre eine zweite gültige Instanz des Musters durch den Knoten A_4 mit der reflexiven Kante gegeben.

Abbildung 2.6: Links ist das Suchmuster einer Graphersetzungsregel zu sehen, rechts ein Arbeitsgraph, der bei monomorpher Passung eine Instanz des Suchmusters, bei homomorpher Passung zwei Instanzen enthält. Alle Knoten sind vom Typ A, die Nummern dienen der Übersicht.

2.3 Ausführungsfäden und Multiprozessorsysteme

Viele heutzutage eingesetzte Rechner führen nicht ein einzelnes, sondern mehrere Programme gleichzeitig aus. In diesem Kontext werden die laufenden Anwendungen *Prozesse* genannt. Trotzdem besitzen die meisten Rechner nur einen einzigen Prozessor. Um dennoch mehrere Prozesse gleichzeitig ausführen zu können, wird den einzelnen Prozessen reihum Rechenzeit zugewiesen, so dass immer nur ein Prozess abgearbeitet wird, während die anderen pausiert werden. Durch einen schnellen Wechsel der einzelnen Prozesse (ca. 10 bis 100ms) entsteht der Eindruck einer gleichzeitigen Ausführung. Die Entscheidung, welcher Prozess als nächstes Rechenzeit zur Verfügung gestellt bekommt, liegt beim *Scheduler* des Betriebssystems. Moderne Betriebssysteme verwenden üblicherweise *unterbrechendes Multitasking*. Hierbei bekommt jeder Prozess eine bestimmte Rechenzeit zugewiesen. Läuft diese ab, wird der Prozess unterbrochen und der Scheduler bestimmt, welcher Prozess als nächstes weiterverarbeitet wird. Dies führt dazu, dass ein Prozess der auf einem solchen System läuft, keine Annahme darüber machen sollte, wie lange er bis zur nächsten Unterbrechung laufen wird.

Jeder Prozess besitzt einen eigenen Befehlszähler, der vor jedem Prozesswechsel gesichert und beim späteren Wiederherstellen des Prozesses erneut geladen wird. Eine Erweiterung des Prozessmodells stellen die Ausführungsfäden (*Threads*) dar: Ein Prozess kann aus beliebig vielen Ausführungsfäden bestehen, die jeweils einen eigenen Befehlszähler besitzen und somit unabhängig voneinander ausgeführt werden können.

Eine Aufteilung eines Prozesses in mehrere Ausführungsfäden ist insbesondere dann sinnvoll, wenn der Prozess mehrere Aufgaben gleichzeitig abarbeiten muss, bei denen es keine Priorisierung gibt. Ein klassisches Beispiel hierfür ist eine Implementierung eines Webservers, bei der für jede eingehende Verbindungsanfrage ein eigener Ausführungsfaden gestartet wird, der die Anfrage bearbeitet. Da Ausführungsfäden genau wie Prozesse dem Scheduler unterliegen, ist somit für eine faire Verteilung der Rechenzeit gesorgt.

Eine Variante der Ausführungsfäden sind die Ausführungsfasern (*Fibers*): Diese sind für das Betriebssystem unsichtbar und laufen innerhalb eines Ausführungsfadens ab. Die Ablaufplanung geschieht in der Anwendung selbst. Wechsel zwischen Fasern sind somit sehr schnell, da kein Kontextwechsel stattfindet. Der Nachteil der Fasern ist, dass sie nicht parallel auf Mehrkernrechnern ablaufen können. Sie dienen somit als rein strukturelles Werkzeug, ähnlich den Ausführungsfäden auf Einprozessormaschinen. Tabelle 2.1 fasst einige wesentliche Unterschiede zwischen Prozessen, Ausführungsfäden und Fasern zusammen.

Tabelle 2.1: Vergleich von Prozessen, Ausführungsfäden und Fasern

Eigenschaft	Prozess	Ausführungsfaden	Faser
Ablaufplanung	Betriebssystem	Betriebssystem	Anwendung
Befehlszähler	Ja	Ja	Implementierungsabhängig
Erstellen	Aufwändig	Mittel	Billig
Kommunikation	IPC[2]	gemeinsamer Speicher	gemeinsamer Speicher
Mehrkernfähig	Ja	Ja	Nein
Eigener Registersatz	Ja	Ja	Nein

Multiprozessorsysteme zeichnet die Fähigkeit aus, mehr als einen Thread gleichzeitig auszuführen. Dies wird durch zusätzliche Rechenwerke ermöglicht, also die Verknüpfung von mehreren Prozessoren. Somit wird eine echte Parallelität in der Ausführung von Anwendungen ermöglicht. Die jeweiligen Architekturen von Multiprozessorsystemen unterscheiden sich jedoch gravierend. Im Folgenden wird ein Überblick über die verbreitetsten Modelle gegeben.

2.3.1 Shared Memory Multiprozessoren

Shared Memory Multiprozessoren bestehen aus einer Menge von Prozessoren, die sich einen gemeinsamen Arbeitsspeicher teilen, den jede CPU vollständig adressieren kann. Diese Multiprozessorform wird in zwei Untermengen aufgeteilt: Uniform Memory Access (UMA) und Nonuniform Memory Access (NUMA).

Bei einer UMA-Architektur greift jeder Prozessor mit der gleichen Geschwindigkeit auf den gemeinsamen Hauptspeicher zu. Dies geschieht bei Systemen mit einigen

[2]Interprozesskommunikation: Verschiedene Prozesse besitzen getrennte Speicherbereiche und müssen daher über Konstrukte wie gemeinsame Dateien oder vom Betriebssystem bereitgestellte Mittel kommunizieren.

wenigen Prozessoren über einen gemeinsamen Bus, den jeweils nur eine CPU nutzen kann. Bei größerer CPU-Anzahl muss zum Speicherzugriff ein Schaltnetzwerk verwendet werden, um den Flaschenhalseffekt des Buses zu entfernen.

Diese Schaltnetzwerke können jedoch unwirtschaftlich teuer werden, so dass NUMA-Systeme für große CPU-Mengen verwendet werden, die auf den gleich schnellen Zugriff auf den gesamten Hauptspeicher verzichten. Diese Systeme besitzen einen lokalen Speicher pro CPU. Alle lokalen Speicher bilden gemeinsam einen Adressraum, so dass eine CPU auch Zugriff auf den Speicher einer anderen CPU besitzt. Dieser entfernte Zugriff erfolgt langsamer als ein Zugriff auf den lokalen Speicher. Bei solchen Systemen ist somit die Lokalität von Daten von großer Bedeutung, um die Dauer von Speicherzugriffen zu minimieren. Die für diese Arbeit wesentlichen Multiprozessoren fallen in die Klasse der symmetrischen Multiprozessoren (SMP): Dies bedeutet, dass jeder Prozessor in der Lage ist, bei einem Betriebssystemaufruf in der Kernelmodus zu wechseln und den Aufruf selbst auszuführen.

2.3.2 Multicomputer

Multicomputer entstehen durch die Verbindung mehrerer unabhängiger Computer mittels einer Netzwerkschnittstelle, sie werden daher auch als Rechnerbündel oder Cluster bezeichnet. Die Architektur ähnelt sehr stark den NUMA Multiprozessoren: Jeder Rechner besitzt einen eigenen Hauptspeicher, kann jedoch über die Netzwerkschnittstelle auch auf den Speicher anderer Rechner zugreifen.

2.4 Multithreading

In diesem Abschnitt werden grundlegende Begriffe aus der Anwendungsentwicklung mit mehreren Ausführungsfäden erläutert.

2.4.1 Wettlaufbedingungen

Führen mehrere Fäden parallel Operationen aus, so kann keine Vorhersage über die exakte Reihenfolge der Ausführung gemacht werden. Diese hängt vom Scheduler des Betriebssystems sowie von den anderen gleichzeitig ablaufenden Prozessen und weiteren Faktoren ab. Sicher wird lediglich die Reihenfolge der Befehlsausführung innerhalb eines Ausführungsfadens beibehalten. Daher sollten die Anweisungen von Ausführungsfäden immer so angelegt sein, dass das Ergebnis auch bei beliebigem Verzahnen der verschiedenen Anweisungsfäden deterministisch bleibt. Ist dies nicht der Fall, so spricht man von **Wettlaufbedingungen** (engl. *race conditions*).

```
void Inc(int account_id, int amount)
{
  int old_amount = Konten[account_id];
  old_amount += amount;
  Konten[account_id] = old_amount;
}
```

Abbildung 2.7: Erhöhen eines Kontostandes.

Angenommen, der Pseudo-Code in Abbildung 2.7 beschreibt die Ausführung einer Überweisung auf ein Konto bei einer Bank. Das Problem an diesem Code ist die Zwischenspeicherung des Kontostandes in der *old_amount*-Variablen: Wird diese Funktion parallel in zwei Fäden ausgeführt, wobei jeweils eine Überweisung auf dasselbe Konto durchgeführt wird, ist eine Anweisungsfolge wie in Abbildung 2.8 möglich.

```
T1: int old_amount = Konten[account_id];  // old_amount = 1000,-
T2: int old_amount = Konten[account_id]; // old_amount = 1000,-
T1: old_amount += amount;  // old_amount = 1500,-
T2: old_amount += amount;  // old_amount = 1200,-
T1: Konten[account_id] = old_amount;  // Kontostand:  1500,-
T2: Konten[account_id] = old_amount;  // Kontostand:  1200,-
```

Abbildung 2.8: Parallele Ausführung der Inc-Methode. Der ursprüngliche Kontostand beträgt 1000, erhöht wird um 200 bzw. 500.

Bei dieser Anweisungsreihenfolge geht die erste Überweisung verloren. Um solche Situationen zu vermeiden, muss der Zugriff auf die gemeinsame Variable, also den Kontostand, synchronisiert werden.

2.4.2 Synchronisierungsmechanismen

Um Fehler wie im obigen Beispiel zu verhindern, kann der Zugriff auf Variablen synchronisiert werden. Ein bewährtes Konstrukt hierzu ist die *Sperre*: Hierbei handelt es sich um eine Variable, die zwei Zustände annehmen kann: *offen* und *geschlossen*, die zugehören Funktionen werden häufig *lock* und *unlock* genannt.

Möchte ein Faden eine gemeinsame Variable benutzen, so ruft er zunächst *lock* für diese Variable auf. Arbeitet aktuell kein anderer Faden mit der Variablen, so kehrt der Aufruf zurück und die Ausführung wird fortgesetzt. Nach der Operation ruft der Faden *unlock* auf, um die Variable wieder freizugeben. Ruft ein anderer Faden *lock* auf, während die Variable bereits belegt ist, so bleibt der Faden im *lock*-Aufruf gefangen, bis die Variable frei wird. Eine korrekt synchronisierte Version der Überweisungsmethode unter Verwendung eines Mutexes ist in Abbildung 2.9 zu sehen.

```
void Inc(int account_id, int amount)
{
  lock(Konten[account_id]);

  int old_amount = Konten[account_id];
  old_amount += amount;
  Konten[account_id] = old_amount;

  unlock(Konten[account_id]);
}
```

Abbildung 2.9: Kontostand synchronisiert erhöhen.

Neben Sperren gibt es noch weitere Synchronisierungsmechanismen. Diese werden im Verlauf dieser Arbeit bei Bedarf eingeführt.

2.4.3 Körnigkeit der Parallelität

Bei der Parallelisierung einer Anwendung fällt oftmals ein gewisser Synchronisierungsaufwand an. Selten werden Aufgaben ausgeführt, die keine Kommunikation untereinander besitzen und auch keine gemeinsamen Daten verwenden. Das Verhältnis zwischen Rechenaufwand und Synchronisierungssaufwand wird als **Körnigkeit** oder **Granularität** bezeichnet. *Grobkörnige Parallelität* liegt vor, wenn nur an wenigen Stellen synchronisiert werden muss. Muss jedoch häufig auf Anweisungsebene mit Sperren und anderen Konstrukten synchronisiert werden, spricht man von *feinkörniger Parallelität*. Beispielsweise könnte der Zugriff auf eine Datenstruktur synchronisiert werden, indem ein Ausführungsfaden beim Zugriff die komplette Datenstruktur sperrt. Dies wäre eine grobkörnige Synchronisierung. Werden hingegen nur die aktuell benötigten Elemente innerhalb der Datenstruktur gesperrt, so liegt eine feinkörnige Synchronisierung vor.

Für die Parallelisierung eines Graphersetzungssystems nimmt die Körnigkeit der Parallelisierung eine zentrale Stellung ein: Arbeiten mehrere Ausführungsfäden auf einem Graphen, muss der Zugriff auf die Elemente synchronisiert werden. Hierbei stellt sich die Frage, ob jeweils einzelne Graphelemente oder ganze Gruppen von Graphelementen gesperrt werden sollten. Dieses Thema wird in den Abschnitten 4 und 5 aufgegriffen.

2.4.4 Speedup

Der *Speedup* (engl. für Beschleunigung) liefert eine Aussage, wie viel schneller ein paralleles Programm im Vergleich zu einem sequentiellen ist. Ist T_s die sequentielle, T_p die parallele Ausführungszeit, so ist der Speedup S wie folgt definiert:

$$S = \frac{T_s}{T_p}$$

2.5 Genetik

In diesem Abschnitt wird ein Überblick über grundlegende genetische Prozesse gegeben. Diese werden in dem in Kapitel 6 vorgestellten Benchmark simuliert. Dieser Abschnitt behandelt keine Themen der Graphersetzung oder Parallelverarbeitung, sondern dient dem interessierten Leser als Einführung in den später verwendeten Anwendungsfall.

2.5.1 DNA und RNA

Die Erbinformation von Lebewesen wird in einem Desoxyribonukleinsäure-Makromolekül codiert, kurz DNS, nach der englischen Bezeichnung *deoxyribonucleic acid* auch mit DNA abgekürzt. Die DNA besteht aus einer Kette von Bausteinen, die Nucleotide genannt werden. Hierbei können 4 verschiedene Nucleotide auftreten: Adenin, Guanin, Cytosin sowie Thymin. Jedes dieser Nucleotide besteht aus drei Teilen:

- Purin- oder Pyrimidinbase
- Fünf-Kohlenstoff-Zucker
- Phosphat-Rest

Die Purin- bzw. Pyrimidinbase ist charakteristisch für das jeweilige Nucleotid und gibt ihm seinen Namen: Adenin und Guanin sind Purine, Cytosin sowie Thymin Pyrimidine. Der Kohlenstoff-Zucker sowie der Phosphat-Rest bilden die Verbindung zwischen einzelnen Nucleotiden, so dass sich eine fortlaufende Kette ergibt. Dies wird in Bild 2.10 noch einmal veranschaulicht:

DNA-Moleküle bestehen aus zwei Strängen, die über Wasserstoffbrücken zwischen den Nucleotiden verbunden sind. Diese Doppelstränge weisen außerdem die für DNA typische Doppelhelix-Form auf. In Bild 2.11 sieht man, dass die Zucker/Phosphat-Elemente ein Rückgrat für das Molekül bilden, während die Nucleotide im Inneren des Moleküls liegen.

Die nach innen gerichteten Nucleotide sind so angeordnet, dass jeweils ein Purin gegenüber einem Pyrimidin liegt. Des Weiteren sind die beiden Stränge komplementär: Adenin bindet mit Thymin, Guanin mit Cytosin. Somit lässt sich aus einem DNA-Strang die Nucleotid-Sequenz des anderen ableiten. Ein der DNA ähnliches Molekül ist die RNA (ribonucleic acid): Der Grundaufbau ist mit dem der DNA identisch, allerdings tritt hier das Pyrimidin Uracil an die Stelle von Thymin. Außerdem tritt die RNA in der Natur meist als Einzelstrang auf. Üblicherweise sind RNA-Moleküle auch deutlich kürzer als RNA-Moleküle und zerfallen nach kurzer Zeit wieder.

2.5.2 Gene, Aminosäuren und Proteine

Von entscheidender Bedeutung ist die genaue Basenabfolge von Adenin, Guanin, Cytosin und Thymin auf einem DNA-Strang: Diese Codierung bestimmt den Aufbau jeglicher Lebensformen und legt charakteristische Eigenschaften wie beispielsweise Antibiotika-Resistenz bei Bakterien fest. Ein großer Teil dieser Eigenschaften wird durch die verschiedenen Proteine, die ein Lebewesen produzieren kann, bestimmt.

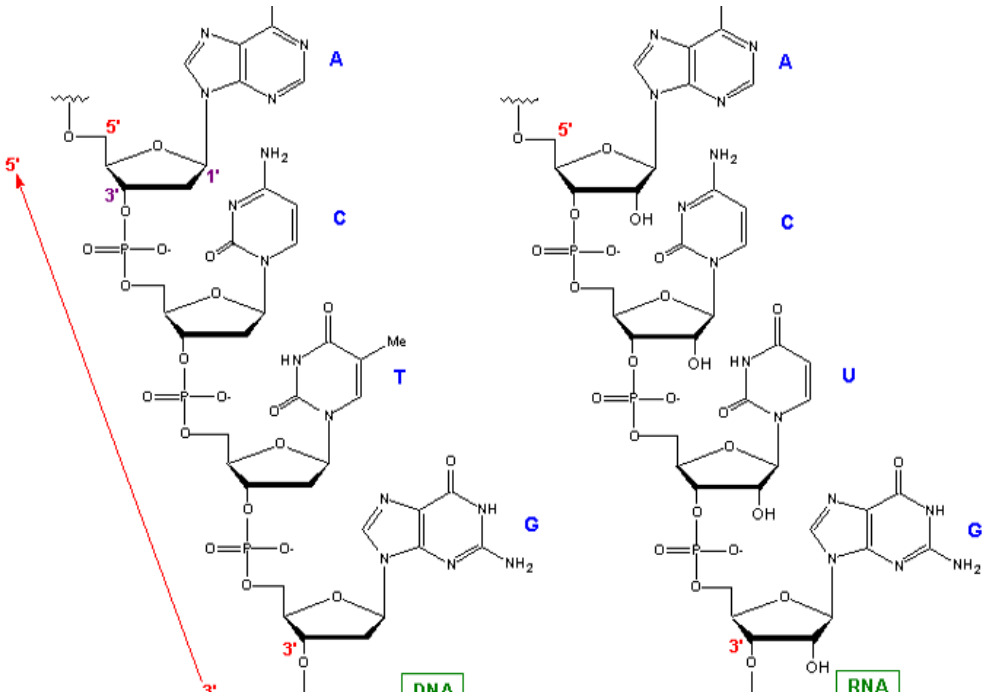

Abbildung 2.10: DNA- und RNA-Nucleotidketten. Die einzelnen Nucleotide sind über Kohlenstoff-Zucker sowie Phosphat-Reste verbunden.

Organismen bestehen zu einem großen Teil aus Proteinen. Diese werden für den Aufbau von Zellen, Organen, aber auch für die Verarbeitung der DNA selbst benötigt: Die Menge der verschiedenen Proteine, die ein Lebewesen produzieren kann, korreliert mit seiner Komplexität: Ein Bakterium benötigt weniger verschiedene Protein-Arten als beispielsweise eine Maus. Proteine bestehen aus einer Kette von Aminosäuren. Insgesamt gibt es 20 verschiedene Aminosäuren. Tabelle 2.2 fasst diese zusammen.

Die genaue Sequenz einer Aminosäuren-Kette ist für ein Protein charakteristisch und wird als *Primärstruktur* des Proteins bezeichnet. Unter der Sekundärstruktur eines Proteins werden Wasserstoffbrückenbindungen bezeichnet, die sich zwischen einzelnen (meist nicht in der Kette aufeinander folgenden) Aminosäuren des Proteins bilden. Die Tertiärstruktur eines Proteins bezeichnet seine Faltung, also die Anordnung der Kette im dreidimensionalen Raum. Diese räumliche Struktur ist für die korrekte Funktion eines Proteins entscheidend: Falsch gefaltete Proteine können ihre Aufgabe in einer Zelle nicht übernehmen. Diese Faltung hat in den letzten Jahren auch in der Informatik für Interesse gesorgt: Da die Vorhersage der Faltung eines Proteins komplex und noch nicht vollständig verstanden ist, helfen Projekte wie Folding@Home[3] bei der Berechnung von Proteinfaltungen.

Die Primärstruktur eines Proteins ist direkt auf der DNA codiert: Die 20 Aminosäuren werden durch Nucleotid-Tripletts codiert, so dass eine Folge von n Nucleotiden ein Protein beschreibt, dass aus $n/3$ Aminosäuren besteht. Durch die 4 verschiedenen Nucleotide lassen sich mit einem Triplett allerdings $4^3 = 64$ Kombinationen bilden, welche den 20 möglichen Aminosäuren gegenüberstehen. Hierbei codieren mehrere verschiedene Tripletts für dieselbe Aminosäure, während drei Kombinatio-

[3] Siehe auch http://folding.stanford.edu/.

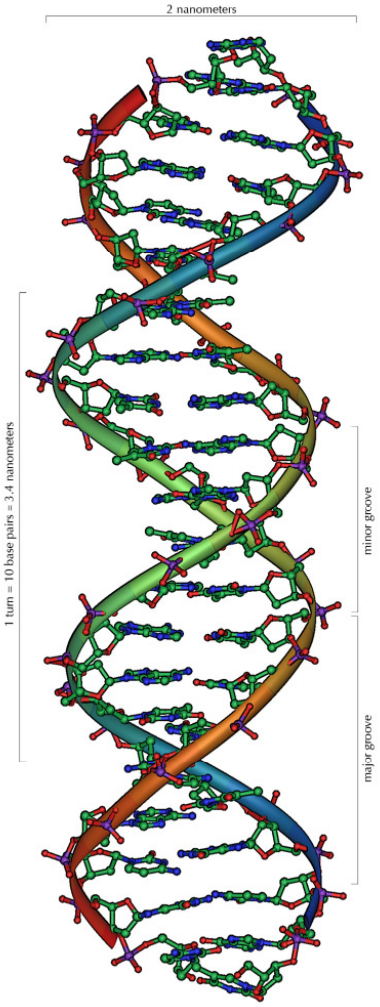

Abbildung 2.11: Die DNA als Doppelhelix.

Tabelle 2.2: Die 20 Aminosäuren

Name	Drei-Buchstaben-Kürzel	Ein-Buchstaben-Kürzel
Glycin	Gly	G
Alanin	Ala	A
Valin	Val	V
Leucin	Leu	L
Isoleucin	Ile	I
Phenylalanin	Phe	F
Tyrosin	Tyr	Y
Tryptophan	Trp	W
Asparaginsäure	Asp	D
Asparagin	Asn	N
Glutaminsäure	Glu	E
Glutamin	Gln	Q
Serin	Ser	S
Threonin	Thr	T
Cystein	Cys	C
Methionin	Met	M
Prolin	Pro	P
Histidin	His	H
Arginin	Arg	R
Lysin	Lys	K

Tabelle 2.3: Codierung von Aminosäuren.

Ala	Arg	Asn	Asp	Cys	Gln	Glu	Gly	His	Ile
GCA	CGA	AAC	GAC	UGC	CAA	GAA	GGA	CAC	AUA
GCC	CGC	AAU	GAU	UGU	CAG	GAG	GGC	CAU	AUC
GCG	CGG						GGG		AUU
GCU	CGU						GGU		
	oder								
	AGA								
	AGG								

Leu	Lys	Met	Phe	Pro	Ser	Thr	Trp	Tyr	Val
CUA	AAA	AUG	UUC	CCA	UCA	ACA	UGG	UAC	GUA
CUC	AAG		UUU	CCC	UCC	ACC		UAU	GUC
CUG				CCG	UCG	ACG			GUG
CUU				CCU	UCU	ACU			GUU
oder					oder				
UUA					AGC				
UUG					AGU				

Tabelle 2.4: Genome einiger Viren, Bakterien und Organismen

Name	Basenpaare (bp)	Zahl der Gene
Simian Virus 40	5243	6
Bakteriophage M13	6407	10
Bakteriophage Lambda	48502	ca. 50
Helicobacter pylori	1667867	1590
Mycobacterium tuberculosis	4411529	3924
Escherichia coli	4639211	4288
Hefe	12 Millionen	6240
Fadenwurm	97 Millionen	18240
Fliege	180 Millionen	13600
Maus	3000 Millionen	25000
Mensch	3000 Millionen	25000
Mais	2400 Millionen	30 - 40000
Reis	440 Millionen	30 - 40000

nen für andere, weiter unten beschriebene, Aufgaben reserviert sind. Die exakten Tripletts können Tabelle 2.3 entnommen werden. Diese Codes sind hoch konserviert und gelten für nahezu alle Lebensformen. Ein Abschnitt einer DNA, der die Primärstruktur eines Proteins codiert, wird als Gen bezeichnet. Wichtig ist hierbei, dass es auch nicht für Gene codierende Abschnitte auf DNA-Molekülen gibt. Zur Veranschaulichung sind in Tabelle 2.4 die Längen einiger DNA-Stränge sowie die Anzahl der darauf befindlichen Gene (und somit der im Organismus befindlichen Gene) angegeben.

2.5.3 Genexpression

2.5.3.1 Transkription

Der Prozess der Bildung eines Proteins aus einem Gen wird als Genexpression bezeichnet. Diese besteht aus zwei zentralen Schritten: Transkription und Translation. Um ein Protein zu synthetisieren, muss die Nucleotidsequenz in Form eines RNA-Moleküls vorliegen. Das Erzeugen dieser RNA-Moleküle durch Umschreiben der DNA wird Transkription genannt. Bei der Genexpression tauchen drei RNA-Arten auf, die in diesem Kapitel eingeführt werden. Die erste RNA-Art ist die „messenger-RNA" (mRNA). Sie ist die Abschrift eines Gens und stellt somit den Bauplan für ein Protein dar. Das Erzeugen der mRNA anhand eines DNA-Strangs wird durch ein Enzym mit der Bezeichnung *DNA-abhängige RNA-Polymerase* vorgenommen. Wie die Bezeichnung andeutet, gibt es verschiedene Arten von Polymerasen. Allen ist gemein, dass sie für die Bildung von Nucleotidketten zuständig sind.

Um eine mRNA zu erzeugen, heftet sich die RNA-Polymerase an den Anfang eines Gens. Anschließend bewegt sie sich auf der DNA vorwärts, wobei jeweils der DNA-Bereich im Zentrum der Polymerase entwunden wird. Entsprechend dem entwundenen Bereich wird ein RNA-Strang synthetisiert, dem zur DNA komplementäre Nucleotide angefügt werden. Hierbei ist zu beachten, dass nur ein Strang der entwundenen DNA als Synthesevorlage dient. Dieser Strang wird auch als Sinn-Strang bezeichnet. Gene können jedoch auf beide Stränge verteilt sein. Während der mRNA-Synthese sind die jeweils zuletzt angefügten Nucleotiden mit ihren Komplementen

auf dem Sinn-Strang über Wasserstoffbrücken verbunden. Bild 2.12 veranschaulicht diesen Vorgang. Während die RNA-Polymerase auf der DNA voran gleitet, wird die DNA beim Verlassen der RNA-Polymerase wieder aufgewunden.

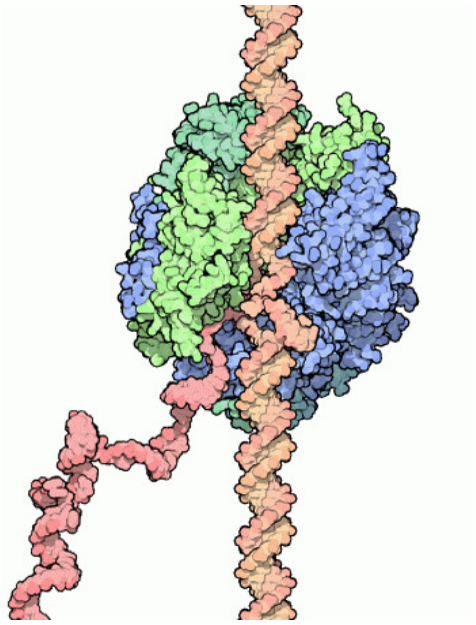

Abbildung 2.12: Eine Polymerase, die entlang eines DNA-Moleküls gleitet und einen RNA-Strang synthetisiert.

Eine RNA-Polymerase darf nicht an beliebiger Stelle mit der mRNA-Synthese beginnen. Der Start muss direkt am Anfang eines Gens erfolgen. Um diesen Punkt zu identifizieren, liegen vor Genen DNA-Bereiche, die eine Polymerase als Gen-Anfang erkennen kann. Solche Stellen nennt man Promotoren. Bei den Promotoren des Bakteriums *Escherichia Coli* sind folgende Gemeinsamkeiten festzustellen:

- Etwa 10 Nucleotide vor dem Transkriptionsstart liegt ein Abschnitt der Form $TATAAT$.
- Circa 35 Nucleotide nach dem Transkriptionsstart liegt die Seqeuenz $TTGACA$ innerhalb einer Region von vielen AT-Sequenzen.

Die oben beschriebenen Regionen werden als Konsensus-Sequenz bezeichnet. Dies deutet schon an, dass kleinere Abweichungen von dieser Form üblich sind. Die RNA-Polymerase erkennt diese Bereiche und kann daran binden. Zum Beenden der Transkription werden bei Escherichia Coli zwei verschiedene Verfahren eingesetzt:

- Rho-unabhängige Termination
- Rho-abhängige Termination

Bei der Rho-unabhängigen Termination werden am Ende des Gens sehr viele GC-Paare an die mRNA angeheftet. Diese GC-Paare können nun wie bei zwei DNA-Strängen hybridisieren und eine Haarnadel-ähnliche Struktur annehmen (siehe Bild

2.13). Diese Haarnadel kann nun in den RNA-Kanal der RNA-Polymerase gelangen und ein Auseinanderfallen der RNA-Polymerase bewirken: die Transkription ist beendet. Das Auseinanderfallen heißt nicht, dass die Polymerase unbrauchbar wäre, sie wird bei späteren Transkriptionsvorgängen wieder zusammengesetzt.

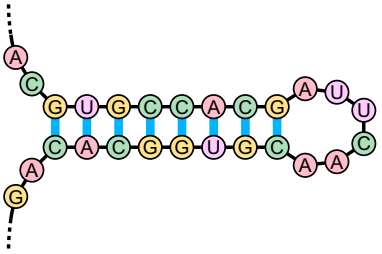

Abbildung 2.13: Eine Haarnadelstruktur.

Bei der Rho-abhängigen Termination wandert das Rho-Protein entlang des neu erzeugten RNA-Fadens in Richtung RNA-Polymerase und löst dort den Polymerase-Komplex auf.

2.5.3.2 Translation

Der Prozess, bei dem anhand von mRNA ein entsprechendes Protein synthetisiert wird, heißt Translation. Die Synthese selbst wird hierbei von einem Enzymkomplex, der als Ribosom bezeichnet wird, durchgeführt. Hierbei spielen noch zwei andere RNA-Arten eine Rolle: Das Ribosom besteht aus Proteinen und ribosomaler RNA, rRNA. Die rRNA übernimmt innerhalb des Ribosoms mehrere Aufgaben: Einerseits bindet sie die mRNA an das Ribosom, andererseits geht sie Bindungen mit der dritten RNA-Art ein, der Transport-RNA, tRNA. Die Aufgabe des Ribosoms ist es, abhängig von der mRNA-Sequenz, Aminosäuren zu Ketten zu verknüpfen. Die verschiedenen tRNAs gehen Bindungen mit einzelnen Aminosäuren ein und werden anschließend an das Ribosom gebunden. tRNA dient sozusagen als Transportmittel, um die Aminosäuren in Position zu bringen. Das Ribosom kann sich dann entlang des mRNA-Fadens vorwärts bewegen und dabei entsprechend dem Triplett-Takt die passenden Aminosäuren verknüpfen. Das Ribosom beginnt allerdings erst bei einem Methionin-Triplett (ATG) mit dem Synthesestart: Ohne dieses definierte Startcodon wäre der Synthesestart zu ungenau und der Triplett-Takt könnte verschoben werden, was ein unbrauchbares Protein zur Folge hätte. Daraus ergibt sich, dass ein Gen immer einige Nucleotide vor der eigentlichen Protein-Sequenz beginnt, um dem Ribosom die Möglichkeit des Bindens zu geben. Die Translation startet dann ab dem ersten gefundenen ATG-Triplett. Das Ende der Translation wird durch eines der folgenden Tripletts definiert: UAG, UAA und UGA. Der Code eines Proteins ist somit zwischen einem definierten Start- und einem definierten Endcodon zu finden.

Die Genexpression stellt einen der grundlegendsten Prozesse der Genetik dar. In diesem Abschnitt konnte nur ein kurzer Überblick über diesen Prozess gegeben werden. Dieser sollte jedoch genügen, um die Simulation der Genexpression in Kapitel 6 nachzuvollziehen. Detailliertere Informationen zur Genexpression sind in [Knip06] zu finden.

2.5.3.3 Escherichia Coli

Für den Benchmark wird das Genom des Bakteriums *Escherichia coli*, kurz *E.coli*, verwendet. *E.coli* wird in der Genetik als Modellorganismus betrachtet und ist Kern vieler Forschungsthemen. Viele moderne Verfahren der Gentechnik wären ohne die durch *E.coli* gewonnenen Erkenntnisse nicht umsetzbar gewesen.

Im Benchmark werden einige Gene von *E.coli* als Eingabedaten verwendet, so dass der Benchmark mittels Graphersetzung die zugehörigen RNA-Moleküle und damit wiederum die passenden Proteine produziert[4]. Daher werden an dieser Stelle einige Eigenschaften des *E.coli*-Genoms aufgezeigt.

Das Genom von *E.coli* besteht aus etwa 4.6 Millionen Basenpaaren und liegt als *ringförmiges DNA-Molekül*[5] vor. Dieses Molekül ist einige hundertmal länger als die Bakterienzelle. Um dennoch in die Zelle zu passen, ist das Molekül Knäuel-artig zusammengefaltet. Dieses Gebilde wird als *Nucleoid* bezeichnet.

In Abbildung 2.14 ist ein Ausschnitt einer Gen-Karte von *E.coli* zu sehen. Solch ein Ausschnitt wird auch im Benchmark als Eingabe für die Graphersetzung verwendet. Die exakten Gensequenzen, ihre zugehörigen Proteine und deren Verwendung können unter [ecoc] eingesehen werden.

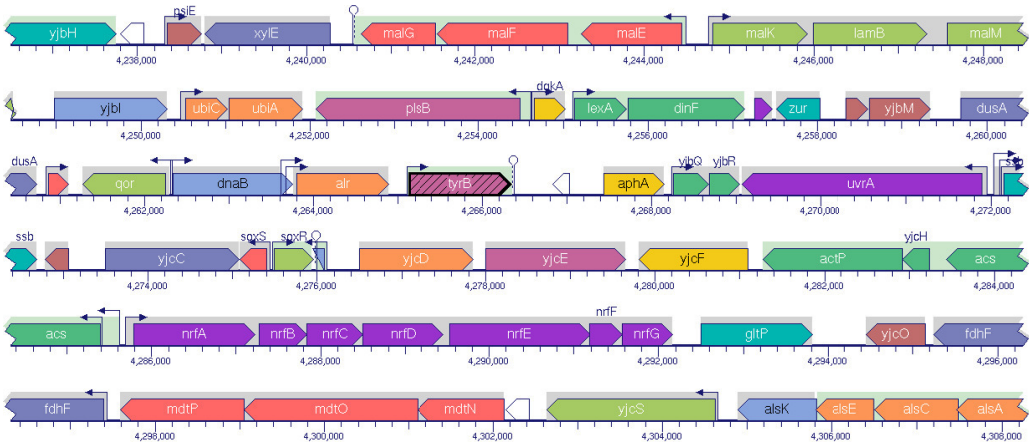

Abbildung 2.14: Ausschnitt der Genkarte von *E. coli*.

[4]Hierbei wird nur die Primärstruktur der Proteine berücksichtigt, also die Aminosäuresequenz.

[5]Das letzte Nucleotid ist mit dem ersten verbunden, so dass das Genom von *E.coli* wie ein Ring aussieht.

3. Verwandte Arbeiten

Die parallele Graphersetzung wird seit den siebziger Jahren untersucht. In diesem Kapitel werden die seither gewonnenen Erkenntnisse zusammengefasst und die Ursprünge der parallelen Graphersetzung aufgezeigt. Des Weiteren werden verwandte Strukturen wie Lindenmayer-Systeme und zelluläre Automaten vorgestellt.

Da dem parallelen Umgang mit Datenstrukturen bei der Parallelisierung einer Anwendung eine hohe Priorität zufällt, werden auch einige parallele Datenstrukturen beschrieben.

Das Kapitel schließt mit einem Überblick über einige aktuell verfügbare Graphersetzungssysteme.

3.1 Graphersetzung und theoretische Betrachtung der Parallelität

Die Graphersetzung findet ihren Ursprung in den Termersetzungs- und Textersetzungssystemen. Bereits bei diesen Systemen wurden die Voraussetzungen für Parallelität untersucht. Diese Ergebnisse flossen anschließend auch in die Theorie der Graphersetzungssysteme ein. Ebenfalls grundlegend waren die Petri-Netze, auf denen häufig parallele Prozesse stattfinden. Petri-Netze lassen sich leicht in gewöhnliche Graphen umformen, denen die ursprüngliche Semantik des Netzes durch eine Graphgrammatik zugeordnet werden kann. Somit lassen sich parallele Konzepte von Petri-Netzen leicht auf die Graphersetzung übertragen.

In diesem Abschnitt werden in 3.1.1 zunächst die Grundvoraussetzungen von Parallelität in der Graphersetzung sowie deren Ursprünge bei den Petri-Netzen und Termersetzungssystemen aufgezeigt. Anschließend werden in 3.1.2 modernere Ansätze wie verteilte Graphersetzung und deren praktische Anwendung vorgestellt. Hier wird auch die Simulation von verteilten Systemen mittels Graphersetzung behandelt.

3.1.1 Ursprung der parallelen Graphersetzung

Bei einem Graphersetzungssystem werden Produktionen nacheinander auf einen Arbeitsgraphen angewandt. Um die Ausführungszeit einer solchen Ableitung des Arbeitsgraphen zu minimieren, wäre es wünschenswert, auf aktuellen Mehrkernsystemen als auch auf verteilten Systemen mehrere solche Produktionen zur gleichen Zeit auszuführen. Dabei muss der abgeleitete Graph allerdings mit dem Ergebnis einer sequentiellen Anwendung der Regeln übereinstimmen. Es liegt nahe, dass zwei Produktionen p_1 und p_2 nur dann parallel ausgeführt werden können, wenn sie semantisch nicht voneinander abhängen, die Reihenfolge ihrer Ausführung also keine Rolle spielt.

Ein Beispiel für solche semantisch unabhängigen Vorgänge bieten die Petri-Netze [Goos00]. In Abbildung 3.1 ist das Problem der speisenden Philosophen [Tane01], dargestellt als Petri-Netz, zu sehen. Für zwei Philosophen sind die Vorgänge des Wechsels von *denken* zu *essen* durch Ellipsen markiert. Wie man sieht, sind diese beiden Vorgänge semantisch unabhängig, können also gleichzeitig ausgeführt werden.

Die gleichzeitige Ausführung mehrerer Prozesse in Petri-Netzen wird schon seit den sechziger Jahren untersucht. Außerdem weisen Petri-Netze eine enge Verwandtschaft zu Graph-Grammatiken und zur Graphersetzung auf: Reisig erläutert in [Reis81], wie Petri-Netze in Graph-Grammatiken überführt werden können. Kreowski wählt in [730287] eine anschauliche Form zur Darstellung von Petri-Netzen: Marken werden nicht in die zugehörige Stelle gemalt, sondern durch Kanten damit verbunden (siehe Abbildung 3.2). Es liegt nahe, die eigentlich getrennten Knotenmengen für Kanten und Transitionen für modernde Graphersetzungssyteme zusammenzuführen und durch Typisierung zu unterscheiden. Einen umfassenden Vergleich der Eigenschaften von Petri-Netzen und Graphersetzungssystemen liefert Kreowski in [Kreo81].

Ehrig schreibt in [Ehri83], dass die Nebenläufigkeit des Feuerns von Transitionen in Petri-Netzen exakt der Parallelität von Produktionen in Graphgrammatiken entspricht, sobald die Graphgrammatik das Petri-Netz modelliert. Des Weiteren liefert Ehrig Ansätze, wie die inhärente Parallelität der Petri-Netze auf die Graphersetzung übertragen werden kann (auch außerhalb der Simulation von Petri-Netzen).

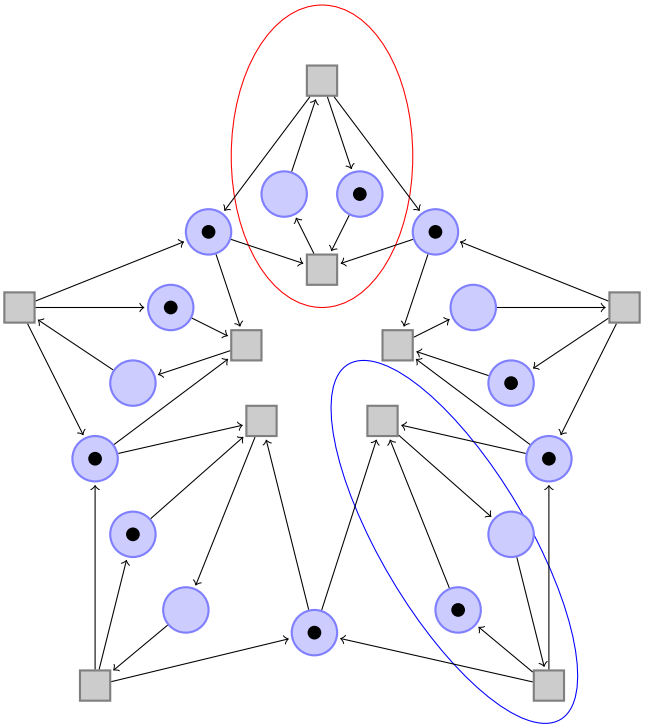

Abbildung 3.1: Die speisenden Philosophen als Petri-Netz.

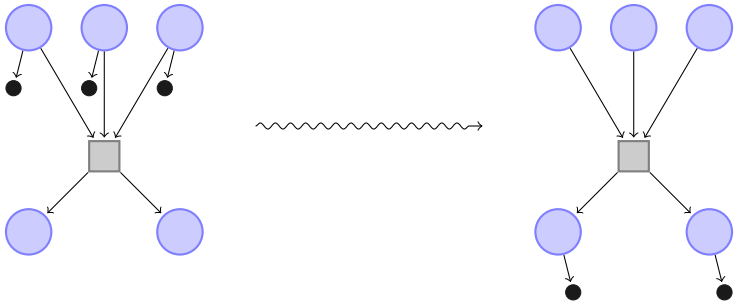

Abbildung 3.2: Eine Transition eines Petrinetzes, dargestellt als Graphtransformation. Die Marken selbst sind über Kanten verbundene Knoten.

3.1.1.1 Parallele Regeln

Im Folgenden werden grundlegende Begriffe der parallelen Graphersetzung anhand der Beschreibung von Kreowski [730287] erläutert. Diese übertragen Ansätze aus den Textersetzungssystemen sowie den Petri-Netzen auf Graphersetzungssysteme. Die hier beschriebenen Ansätze wurden in [DBLP83] erstmalig publiziert. Außerdem wird aufgezeigt, wie sich eine Menge von Graphersetzungsregeln optimal parallelisieren lässt.

Die folgenden Definitionen gelten für einen leicht vereinfachten DPO-Ansatz, lassen sich aber auch auf SPO übertragen. In Abschnitt 4 werden diese formalen Voraussetzungen wieder benötigt.

Eine *parallele Regel* ist die disjunkte Vereinigung der Mustergraphen zweier Regeln. Hierdurch ergibt sich ein etwas anderes Verhalten als bei aufeinander folgender Ausführung der ursprünglichen Regeln: Wird beispielsweise zunächst Regel A und anschließend Regel B ausgeführt, so „sieht" Regel B die Änderungen von Regel A. Bei der Ausführung einer parallelen Regel $A + B$ hingegen wird zunächst nach den Mustern von A und B gesucht, um anschließend die Änderungen am Graphen durchzuführen.

Definition 3.1.1.1 (Parallele Regel) *1. Seien $r_i = (L_i, R_i, K_i)$ für $i = 1, 2$ zwei Regeln. Dann ist $r_1 + r_2 = (L_1 + L_2, R_1 + R_2, K_1 + K_2)$ die **parallele Regel** von r_1 und r_2.*

2. Gegeben sei eine Menge P von Regeln. Die Menge P^+ paralleler Regeln über P ist dann durch $P \subseteq P^+ \wedge r_1, r_2 \in P^+ \rightarrow r_1 + r_2 \in P^+$ rekursiv definiert.

Insbesondere ist von zu beachten, dass die Bezeichnung *parallel* in einem rein graphentheoretischen Kontext zu sehen ist und nicht mit einer parallelen Ausführung im Sinne von mehreren Ausführungsfäden gleichgesetzt werden darf[1]. Würden mehrere Ausführungsfäden verwendet, so wäre völlig unklar, in welcher Reihenfolge die Aktionen *finde linke Seite von A*, *finde linke Seite von B*, *ersetze A* und *ersetze B* ausgeführt werden[2]. Bei einer parallelen Ausführung im Sinne der obigen Definition werden jedoch unabänderlich zunächst die Instanzen von A und B im Arbeitsgraphen gesucht und anschließend ersetzt.

Hierauf aufbauend wird folgendes Theorem gegeben:

Theorem 3.1.1.1 (Sequentialisierungstheorem) *Sei $M \xrightarrow[r_1+r_2]{} N$ eine parallele Regel, M der Graph, auf den sie angewendet wird und N der Graph nach der Anwendung. Dann existieren auch $M \underset{r_1}{\rightarrow} N_1 \underset{r_2}{\rightarrow} N$ und $M \underset{r_2}{\rightarrow} N_2 \underset{r_1}{\rightarrow} N$.*

Somit lässt sich jede parallele Regel wieder in zwei unabhängige, nicht-parallele Regeln aufteilen.

[1] Dies erklärt auch den Titel von [730287]: *Is parallelism already concurrency?* Parallelität und Nebenläufigkeit sind im Kontext der Graphersetzung zwei verschiedene Begriffe.

[2] Wobei ein Muster natürlich zunächst gefunden werden muss, bevor es ersetzt werden kann.

Definition 3.1.1.2 *Sei* $M \underset{r_1}{\longrightarrow} N \underset{r_2}{\longrightarrow} X$*, wobei* $h : R_1 \to N$ *die rechte Seite von* r_1 *und* $h : L_2 \to N$ *die linke Seite von* r_2 *ist. Dann ist die gegebene Ableitung* ***sequentiell unabhängig****, wenn folgende Bedingung erfüllt ist:*

$$h(R_1) \cap g(L_2) \subseteq h(K_1) \cap g(K_2)$$

Sequentielle Unabhängigkeit bedeutet somit, dass die Matches von r_1 und r_2 sich überlappen dürfen, aber nur Klebeelemente gemeinsam haben. Somit verändert eine Änderung der Ausführungsreihenfolge von r_1 und r_2 nicht die Semantik. Solche Regeln lassen sich parallelisieren, wie im Parallelisierungstheorem I und der *parallelen Unabhängigkeit* festgehalten ist:

Theorem 3.1.1.2 (Parallelisierungstheorem I) *Sei* $M \underset{r_1}{\longrightarrow} N \underset{r_2}{\longrightarrow} X$ *sequentiell unabhängig. Dann existiert* $M \xrightarrow[r_1+r_2]{} X$*.*

Definition 3.1.1.3 *Sei* $M \underset{r_1}{\longrightarrow} N_1$ *und* $M \underset{r_2}{\longrightarrow} N_2$*, wobei* $g_1 : L_1 \to M$ *und* $g_2 : L_2 \to M$ *die Matches sind. Dann sind die beiden Ableitungsschritte* ***parallel unabhängig****, wenn sie die folgende Bedingung erfüllen:*

$$g_1(L_1) \cap g_2(L_2) \subseteq g_1(K_1) \cap g_2(K_2)$$

Hierdurch lässt sich das Parallelisierungstheorem II wie folgt formulieren:

Theorem 3.1.1.3 (Parallelisierungstheorem II) *Seien* $M \underset{r_1}{\longrightarrow} N_1$*,* $M \underset{r_2}{\longrightarrow} N_2$ *(oder* $N_1 \underset{r_1}{\longrightarrow} X$*,* $N_2 \underset{r_1}{\longrightarrow} X$*) parallel unabhängig. Dann existiert eine parallele Ableitung* $M \xrightarrow[r_1+r_2]{} X$*.*

Die hiermit abgedeckten, parallelisierbaren Regelanwendungen weisen keine Schreib-/Lesekonflikte auf. Regelanwendungen, die Elemente des Klebegraphen löschen und somit Synchronisierungsmechanismen für einen korrekten Ablauf benötigen, werden in diesem Ansatz explizit ausgeschlossen und auch nicht weiter untersucht. Insbesondere wurde hierbei nicht geklärt, wie ein Softwaresystem auf effiziente Weise feststellen kann, ob eine Regelanwendung parallelisierbar ist oder nicht. In Kapitel 4 wird diese Problematik behandelt.

3.1.1.2 Äquivalenz und optimale Parallelisierung

Laut [730287] können parallele Regeln als identisch zu der unabhängigen Sequenz von Produktionen betrachtet werden. Daher kann eine Äquivalenzrelation zwischen parallelen und sequentiellen Regelsequenzen definiert werden: ist s eine Sequenz von Regelanwendungen der Form a, b, wobei a und b Regeln sind, so kann, falls eine parallele Regel $a + b$ existiert, $s \sim s'$ geschrieben werden, wenn s' die Ausführung von $a + b$ bezeichnet. Somit besteht eine Äquivalenzklasseneinteilung über den Regelsequenzen.

Als Repräsentanten der einzelnen Klassen wählt Kreowski jeweils diejenige Sequenz, welche die „optimale" Parallelisierung besitzt: Um eine Vergleichbarkeit bezüglich des

Parallelisierungsverhaltens zu ermöglichen, wird der sogenannte *Verzögerungsindex* eingeführt: Hierbei wird davon ausgegangen, dass eine Ableitung s aus einer Menge aufeinander folgender paralleler Regeln besteht, die selbst aus mehreren Regeln bestehen, formal wie folgt ausgedrückt:

$$M_0 \underset{r_1}{\longrightarrow} M_1 \underset{r_2}{\longrightarrow} ... \underset{r_m}{\longrightarrow} M_m$$

wobei $r_i = r_{i1} + ... + r_{in(i)}$ mit $n(i) > 0$ und $r_{ij} \in P$ für $i = 1, ..., m$ und $j = 1, ..., n(i)$. Dann ist der Verzögerungsindex $d(s)$ definiert als:

$$d(s) = \sum_{i=1}^{m} \sum_{j=1}^{n(i)} (i-1) = \sum_{i=1}^{m} (i-1) \times n(i)$$

Der Verzögerungsindex „bestraft" somit Produktionen, die erst sehr spät ausgeführt werden. Ein optimaler Verzögerungsindex von 0 wird erreicht, wenn alle Regeln in einem Schritt parallel durchgeführt werden können und somit keinerlei sequentielle Abhängigkeiten bestehen.

Produktionssequenzen lassen sich in Bezug auf den Verzögerungsindex optimieren, indem möglichst viele Produktionen so früh wie möglich durchgeführt werden. Eine Veränderung einer Produktionssequenz, die dies ermöglicht, ist formal durch eine *Verschiebung* definiert:

Definition 3.1.1.4 *Sei s eine Ableitung der Form $M_0 \overset{*}{\rightarrow} M_i \underset{p}{\rightarrow} M_{i+1} \underset{q+r}{\longrightarrow} M_{i+3} \overset{*}{\rightarrow} M_m$ und $M_{i+1} \underset{q}{\rightarrow} M_{i+2} \underset{r}{\rightarrow} M_{i+3}$ eine Sequentialisierung des parallelen Schrittes. Sei des Weiteren $M_i \underset{p}{\rightarrow} m_{i+1} \underset{q}{\rightarrow} M_{i+2}$ sequentiell unabhängig und $M_i \underset{p+q}{\longrightarrow} M_{i+2}$ die zugehörige Parallelisierung. Dann wird die Ableitung $M_0 \underset{*}{\rightarrow} M_i \underset{p+q}{\longrightarrow} M_{i+2} \underset{r}{\rightarrow} M_{i+3} \underset{*}{\rightarrow} M_m$ **Verschiebung** von q in s genannt.*

Durch die Ausführung einer Verschiebung kann somit der Verzögerungsindex einer Produktionsseqeuenz gesenkt und daher die potentielle Ausführungsgeschwindigkeit durch einen höheren Grad an Parallelität verbessert werden. Eine Produktionssequenz ist daher genau dann optimal, wenn keine Verschiebung mehr durchführbar ist.

3.1.1.3 Nicht-sequentielle Prozesse in Graphgrammatiken

In 3.1.1.1 wurden parallele Regeln definiert. Diese entsprechen noch nicht einer tatsächlichen gleichzeitigen Ausführung mit mehreren Anwendungsfäden. In [730387] beschreiben Kreowski und Wilharm, welche Erweiterungen hierzu nötig sind. Dieser Ansatz wird hier kurz zusammengefasst.

Bei einer nicht-sequentiellen Ausführung werden mehrere Produktionen tatsächlich gleichzeitig auf einen Arbeitsgraphen angewendet. Dies steht im Gegensatz zu einer sequentiellen Ausführung, welche einer totalen Ordnung der auszuführenden Produktionen entspricht. Dies ist in Abbildung 3.3 veranschaulicht: Während die linke

Abbildung 3.3: Links ist eine sequentielle Ausführungsanordnung zu sehen, rechts eine nicht-sequentielle. Knoten entsprechen Regelanwendungen, Kanten bedeuten *Quelle wird vor Senke ausgeführt.*

Seite mit einer sequentiellen Ausführung eine klare zeitliche Abfolge vorgibt, können auf der rechten Seite mehrere Produktionen gleichzeitig ablaufen.

Die nicht-sequentielle Ordnung spiegelt eine partielle Ordnung wider, welche die kausalen Abhängigkeiten darstellt. Wird eine nicht-sequentielle Ordnung verwendet, so wird die zugehörige Ableitung von Kreowski durch einen *Prozess* beschrieben[3]. Prozesse entsprechen einem zur Ableitung gehörigen, zyklenfreien Graphen, der unterschiedliche Reihenfolgen der Produktionsanwendung beinhaltet. Seien beispielsweise die Produktionen A, B, sowie die parallele Produktion $A + B$ gegeben. Der Prozess zu einer zweifachen Anwendung von A und einer einfachen Anwendung von B könnte dann wie Abbildung 3.4 aussehen:

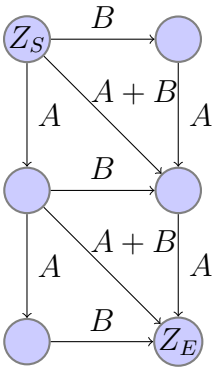

Abbildung 3.4: Darstellung eines Prozesses im Sinne von [730387]. Knoten entsprechen Zuständen des Graphen, Kanten Anwendungen von Produktionen. Z_S bezeichnet den Startzustand, Z_E den Endzustand.

Anhand dieser Darstellung der verschiedenen möglichen Anwendungsreihenfolgen der beteiligten Produktionen beschreibt Kreowski nun ein Verfahren, mit dem es möglich ist, Konflikte bei der Anwendung der Produktionen zu erkennen. Hierzu ist jedoch zunächst der Aufbau eines solchen Prozessgraphen nötig. Steigt die Anzahl der verwendeten Produktionen, beziehungsweise die Anzahl der Anwendungen der einzelnen Produktionen, so wird auch dieser Graph deutlich größer und komplexer. Des Weiteren werden in [730387] nur Beispiele mit zwei Regeln und wenigen Anwendungen erwähnt, Aussagen über die Wirksamkeit und Komplexität des Verfahrens in echten und größeren Anwendungsfällen gibt es bislang keine. Daher werden diese Verfahren zur Konflikterkennung in dieser Arbeit nicht weiter verfolgt. Konflikte werden, wie später zu sehen sein wird, nicht vor der Regelanwendung erkannt und dann umgangen, sondern durch gegenseitigen Ausschluß vermieden.

[3]Wobei auch sequentielle Ordnungen als Prozess aufgefasst werden können.

3.1.2 Verteilte Graphersetzung im Kontext verteilter Systeme

Die von Ehrig[DBLP83] und Kreowski[730287] gelegten Grundlagen im Bereich der parallelen Graphersetzung werden unter anderem von Taentzer auf verteilte Graphersetzung erweitert[Taen96b]. Motivation für diese Erweiterung lieferten verteilte Systeme: Diese werden häufig durch graphische Darstellungen repräsentiert, wobei Knoten meist für Teilsysteme und Kanten für Verbindungen stehen. Allerdings beschränken sich die graphischen Repräsentationen dieser Systeme meist auf eine Beschreibung der Systemstruktur, die exakte Spezifikation oder Implementierung wird mit anderen Mitteln realisiert. In dem in [Taen96b] verfolgten Ansatz werden Möglichkeiten aufgezeigt, die komplette Spezifikation durch einen graphenbasierten Ansatz darzustellen. Hierzu entsprechen Graphen den Systemzuständen, Zustandsübergänge werden durch Graphtransformationen umgesetzt.

Hierbei wird auch erläutert, wie Regeln in verschiedenen Teilsystemen simultan abgearbeitet werden können. Diese Parallelität wird durch die natürliche Partitionierung ermöglicht, die einem verteilten System zugrunde liegt. Diese Ansätze werden im Folgenden beschrieben, da auch in dieser Arbeit eine Zerlegung des Arbeitsgraphen genutzt wird, um eine gleichzeitige Regelanwendung zu ermöglichen, obwohl die Zerlegung des Graphen auf andere Art und Weise erfolgt.

3.1.2.1 Verteilte Zustände

Der Zustand eines Systems kann durch einen Graphen beschrieben werden, wobei die Knoten Objekte darstellen und die Kanten Relationen zwischen dieses Objekten. Im Falle eines verteilten Systems ist der Zustand des Systems verteilt. Dieser verteilte Zustand lässt sich dann durch eine Menge von Graphen darstellen, einen pro Subsystem. Der Graph, der ein einzelnes Subsystem beschreibt, wird im Folgenden als *lokaler Graph* bezeichnet.

Die meisten Subsysteme sind nicht isoliert, sondern stehen in Verbindung zu anderen Subsystemen. Diese Verbindungen werden als *Netzwerk* bezeichnet. Das Netzwerk besitzt genau wie die Subsysteme einen Zustand und kann diesen ändern. Es wird selbst als ein Graph modelliert: der *Netzwerkgraph*. Die Knoten des Netzwerkgraphen entsprechen hierbei den Subsystemen, die Kanten den Verbindungen zwischen den Subsystemen. Somit ist der Netzwerkgraph eine abstrahierte Sicht auf das Gesamtsystem, welches nur die Beziehungen der Systeme darstellt, deren konkreten Aufbau jedoch ausblendet.

In Kapitel 4 dieser Arbeit wird ebenfalls eine Verteilung von Graphen präsentiert, *Partitionierung* genannt. Auch die Objekte der Graphpartitionen können partitionsübergreifende Beziehungen besitzen. Es wird gezeigt, dass auch hier eine eigene Struktur verwendet werden kann, welche ausschließlich die Beziehungen zwischen Partitionen widerspiegelt, die den hier vorgestellten Netzwerkgraphen sehr ähnlich ist.

Um eine Kommunikation zwischen verschiedenen Subsystemen zu ermöglichen, wird jeweils zwischen zwei lokalen Graphen ein sogenannter *Schnittstellengraph* definiert. Dieser beinhaltet diejenigen Elemente, die beide lokale Graphen (und somit die hierdurch dargestellten Subsysteme) gemeinsam verwenden. Beide Subsysteme besitzen hierzu eine Kopie der Elemente des Schnittstellengraphen.

Ein Beispiel hierfür wird in [Taen96a] gegeben: Entwicklungsgraphen zeigen bei der Softwareentwicklung die Abhängigkeiten zwischen Ressourcen wie Quellcodedateien, Compilern, Tools, Editoren etc. auf. Die einzelnen Entwickler entsprechen hierbei den Subsystemen: Jeder Entwickler wird durch einen lokalen Graphen repräsentiert. Gibt es Elemente, die von mehr als einem Entwickler verwendet werden, so wird für diese Beziehung ein Schnittstellengraph angegeben. Ein Beispiel ist in Abbildung 3.5 zu sehen:

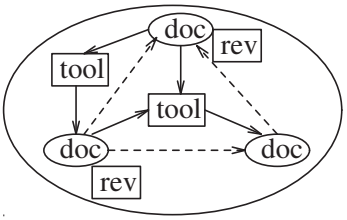

Abbildung 3.5: Ein Entwicklungsgraph (aus [Taen96a]).

3.1.2.2 Verteilte Produktionen

Verteilte Produktionen stellen Änderungen dar, die mehrere lokale Graphen betreffen. Das Erzeugen einer neuen Quellcodedatei für ein Modul müsste bei dem Beispiel aus dem vorherigen Abschnitt an alle am Modul arbeitenden Entwickler übermittelt werden. Eine verteilte Produktion besteht zu diesem Zweck aus einer *Netzwerkproduktion* sowie einer lokalen Produktion für jeden beteiligten lokalen Graphen. Verteilte Produktionen sind nötig, da in verschiedenen lokalen Systemen Repräsentanten derselben logischen Entität auftreten können. Diese müssen somit „in einem Arbeitsschritt" verändert werden. Solch eine Duplizierung von Objekten wird in dieser Arbeit nicht verwendet: Die in Kapitel 4 vorgestellten Partitionen verfügen über keinen anwendungsspezifischen Kontext und werden vom System implizit erstellt, um eine parallele Verarbeitung zu ermöglichen. Hier dient die Aufteilung jedoch der Repräsentation des Systems und wird vom Entwickler selbst vorgenommen.

Änderungen am Netzwerk, beispielsweise das Erzeugen und Löschen von Subsystemen und Verbindungen zwischen diesen, wird durch Transformationen am Netzwerkgraphen ausgedrückt.

3.1.2.3 Einordnung

Die Parallelität dieses Ansatzes basiert auf dem Anwendungsszenario: verteilte Systeme besitzen eine inhärente Partitionierung, die hier mittels Graphersetzungsregeln und Hilfskonstrukten wie den Netzwerkgraphen, die mit den lokalen Graphen eine

Hierarchie bilden, umgesetzt wird. Somit ist implizit klar, welche Aktionen parallel ablaufen können. Ist bei einem Anwendungsszenario nicht erkennbar, wo Prozesse parallel ablaufen, kann mit diesem Ansatz keine Parallelität erreicht werden. Ähnlich verhält es sich mit den in 3.1.3 vorgestellten Lindenmayer-Systemen. In Kapitel 4 wird ein Partitionierungverfahren vorgestellt, das einen ähnlichen Ansatz verfolgt wie diese Arbeit: Der Arbeitsgraph wird ebenfalls in kleinere Einheiten zerlegt, um eine parallele Ausführung zu ermöglichen. Dies geschieht allerdings ohne das Kontextwissen der konkreten Anwendung.

3.1.3 Lindenmayer-Systeme und Graphersetzung

3.1.3.1 Lindenmayer-Systeme

Lindenmayer-Systeme, kurz *L-Systeme*, wurden ursprünglich entwickelt, um Pflanzenwachstum auf einer mathematischen Grundlage beschreiben zu können. Man kann Lindenmayer-Systeme als eine anwendungsbezogene Weiterentwicklung der Chomsky-Grammatiken auffassen. Allerdings gibt es wesentliche Unterschiede zwischen Lindenmayer-Systemen und Chomsky-Grammatiken:

1. Bei den Lindenmayer-Systemen werden alle Produktionsregeln gleichzeitig angewendet.

2. Es findet keine Unterscheidung zwischen Terminalen und Nicht-Terminalen statt.

Die gleichzeitige Anwendung der Produktionsregeln ist eine Folge des natürlichen Vorbilds: Auch in natürlichen Organismen finden viele Prozesse simultan statt. Ein Beispiel ist in Abbildung 3.6 zu sehen. An diesem Beispiel ist auch zu sehen, dass sich die Parallelität, die auf einem Einzelprozessor nur im Modell, nicht aber physikalisch vorhanden ist, als eine Breitensuche mit direktem Ersetzen der gefundenen Elemente auffassen lässt.

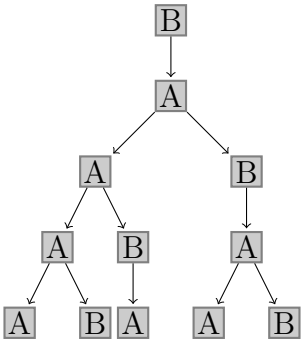

Abbildung 3.6: Ein Lindenmayer-System. Die Regeln sind $B \rightarrow A$ und $A \rightarrow AB$.

Häufig werden L-Systeme verwendet, um Graphiken zu generieren. Ein Ansatz hierzu ist die Verwendung von Grafikbefehlen, wie sie auch bei Kurvenschreibern zum Einsatz kommen (*Turtle-Graphiken*[4]). Diese nutzt den vom L-System erzeugten String als Eingabe um ein 2D- oder 3D-Bild zu generieren.

Ehrig schreibt in [Ehri83], dass die L-Systeme als Ausgangspunkt für die Entwicklung der Theorie der parallelen Graphersetzung gesehen werden können. Eine Weiterentwicklung der L-Systeme in Richtung der Graphersetzungssysteme sind die *Graph L-Systeme* von Nagl [Nagl77]. Diese werden im Folgenden kurz beschrieben.

[4]Eine *turtle* entspricht einer Bildbeschreibungssprache, bei der eine Art Stifte-tragender Roboter Befehle wie *hebe Stift*, *senke Stift* oder *nach links drehen* befolgt, um somit ein Bild zu malen.

3.1.3.2 Graph L-Systeme

Nagl betrachtet die parallele Graphersetzung in [Nagl77] als Verallgemeinerung der parallelen Ersetzung wie bei L-Systemen. Das zentrale Merkmal sei hierbei, dass, genau wie bei den L-Systemen, der Graph als Ganzes in einem Verarbeitungsschritt ersetzt wird. Als zusätzliche Voraussetzung wird angenommen, dass die linken Seiten der Graphersetzungsregeln jeweils nur aus einem Knoten bestehen. Somit sind alle Regeln in der Begriffswelt der Chomsky-Grammatiken *kontextfrei*. Dies steht in Analogie zum ursprünglichen biologischen Vorbild und ermöglicht eine konfliktfreie, parallele Anwendung, da somit gesichert ist, dass alle Vorkommen der linken Seiten parallel unabhängig im Sinne von 3.1.1.1 sind. Systeme, die diese Eigenschaften besitzen, werden *Graph L-Systeme* genannt.

Ein Vergleich auf algebraischer Basis zwischen Graph L-Systemen und (normalen) Graph-Grammatiken ist in [Nagl77] zu finden.

3.1.3.3 Einordnung

L-Systeme und Graph L-Systeme beziehen ihre massive Parallelität aus ihrem Anwendungsfall: L-Systeme entsprechen Baum-Strukturen, an denen jeweils nur die Blätter weiterverarbeitet werden. Diese werden hierbei unabhängig voneinander verändert, so dass diese Anwendung eine implizite Aufteilung der auszuführenden Arbeitsschritte besitzt. Anwendungsspezifische Möglichkeiten zur Zerteilung und parallelen Ausführung von Graphersetzung wurden bereits in 3.1.2 bei den verteilten Systemen beschrieben. Solche Vorteile lassen sich jedoch nicht in einem anwendungsübergreifenden Graphersetzungssystem ausnutzen: Während die Graphen der Graph L-Systeme eine besondere Form besitzen, muss ein allgemeines Graphersetzungssystem jegliche Form von Graphen und Regeln verarbeiten können. Daher gestaltet sich die parallele Ausführung von Graphersetzungsregeln im Allgemeinen deutlich schwieriger als in den hier beschriebenen Fällen.

3.1.4 Automatische Verkettung von Ersetzungsregeln

Bei der Regelausführung in Graphersetzungssystemen werden Schleifen verwendet, um Sequenzen von Produktionen wiederholt auf einen Arbeitsgraphen anzuwenden. In der Regelsprache von GrGen.NET[5] könnte man die wiederholte, aufeinanderfolgende Ausführung zweier Produktionen A und B wie folgt ausdrücken: $(A; B)*$.

Solche Regelsequenzen können semantisch voneinander abhängen: A könnte eine Markierung in Form eines Knotens an seine Instanz anfügen, während B diese dann weiterverarbeitet.

In [MüGe07] wird eine Methode vorgestellt, wie solche semantisch abhängigen Regelsequenzen auf eine einzige Regel reduziert werden kann. Diese Methode wird nun kurz vorgestellt.

Seien L_i und R_i für $i = 1, 2$ die linken bzw. rechten Seiten zweier semantisch abhängiger Regeln R_1 und R_2. Die zu konstruierende neue Regel sei R_{new}, ihre Seiten L_{new} bzw. R_{new}. L_{new} besteht nun aus L_1 sowie den ungebundenen Elementen von L_2. R_{new} besteht aus R_2 sowie den ungebundenen Elementen von R_1. Ein Beispiel ist in Abbildung 3.7 zu sehen:

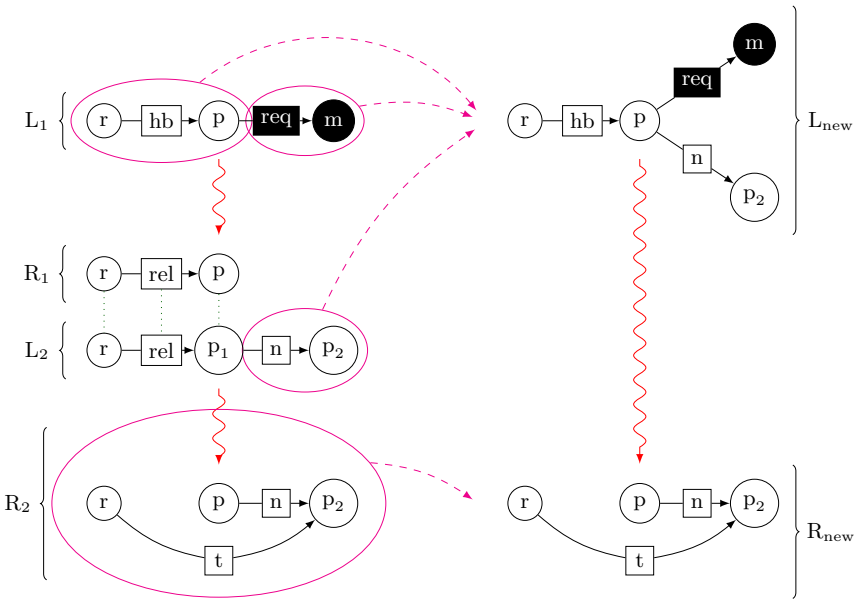

Abbildung 3.7: Zusammenführen zweier semantisch abhängiger Regeln (aus [MüGe07]).

Semantisch abhängige Regelsequenzen können nur schwer parallelisiert werden, da zwischen den einzelnen Produktionen durch die Semantik eine eindeutige zeitliche Reihenfolge festgelegt ist, die bei der Ausführung eingehalten werden muss. Die

[5]Bei GrGen.NET handelt es sich um ein Graphersetzungssystem, weitere Informationen sind in Abschnitt 3.4.3 zu finden.

Verwendung einer semantisch äquivalenten, einzelnen Regel anstatt einer Regelsequenz verkürzt nicht nur die Ausführungszeit bei sequentiellen Systemen, sondern minimiert auch den sequentiellen Laufzeitanteil bei einem parallel verarbeitenden System.

3.2 Parallele Algorithmen und Datenstrukturen

Die Ausführungsgeschwindigkeit einer Anwendung hängt maßgeblich von den verwendeten Datenstrukturen und Algorithmen ab. Daher steht der Zugriff auf die verwendeten Datenstrukturen genauso wie die darauf arbeitenden Algorithmen im Mittelpunkt einer Parallelisierung. In diesem Abschnitt werden exemplarisch einige Ansätze vorgestellt, die für die Arbeit relevante Algorithmen und Datenstrukturen für ein Mehrkernsystem anpassen.

3.2.1 Parallelverarbeitung von dynamisch verketteten Listen

Dynamisch verkettete Listen finden breite Anwendung quer durch alle Bereiche der Informatik. Sie erlauben es, Daten, die in nicht-aufeinanderfolgenden Speicherblöcken liegen, als sequentielle Datenstruktur ohne Unterbrechungen zu betrachten. Ein großer Vorteil gegenüber anderen sequentiellen Datenstrukturen wie beispielsweise Arrays oder Vektoren ist die Möglichkeit, Elemente in O(1) einzufügen, zu löschen und zu verschieben.

Dynamisch verkette Listen werden auch intensiv in GrGen.NET verwendet: Alle Graphelemente eines bestimmten Typs in einem Graphen werden in GrGen.NET in einer verketteten Liste zusammengefasst. Auch die Menge der ein- und ausgehenden Kanten eines einzelnen Knotens werden als verkettete Liste implementiert.

Probleme bei der Parallelisierung bereiten verkettete Listen allerdings dadurch, dass in der Regel nur auf ein Element der Liste direkt zugegriffen werden kann, ein wahlfreier Elementzugriff wie beispielsweise bei Feldern ist nur in $O(n)$ möglich. Sollen die Elemente einer verketteten Liste an mehrere Programmfäden verteilt werden, wird dies durch den ungünstigen wahlfreien Zugriff erschwert. Ein Ansatz, dies zu umgehen, ist in [TaYZ89] gegeben. Dieser wird im Folgenden beschrieben.

Ein nebenläufiges Abarbeiten der Liste wird in [TaYZ89] durch die Einführung eines Markierungs-Elements erreicht: Dieses befindet sich initial als Listenelement direkt hinter dem Listenkopf. Alle Listenelemente vor dem Markierungs-Element werden entweder gerade bearbeitet oder wurden bereits vollständig bearbeitet. Elemente nach der Markierung sind noch zu bearbeiten. Jedes Listenelement besitzt eine eigene Sperre, genau wie die Liste selbst.

Der Hauptfaden der Anwendung kann nach dem Einfügen des Markierungs-Elements nun beliebig viele Arbeiter-Fäden erzeugen. Diese versuchen die Sperre des Markierungs-Elements zu erlangen. Bei Erfolg wird das Markierungs-Element in der Liste um eins nach vorne geschoben, tauscht also den Platz in der Liste mit seinem Nachfolger. Anschließend wird eine Sperre auf den (ehemaligen) Nachfolger erworben und das Markierungs-Element freigegeben. So kann reihum jeder Arbeiter-Faden sich selbst ein zu bearbeitendes Element aus der Liste nehmen. Die Markierung wird also einmal durch die komplette Liste geschoben.

Für die Graphersetzung eignet sich dieser Ansatz nur bedingt: Sind nur wenige Arbeitsschritte pro Listenelement nötig, so können sich die arbeitenden Fäden an der Sperre der Markierung stauen und die Leistungssteigerung durch die Parallelisierung nur gering ausfallen. In diesem Fall möchte man, dass sich jeder Faden mehrere Elemente auf einmal nimmt, um länger beschäftigt zu sein. Dies würde bei obigem

Ansatz aber dazu führen, dass jeder Faden die Markierungs-Sperre länger besetzt, so dass sich das Problem lediglich verschiebt. In dieser Arbeit wird in Abschnitt 5.3 ein alternativer Ansatz präsentiert, bei dem eine Liste von Zeigern in die verkettete Liste verwaltet wird, so dass die Liste in O(1) in Segmente für die Arbeitsfäden zerlegt werden kann.

Neben dem parallelen Abarbeiten von dynamisch verketteten Listen möchte man idealerweise auch fundamentale Operationen wie Einfügen oder Löschen nebenläufig ausführen. In [TaYZ89] wird beschrieben, wie dies auf Basis von Sperren geschehen kann. Hierzu wird eine doppelt verkettete Liste verwendet, in der jedes Element, wie oben, eine Sperre besitzt. Die Sperren können hierbei drei verschiedene Zustände einnehmen: *blockiert*, *nicht blockiert* und *geschlossen*. Das Zustandsübergangsdiagramm ist in Abbildung 3.8 zu sehen. Folgende Schritte müssen zum Entfernen eines Knoten durchgeführt werden:

1. Schließe den zu löschenden Knoten. Ist der Knoten blockiert, warte bis der Knoten frei wird (busy-waiting).
2. Blockiere den Vorgänger. Ist er gesperrt, aktualisiere die Adresse des Vorgängers und wiederhole diesen Schritt.
3. Aktualisiere den *next*-Zeiger des Vorgängers und den *previous*-Zeiger des Nachfolgers.
4. Entsperre den Vorgängerknoten.

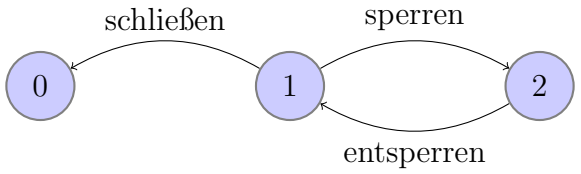

Abbildung 3.8: Zustandsübergangsdiagramm einer Sperre. 0 steht für den Zustand *geschlossen*, 1 für *entsperrt* und 2 für *gesperrt*.

Dieser Ansatz ermöglicht es, gleichzeitig an verschiedenen Stellen in der Liste Elemente einzufügen und zu löschen, wobei benachbarte Elemente jedoch ausgeschlossen werden. Allerdings hat dieser Ansatz einen erhöhten Speicherbedarf, da pro Listenelement eine Sperre erzeugt werden muss. Des Weiteren sind Sperr-Operationen relativ teuer, so dass es sehr stark vom Anwendungsfall abhängt, ob hierdurch die Leistung gegenüber sequentiellen Operationen erhöht werden kann: Sind beispielsweise die Sperroperationen genauso aufwändig wie das eigentliche Neusetzen der Zeiger, so wäre die Anwendung auf einem Zweikern-System bei vielen Löschoperationen in etwa gleich schnell wie eine sequentielle Implementierung (der erhöhte Speicherbedarf bleibt jedoch weiterhin vorhanden).

3.2.2 Partitionierung von Graphen

Möchte man Anwendungen für Mehrkernsysteme oder verteilte Systeme parallelisieren, so müssen häufig die Daten, auf denen die Anwendung arbeitet, zerteilt werden, um den Mehrwert solcher Systeme ausnutzen zu können. Handelt es sich bei

der zugrunde liegenden Datenstruktur um einen Graphen, so tritt das Problem der Graphpartitionierung auf. In [HeLe95] wird das Problem wie folgt beschrieben:

- **Gegeben:** Ein (gewichteter) Graph $G = (V, E)$ und ein Parameter p.
- **Gesucht:** Eine Partitionierung der Knoten von G in p Mengen, so dass die Summe der Knotengewichte in jeder Menge möglichst identisch ist und die Summe der Kantengewichte von Kanten zwischen zwei Mengen minimiert wird.

Wie in Abschnitt 4.1 gezeigt wird, spielt die Lösung dieses Problems auch für die parallele und verteilte Graphersetzung eine Rolle. Leider liegt das Problem in der Klasse der NP-vollständigen Probleme[GaJS74].

3.2.2.1 Ein Algorithmus zur Graphpartitionierung

In [HeLe95] schlagen Hendrickson und Leland einen Algorithmus vor, mit dem das Problem der Graphpartitionierung in linearer Zeit näherungsweise gelöst werden kann. Wie so oft bei NP-vollständigen und NP-harten Problemen wird eine näherungsweise Lösung bestimmt, indem der Lösungsraum des Problems eingeschränkt wird. So ist die optimale Lösung gegebenenfalls nicht mehr zu bestimmen, jedoch oft eine hinreichend gute Lösung.

Diesem Ansatz folgen auch Hendrickson und Leland. Ihr Algorithmus ist hierbei in drei wesentliche Schritte unterteilt:

1. Entferne schrittweise Details[6] aus dem zu partitionierenden Graphen.
2. Partitioniere die detailärmste Version des Graphen.
3. Fülle die Partitionen wieder mit den entfernten Details, wende hierbei ein lokales Verfeinerungsverfahren an.

Die Anzahl der möglichen Partitionierungen eines Graphen wächst mit der Anzahl seiner Graphelemente. Somit führt das Entfernen von Details, also von Graphelementen, zu einer Einschränkung des Suchraums und einem schnelleren Auffinden einer Lösung. Hierzu wird eine Lösung des Problems der maximalen Paarung[7] angewendet: Dies lässt sich in zur Anzahl der Kanten proportionaler Zeit durchführen.

In Schritt zwei wird ein solcher, vereinfachter Graph partitioniert. Dies kann mit verschiedenen Partitionierungsmethoden geschehen, vorgeschlagen wird von den Autoren das Kernighan-Lin-Verfahren[KeLi70].

Im letzten Schritt werden die zuvor entfernten Knoten und Kanten wieder in den Graphen (beziehungsweise in die Partitionen) eingefügt.

Eine Implementierung dieses Algorithmus ist in der Chaco 2.0 Bibliothek verfügbar [B. H94]. Hierbei handelt es sich um eine C-Bibliothek, die verschiedene Graphpartitionierungsalgorithmen anbietet und in vielen Bereichen eingesetzt wird. Hierzu gehört die Problemzerlegung für parallele Computer, Entwurf von Schaltungen oder auch die Organisation von Datenbanken.

[6] Hiermit ist gemeint, dass Knoten und Kanten aus dem Graphen entfernt werden, so dass der dadurch entstehende Graph eine Annäherung des ursprünglichen Graphen darstellt.

[7] Finde eine maximale Menge an Kanten in einem Graphen, so dass keine zwei Kanten denselben Knoten berühren.

3.3 Graphersetzung im Anwendungsbereich der Biologie

3.3.1 Graph Replacement Chemistry for DNA Processing

Der Ansatz, biochemische Prozesse auf DNA-Molekülen durch Graphersetzungssysteme zu simulieren, wurde bereits 2000 veröffentlicht: McCaskill und Niemann beschreiben in [McNi01] einen Formalismus für die Handhabung der kombinatorischen Komplexität von enzymatisch katalysierten Reaktionen in Nucleinsäurekomplexen. Dies geschieht, indem biochemische Reaktionen als Produktionen eines Graphersetzungssystems dargestellt werden. Als Beispiel wird die Synthese von RNA-Molekülen aus DNA aufgezeigt. Der vorgestelle Ansatz enthält viele Parallelen zu dem im Rahmen dieser Arbeit entwickelten Benchmark. Hier soll nun die Vorgehensweise aus [McNi01] aufgezeigt werden. In Kapitel 6 wird bei der Beschreibung des Benchmarks auf Unterschiede und Gemeinsamkeiten hingewiesen.

3.3.1.1 Molekular-Graphen

Als grundlegend für die Umsetzung von biochemischen Reaktionen mittels Graphersetzungssystemen wird eine Erweiterung des Graph-Formalismus angegeben. Die Umsetzung von Atomen und Bindungen zwischen diesen ist naheliegend, funktionale Gruppen (beispielsweise OH oder CH_3), die in Reaktionen als Einheit auftreten, können allerdings auch als ein einzelner Knoten repräsentiert werden. Um solche Gruppen austauschbar zu machen, sind jedoch zwei Erweiterungen an den Markierungen von Knoten und Kanten notwendig:

1. Jede Knoten- und Kantenmarkierung wird in eine Liste von Attribut- und Wertepaaren[8] aufgeteilt.
2. Jeder Attributwert wird mit einer Eigenschaft versehen, welche die Werte *variabel* oder *konstant* annehmen kann.

Hierdurch ergibt sich die folgende Definition von *Variablen-Graphen*:

Definition 3.3.1.1 *Ein **Variablen-Graph** ist ein Graph mit Knoten- und Kantenmarkierungen mit einer Markierungsmenge $\Sigma_{VE} = A \times V$, wobei A eine Menge von Attributwerten ist und V eine Menge von Attributeigenschaften, welche Variablennamen vom Typ String repräsentieren und die beiden Strings **don't care** und **match** enthält.*

Die Einführung von Variablen-Graphen erlaubt eine allgemeinere Strategie bei der Mustersuche: Nicht nur die Struktur des Graphen wird in die Suche mit einbezogen, sondern auch die Belegung von Attributwerten. Mit den Flags *don't care*, *match* oder einem *Variablennamen* kann bestimmt werden, wie der zugehörige Attributwert bei einem Matchingvorgang behandelt wird. Ist ein Variablenname gesetzt, der mehrfach im Graphen vorkommt, so wird die Variablenbindung als *konstant* betrachtet, so

[8] Dies wäre schon durch die Verwendung von attributierten Graphen möglich, wird hier jedoch in leicht veränderter Form eingeführt.

dass überall die gleichen Ersetzungsvorgänge durchgeführt werden. Beispielsweise wäre bei einem Graphen, der ein Molekül der Form $R - OH$ darstellt, dem R ein *don't care* zugeordnet. Bei einem Ether der Form $R - O - R$ wäre hingegen eine echte Variable zugeordnet, so dass die R-Gruppen jeweils identisch sein müssen[9]. Dies ist in Abbildung 3.9 noch einmal veranschaulicht.

Abbildung 3.9: Es sind zwei Variablengraphen zu sehen. Einmal kommt die R-Variable doppelt vor, und wird bei Anwendung der Regel $R \rightarrow H_3C$ zweimal ersetzt.

Auf den *Variablen-Graphen* aufbauend werden nun *Nucleinsäure-Graphen* definiert:

Definition 3.3.1.2 *Ein* ***Nucleinsäure-Graph*** $G = (V, E, I, \Sigma_{VE})$ *ist ein Graph mit Knoten- und Kantenmarkierung, einem maximalen Kantengrad von 3, einer Markierungsmenge* $\Sigma_{VE} = (|lab| \times |type| \times |B| \times |rev| \times |enz|) \cup \{prev, next, hybrid\}$, *wobei* $|lab|$ *eine Markierungsmenge,* $|type| = \{RNA, DNA\}$, $|B| = \{A, G, C, T, U\}$, $|rev| = \{0, 1\}$, $|enz|$ *eine Menge von Enzymmarkierungen und* $|prev|$, $|next|$ *und* $|hybrid|$ *Kantenmarkierungen sind. Die gerichteten Kanten der Klasse* $|next|$ *(*$|prev|$*) zeigen in 5'-3'(3'-5')-Richtung des DNA-Moleküls. Kanten des Typs* $|hybrid|$ *verbinden über Wasserstoffbrücken verbundene Basenpaare.*

Um mehrere identische Monomere zu Oligonucleotiden zusammenzufassen, werden *Polynucleotid-Graphen* wie folgt definiert:

Definition 3.3.1.3 *Ein* ***Polynucleotid-Graph*** $G = (V, E, L, \Sigma_{VE})$ *ist ein Nucleinsäure-Graph, bei dem das* $|base|$*-Attribut der Markierungsmenge* Σ_{VE} *durch* $|seq| = B^+$ *ersetzt wird, also durch die Menge aller Nucleinsäurebasensequenzen.*

Somit repräsentiert ein Knoten in einem Polynucleotid-Graphen eine Sequenz von Nucleotiden. Man beachte, dass bei beiden Graph-Spezialisierungen die Bindung zwischen aufeinanderfolgenden Nucleotiden mit zwei Kanten (*prev* und *next*) und nicht mit einer repräsentiert wird.

[9]Bei typisierten Graphen wären solche Bedingungen leicht durch Basisklassenbildung zu formulieren, worauf die Autoren von [McNi01] jedoch verzichtet haben. Im Rahmen dieses Artikels wurde auch eine Implementierung in der Programmiersprache C entwickelt. Der Gedanke liegt nahe, dass die fehlende Objektorientierung der Sprache bereits beim formalen Konzept durch die obige Attributierungsweise abgefangen wurde.

3.3.1.2 Molekulare Reaktionen durch Graphersetzung

Folgende Arten von chemischen Reaktionen werden für das Modell unterschieden:

- **Unimolekulare Reaktionen**: Nur eine Molekül ist an der Reaktion beteiligt und kann somit durch einen Subgraphen bestimmt werden.
- **Bimolekulare Reaktionen**: Zwei lokale Bereiche zweier Moleküle reagieren miteinander, hierbei werden neue Knoten/Kanten zwischen den Molekülsubgraphen ausgebildet, so dass die beiden Subgraphen zu einem einzigen Subgraphen verschmelzen.
- **Enzymatische Reaktionen**: Dieser Fall wird auf Enzyme beschränkt, die sich im Verlauf der Reaktion nicht verändern. Somit kann die Reaktion in vielen Fällen wie eine unimolekulare Reaktion behandelt werden, das Enzym selbst muss keine explizite Darstellung im Graphen besitzen. Umgesetzt werden kann dies durch eine Attributerweiterung der Kanten, so dass der aktuelle Reaktionspunkt des Enzyms markiert werden kann. Alternativ kann das Enzym aber auch durch einen Knoten im Graphen dargestellt werden, der über eine Kante mit dem aktuellen Reaktionsort verbunden ist[10].

Der Unterschied zwischen unimolekularen und bimolekularen Reaktionen ist insofern wichtig, als dass er eine Steuerung der Ausführungsreihenfolge von Regeln ermöglicht: Beispielsweise wäre es eine aus chemischer Sicht sinnvolle Einschränkung, dass auf einem Molekül erst alle möglichen unimolekularen Reaktionen ausgeführt werden müssen, bevor das Molekül bimolekulare Reaktionen eingehen darf. Hierdurch wird das Produktspektrum der Reaktion eingeschränkt und kurzlebige chemische Zwischenprodukte vermieden.

Ein Beispiel für enzymatische Reaktionen ist die RNA Synthese: Die Reaktion einer RNA-Polymerase kann in drei Einzelschritte zerlegt werden: Initiierung, Elongation, Termination. Jeder Schritt beinhaltet eine Vielzahl von einzelnen Proteinen und Reaktionen zwischen diesen. Nach dem fertigen Prozess liegen diese Elemente jedoch wieder in ihrer ursprünglichen Form vor und können einen weiteren Prozess starten. Daher kann auch die exakte Darstellung der temporären Zwischenzustände verzichtet werden. Dieser Ansatz wird auch im Genexpressions-Benchmark verfolgt, der in dieser Arbeit entwickelt wurde.

3.3.1.3 Zusammenfassung

Diese Arbeit von McCaskill und Niemann zeigt, dass die Graphersetzung das Potential besitzt, als Grundlage der Simulation von genetischen Prozessen zu dienen. Die hier vorgestellten Überlegungen zur Umsetzung solcher Graphersetzungssysteme wird in Kapitel 6.1.1 aufgegriffen und auf modernere Graphersetzungssysteme übertragen. Die Verwendung von aktuelleren graphentheoretischen Strukturen wie typisierten Graphen stellt somit eine Weiterentwicklung zur in diesem Abschnitt vorgestellten Arbeit dar.

[10] Es wird davon ausgegangen, dass temporäre Änderungen am Enzym nach der Reaktion nicht mehr von Bedeutung sind und somit keine genaue Darstellung des Enzyms auf molekularer Ebene nötig ist. Dieses Vorgehen wird auch im Genexpression-Benchmark verwendet.

3.3.2 Zelluläre Hypergraphen

Zellularautomaten[ToMa87] eignen sich, um die Selbstorganisation von komplexen Systemen zu untersuchen: Elemente, sogenannte Zellen, sind nach einem bestimmten Muster angeordnet, so dass jede Zelle eine Menge von Nachbarn hat. Jede Zelle ist mit einem endlichen Automaten ausgestattet. Die Zustandsübergänge hängen von den aktuellen Zuständen der Nachbarn einer Zelle ab. Dieses System besitzt einen inhärenten Parallelismus, es lässt sich somit gut für Multiprozessor- und verteilte Systeme umsetzen.

Zellularautomaten bieten keine Unterstützung für strukturelle Änderungen am System. So ist beispielsweise das Entfernen und Hinzufügen von Zellen nicht vorgesehen. Dies ist aber für die Simulation von biologischen Prozessen wie Zellproliferation oder Zellmigration unerlässlich.

Um dieses Problem zu umgehen, beschreibt Hartmann in [Hart] eine Kombination aus Zellularautomaten und Graphersetzung, welche er als *zelluläre Hypergraphen* bezeichnet. Hierbei werden die Automaten, die den Zustand einer Zelle beschreiben, durch eine Menge von Graphproduktionen ersetzt. Hierbei sind explizit Produktionen erlaubt, die Zellen teilen oder auch vereinen.

Hyperkanten[11] werden in diesem System verwendet, um Zellen zu repräsentieren. Eine Zelle entspricht somit einer Hyperkante, die all diejenigen Knoten verbindet, welche die Zelle von ihrer Umgebung abgrenzen. In Abbildung 3.10 würde beispielsweise $\{a, b, c, d, e\}$ eine Zelle beschreiben.

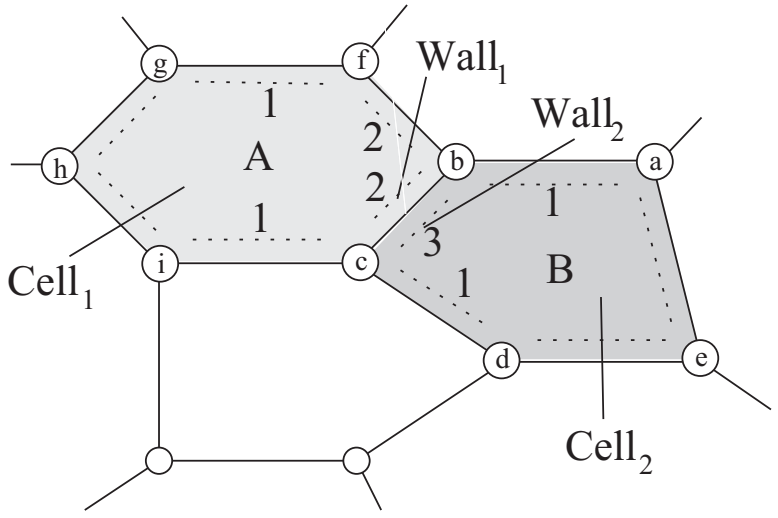

Abbildung 3.10: Ein zellulärer Hypergraph.

Zellen besitzen somit eine explizite Repräsentation im System. Hierdurch sind sie als Objekte greifbar, was dazu führt, dass das NP-vollständige Subgraph-Matching-Problem, welches bei der Graphersetzung grundsätzlich auftritt, in $O(n)$ gelöst werden kann[Hart].

[11] Hyperkanten sind Kanten, welche mehr als zwei Knoten miteinander verbinden können. Ein Graph, der Hyperkanten erlaubt, wird auch Hypergraph genannt.

Des Weiteren kann die Darstellung von Zellen auch als Partitionierung des Graphen aufgefasst werden, wobei jede Zelle selbst einer Partition entspricht. Zelluläre Hypergraphen erben weiterhin von den Zellularautomaten die Eigenschaft, dass jede Zelle nur von einer wohldefinierten Untermenge der vorhandenen Zellen beeinflusst wird, den Nachbarzellen. Führt man eine Graphersetzungsregel auf eine Zelle aus, so hat dies höchstens auf die Zelle selbst und ihre Nachbarn Auswirkungen[12]. Somit ergibt sich, dass Regeln immer dann parallel angewendet werden können, wenn die betroffenen Zellen keine gemeinsamen Nachbarn besitzen. Daher eignen sich zelluläre Hypergraphen gut, um verteilte und parallele Graphersetzung zu betreiben.

In [Hart95] beschreibt Hartmann, wie zelluläre Hypergraphen für parallele und verteilte Systeme umgesetzt werden können. Grundelement sind hierbei die Zellen, welche bei Bedarf zwischen den einzelnen Systemen getauscht werden können. Dies ist beispielsweise zur Lastverteilung wie auch zur Ausführung von Regeln, die „auf dem Rand" zwischen zwei Systemen ausgeführt werden, nötig.

Parallelität wird in diesem Fall also durch die inhärente Unterteilung des Problems selbst gewonnen. Dies entspricht dem in 3.1.2 beschriebenen Ansatz: Hier wird die logische Zerlegung, die bei verteilten Systemen nur natürlich ist, verwendet, um eine parallele Ausführung zu ermöglichen.

[12] Zu beachten ist hier jedoch der besondere Fall, dass sich eine Zelle teilt: Hierdurch wird eine neue Zelle erzeugt, die selbst wieder eine eigene Nachbarmenge besitzt und auch die Nachbarn der umgebenden Zellen verändert.

3.4 Graphersetzungssysteme

3.4.1 AGG - Attributed Graph Grammars

Bei AGG[Taen99] handelt es sich um ein Mehrzweckgraphersetzungssystem. Einen Schwerpunkt bildet hier die Benutzeroberfläche: Arbeitsgraphen können genauso wie Regeln mit der Oberfläche erzeugt und mittels integrierten Layoutalgorithmen ausgerichtet werden. AGG unterstützt negative Anwendungsbedingungen sowie attributierte und typisierte Knoten und Kanten.

Knoten und Kanten können mit Java-Objekten attributiert werden, für die eigentliche Graphersetzung wird der Single-Pushout-Ansatz unterstützt. Die Graphersetzung kann entweder automatisch erfolgen oder vom Benutzer schrittweise gesteuert werden. Hierbei wird auch eine benutzerdefinierte Selektion des nächsten zu verwendenden Vorkommens der Regel unterstützt.

Produktionen werden von AGG in zweifacher Art indeterministisch angewendet: die nächste auszuführende Regel wird indeterministisch aus den zur Verfügung stehenden Regeln ausgewählt, anschließend wird auch das Match wieder zufällig selektiert. Eine Ausnahme bei der Matchwahl bildet der oben genannte interaktive Modus. Um die Reihenfolge der Regelauswertung zu steuern, gibt es die Möglichkeit, die Regeln in sogenannte *Layer* einzusortieren. Diese bilden somit jeweils eine Submenge der verfügbaren Regeln. Zu jedem Ausführungszeitpunkt ist genau ein Layer aktiv, die verwendeten Regeln werden aus diesem indeterministisch gewählt. Der Anwender kann nun jedoch die Reihenfolge der Layer vorgeben.

AGG bietet folgende Möglichkeiten zur Validierung:

- Konsistenzprüfung für Graphen
- Abhängigkeitserkennung auf Basis von kritischen Paaren
- Terminierungsprüfungen

In [Taen99] wird angegeben, dass die Implementierung von AGG auch die nötigen Voraussetzungen, um parallele und verteilte Graphersetzung durchzuführen, beinhaltet. Bisher wird dies jedoch nur auf sequentieller Basis ausgeführt: Ein echtes Multithreading wird nicht betrieben. Gerade für eine verteilte Anwendung bietet die Java-Implementierung bereits gute Voraussetzungen, da nach [Taen99] die Java-Umgebung die Möglichkeit bietet, Objekte über Systemgrenzen zu migrieren. Die vorhandene Abhängigkeitserkennung auf Basis von kritischen Paaren liefert die Voraussetzung, um entscheiden zu können, welche Regeln tatsächlich konfliktfrei parallel ablaufen können.

3.4.2 VIATRA - VIsual Automated model TRAnsformations

Bei VIATRA [CHMP+02] handelt es sich um ein Graphersetzungssystem, welches für Modelltransformationen spezialisiert wurde. VIATRA beinhaltet zwei Sprachen: VTML (VIATRA Textual (Meta)Modelling Language) wird verwendet, um Metamodelle und Modelle zu definieren. in VTCL (VIATRA Textual Command Language) werden Regeln definiert, welche die in VTML notierten Modelle und Objekte bearbeiten. Des Weiteren wird in VTCL die genaue Ablaufsteuerung angegeben, nach der die Regeln angewendet werden.

VIATRA bietet die Möglichkeit einer parallelen Regelanwendung. Hierbei sind durch die VTCL-Schlüsselworte *forall* und *parallel* zwei verschiedene Modi definiert: Mit *forall* werden zunächst alle Vorkommen einer Regel gesucht und anschließend ersetzt[13]. Mit *parallel* werden verschiedene Regeln gleichzeitig ausgeführt. Beide Parallelitätsansätze werden jedoch unter Verwendung eines einzigen Ausführungsfadens ausgeführt. Insbesondere führt die Verwendung von *parallel* somit dazu, dass die Reihenfolge der Befehlsausführung zufällig, aber sequentiell erfolgt. VIATRA verwendet den Begriff *Parallelität* somit im graphentheoretischen Sinne und verbindet diesen nicht mit einer tatsächlichen gleichzeitigen Ausführung auf Mehrkernsystemen.

Dass die potentiellen Möglichkeiten zu einer echten, parallelen Ausführungen nicht ausgeschöpft werden, ist bei Graphersetzungssystemen üblich. In [GTBB+08] wird im Rahmen eines Vergleiches mehrerer Graphersetzungssysteme geschrieben, dass nahezu kein System irgendeine Form der Ausführung unter Verwendung mehrerer Ausführungsfäden bietet.

[13] Dies entspricht dem *ApplyAll*-Ansatz von GrGen.NET.

3.4.3 GrGen.NET

GrGen.NET ist ein Mehrzweckgraphersetzungssystem. GrGen.NET unterstützt benutzerdefinierte Metamodelle für Graphregeln. Hierbei werden Mehrfachvererbung für Knoten und Kanten unterstützt. Graphelemente können mit beliebig vielen Elementen einer vorgegebenen Typmenge attributiert werden. Für Metamodelle können zusätzlich Einschränkungen definiert werden, so können bestehende Graphen auf Konformität zu einem gegebenen Metamodell geprüft werden.

Regeln werden in einer eigenen Regelsprache formuliert. Für die Mustersuche können negative Anwendungsbedingungen sowie Typeinschränkungen verwendet werden. Des Weiteren können Wertbelegungen von Attributen die Suche verfeinern. Die Mustersuche kann wahlweise isomorph oder homomorph durchgeführt werden. Beim Anwenden der rechten Seite einer Regel können Attribute neu belegt, Typen von Graphelementen verändert, sowie neue Elemente vom exakten Typ eines gefundenen Elementes erstellt werden. Die Abbildungen 3.11 bis 3.13 zeigen die Definition eines Metamodells, eine Regel, die auf diesem Metamodell basiert, sowie eine graphische Darstellung der Regel.

```
node class Node1;
node class Node2;
node class Node3;

edge class Edge1;
edge class Edge2;
edge class EdgeSperr;
```

Abbildung 3.11: Ein einfaches Metamodell für GrGen.NET.

```
rule makeFlake2 {
 pattern {
  a:Node -:Edge2-> b:Node;
 }
 replace {
  a -:Edge1-> c:Node -:Edge1-> d:Node1 -:Edge1-> e:Node -:Edge1-> b;
  c -:EdgeSperr-> e;
 }
}
```

Abbildung 3.12: Eine Graphersetzungsregel, welche das Metamodell aus Abbildung 3.11 verwendet.

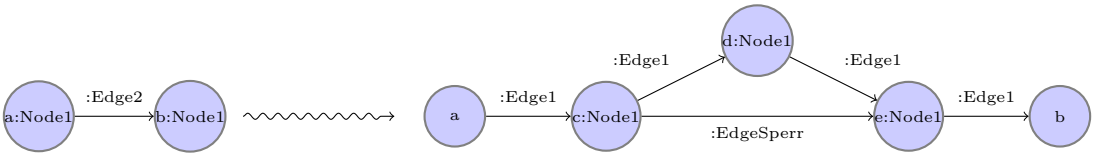

Abbildung 3.13: Graphische Darstellung der Regel aus Abbildung 3.12.

Tabelle 3.1: Die verschiedenen Dateitypen von GrGen.NET.

Dateityp	Verwendung
Modelldatei (*.gm)	In diesen Dateien wird definiert, aus welchen Knoten- und Kantentypen ein Graph besteht und welche Attribute diese Graphelemente besitzen. Des Weiteren werden die vorhandenen Vererbungsbeziehungen angegeben.
Regeldatei (*.grg)	Dieser Dateityp dient zur Definition der Graphersetzungsregeln. Diese sind hier lediglich deklarativ angegeben und bilden somit noch kein ausführbares GrGen-NET-Programm. Jede Regeldatei referenziert eine Modelldatei. Das darin deklarierte Modell kann zur Definition von Regeln verwendet werden.
GrShell-Skript (*.grs)	GrShell.Skripte wenden die in einer Regeldatei definierten Regeln auf einen Arbeitsgraphen an. Hierbei können auch logische Ausdrücke zur Ablaufsteuerung sowie Wiederholungsanweisungen verwendet werden.

GrGen.NET bietet eine Ablaufsteuerung der Regelanwendung wahlweise durch eine .NET-Programmierschnittstelle oder interaktiv durch eine Eingabeaufforderung. Für die Eingabeaufforderung ist eine Skriptsprache definiert, die Regeln anhand von logischen Ausdrücken anwendet. So kann beispielsweise angegeben werden, dass eine Regel solange angewendet werden soll, wie Muster dieser Regel gefunden wurden. Auch die logische Verknüpfung von Regeln ist hierbei möglich. Um Anwendungen für GrGen.NET zu entwickeln, werden drei verschiedene Dateitypen verwendet. Diese sind in Tabelle 3.1 aufgeführt.

Der Eingabeaufforderungsmodus von GrGen.NET unterstützt einen Debug-Modus, mit dem die Regelauswertung schrittweise erfolgen kann. Zur weiteren Unterstützung steht mit yComp eine graphische Ausgabe für GrGen.NET zur Verfügung.

Intern arbeitet GrGen.NET mit einem Compiler, der Dateien mit Graphersetzungsregeln in Module für das .NET-Framework übersetzt. Auf diese Weise werden Regelanwendungen äußerst schnell durchgeführt. GrGen.NET ist nach aktuellem Stand das schnellste verfügbare Graphersetzungssystem.

3.4.4 Werkzeugauswahl

Keines der oben beschriebenen Systeme unterstützt aktuell eine Ausführung unter Verwendung mehrerer Ausführungsfäden. Eine Regelanwendung mit simultaner Ausführungssemantik ist in verschiedener Form jedoch bei allen oben beschriebenen Werkzeugen möglich.

AGG wurde speziell unter Berücksichtigung des Anwendungsfalles der verteilten Systeme entwickelt und sollte sich daher gut für eine Parallelisierung eignen. In der zu AGG verfügbaren Literatur wird auch darauf hingewiesen, dass erste Ansätze zu einer parallelen Implementierung bereits angedacht wurden.

VIATRA bietet die Möglichkeit, Regeln mit paralleler Semantik, also in zufälliger Reihenfolge, auszuführen. Ansätze für eine echt parallele Implementierung sind jedoch bisher nicht bekannt.

GrGen.NET wurde als Allzweck-Graphersetzungssystem entwickelt. Ziel war es, das schnellste verfügbare Graphersetzungssystem zu entwickeln, was nach aktuellen Ergebnissen [VaSV05] auch erreicht wurde. Dies wird unter anderem durch eine performante Wahl der verwendeten Datenstrukturen, aber auch durch die Verwendung eines eigens für GrGen.NET entwickelten Code-Generators erreicht.

In dieser Arbeit wird die Parallelisierung von Graphersetzungssystemen unter dem Gesichtspunkt einer schnelleren Ausführung von Regeln und Regelsequenzen betrachtet. Daher wird GrGen.NET zur Umsetzung verwendet: Es scheint schlüssig, das aktuell schnellste System „noch schneller" machen zu wollen. Die hierbei gewonnenen Erkenntnisse sollten sich zumindest teilweise auch auf andere Graphersetzungssysteme übertragen lassen.

Ein weiterer Vorteil von GrGen.NET ist die Unabhängigkeit von bestimmten Anwendungsfällen: Die praktische Verwendung von Graphersetzungssystemen steckt immer noch in den Anfängen, trotz der bereits gut erforschten formalen Hintergründe. Es ist somit zu erwarten, dass wirklich zeitkritische Anwendungen erst in Zukunft behandelt werden. Diese benötigen dann ein sehr schnelles System, welches sich leicht an neue Aufgabenstellungen anpassen lässt. GrGen.NET scheint hierfür am besten geeignet.

4. Analyse

In diesem Kapitel wird aufgezeigt, wie die Parallelisierung eines Graphersetzungssystems erfolgen kann. Hierzu werden grundlegende Konzepte vorgestellt, die unabhängig von einer konkreten Implementierung sind.

Abschnitt 4.1 behandelt, wie eine parallele Verarbeitung von Graphersetzungsregeln durch eine Partitionierung des Arbeitsgraphen umgesetzt werden kann. Im Rahmen dieser Arbeit wurde ein eigenes Partitionierungsverfahren entwickelt, welches ebenfalls in 4.1 vorgestellt wird.

Zusätzliche Parallelisierungsmöglichkeiten werden in Abschnitt 4.2 vorgestellt. Hierzu zählen insbesondere verschiedene Formen der parallelen Suche.

4.1 Parallelisierung durch Partitionierung des Arbeitsgraphen

In diesem Abschnitt wird gezeigt, wie die parallele Verarbeitung von Graphersetzungsregeln durch eine Partitionierung des Arbeitsgraphen profitieren kann. Wie Graphpartitionen erzeugt werden, wird in Abschnitt 4.1.2 behandelt. Da sich ein Arbeitsgraph durch die Anwendung von Graphersetzungsregeln verändert, wird in Abschnitt 4.1.3 gezeigt, wie sich diese Veränderungen auf die Partitionierung auswirken. Abschnitt 4.1.4 sowie 4.1.5 zeigen schließlich auf, wie eine parallele Mustersuche und Veränderung des Graphen vorgenommen werden kann.

4.1.1 Motivation

Um mehrere gefundene Treffer von Suchmustern in einem Arbeitsgraphen gleichzeitig zu verarbeiten, müssen diese parallel unabhängig sein. Die parallele Unabhängigkeit wurde bereits in 3.1.1.1 beschrieben. In diesem Abschnitt werden zwei Methoden vorgestellt, mit denen n gegebene Treffer von Suchmustern auf parallele Unabhängigkeit überprüft werden können. Diese beiden Verfahren skalieren jedoch nur schlecht mit einer steigenden Anzahl an Treffern. Daher wird mit der Partitionierung des Arbeitsgraphen eine Strukturierung vorgeschlagen, mit der eine bessere Skalierbarkeit erzielt werden kann.

4.1.1.1 Elementweises Vergleichen

Instanzen von Suchmustern sind parallel unabhängig, wenn nur ihre Klebegraphen (vergleiche 3.1.1.1) gemeinsame Elemente aufweisen. Somit kann die parallele Unabhängigkeit zweier Musterinstanzen durch Vergleich der Graphelemente festgestellt werden. Finden sich gemeinsame Graphelemente außerhalb der Klebegraphen, liegt ein Konflikt vor und die Ausführung muss sequentiell geschehen. Der Aufwand, dies festzustellen, kann groß werden: Einerseits steigt der Aufwand direkt mit der Komplexität der Regel selbst, andererseits mit der Anzahl an Musterinstanzen, die gleichzeitig verarbeitet werden sollen. Jede Musterinstanz muss mit jeder anderen auf gemeinsame Elemente außerhalb des Klebegraphen untersucht werden. Die Anzahl der durchzuführenden Tests $T(n)$ lässt sich somit durch den Binomialkoeffizienten bestimmen:

$$T(n) = \binom{n}{2} = \frac{n!}{2!(n-2)!}$$

Hierbei ist n die Anzahl der Prozessoren, also der gleichzeitig ausführbaren Regeln. Die verfügbaren Prozessoren können bereits hier parallel genutzt werden, da die einzelnen Tests voneinander unabhängig ausgeführt werden können. Die Anzahl der sequentiell durchzuführenden Tests $S(n)$ ergibt sich somit wie folgt:

$$S(n) = \lfloor \frac{\binom{n}{2}}{n} \rfloor$$

Hierbei ist jedoch zu beachten, dass die Verarbeitungsdauer der einzelnen Tests nicht zwangsläufig gleich lang sein muss: Sollen unterschiedliche Regeln parallel angewendet werden, beeinflusst dies auch die Laufzeit der Tests.

Ob sich eine parallele Ausführung der Tests lohnt, hängt von der Größe der Muster ab. Sind die Muster klein, so kann der Initialisierungsaufwand unverhältnismäßig hoch werden.

Unklar ist, mit welcher Wahrscheinlichkeit Musterinstanzen keine gemeinsamen Elemente außerhalb des Klebegraphen besitzen. Eine allgemeingültige Lösung kann hierfür nicht gegeben werden, da dies abhängig von der konkreten Anwendung ist. Allerdings sind grundsätzlich Situationen ungünstig, in denen Musterinstanzen gefunden wurden, deren Verarbeitung jedoch mit anderen Musterinstanzen in Konflikt steht: Dies bedeutet nicht nur, dass der zusätzliche Aufwand des Vergleichs der Klebegraphen nicht „belohnt" wird, sondern auch, dass eine bereits gefundene Instanz wieder verworfen werden muss.

4.1.1.2 Abstandsberechnung zwischen Musterinstanzen

Eine andere Variante zu entscheiden, ob zwei Musterinstanzen parallel unabhängig sind, besteht darin, ihren *Abstand* zu bestimmen. Was hierbei unter *Abstand* zu verstehen ist, wird im Folgenden geklärt.

Zwei Regelanwendungen können nach dem Parallelisierungstheorem gleichzeitig angewendet werden, wenn ihre Musterinstanzen höchstens im Klebegraphen gemeinsame Elemente besitzen. Das heißt, dass zwei Musterinstanzen insbesondere dann gleichzeitig angewendet werden können, wenn sie keine gemeinsamen Elemente besitzen. Sie besitzen sicherlich dann keine gemeinsamen Elemente, wenn sie in unterschiedlichen Bereichen des Arbeitsgraphen zu finden sind, die Orte der beiden Regelanwendungen sozusagen „weit voneinander entfernt" sind.

Um zu prüfen, ob zwei Musterinstanzen „weit voneinander entfernt" sind, werden zunächst zwei Begriffe eingeführt:

Definition 4.1.1.1 (Zentrum) *Wähle für die linke Seite einer Graphersetzungsregel einen (beliebigen) Knoten. Sowohl dieser Knoten als auch sein homomorphes Bild in einer Musterinstanz der Regel wird dann als Zentrum bezeichnet. Besitzt die linke Seite der Regel mehrere unverbundene Teilgraphen, so wird für jeden Teilgraphen ein Zentrum festgelegt.*

Definition 4.1.1.2 (Ausbreitung) *Die Ausbreitung ist die maximale Entfernung in der linken Seite einer Graphersetzungsregel, die ein Knoten vom Zentrum des Teilgraphs besitzt, zu dem er gehört. Die Ausbreitung ist somit pro Teilgraph definiert und abhängig vom gewählten Zentrum. Die Entfernung wird in Anzahl der zu überquerenden Kanten gemessen.*

Optimalerweise wird als Zentrum ein *Zentralpunkt*[1] der Teilgraphen gewählt. Die Ausbreitung entspricht dann dem Radius des Teilgraphen. Die Bestimmung der Zentren und der Ausbreitungswerte der einzelnen Regeln kann in einem Vorverarbeitungsschritt geschehen. Diese Werte sind konstant, da sich Graphersetzungsregeln während der Ausführung nicht ändern. Der Zusammenhang zwischen Zentrum und Ausbreitung ist in Abbildung 4.1 veranschaulicht.

Mit diesen Begriffen kann nun der Abstand zweier Musterinstanzen definiert werden:

[1] Ein Zentralpunkt eines zusammenhängenden Graphen G ist ein Knoten $v \in G$, dessen Abstand zu keinem anderen Knoten $w \in G$ größer ist, als der Radius von G.

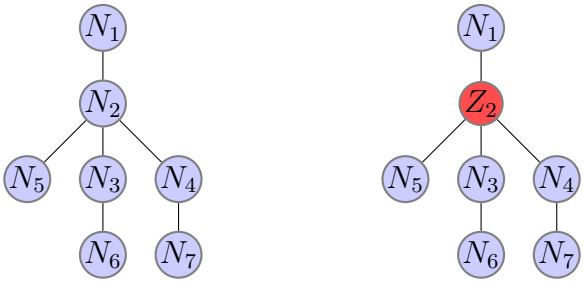

Abbildung 4.1: Links ist ein Suchmuster zu sehen. Rechts wurde für dieses Muster ein Zentrum gewählt. Es ist der Zentralpunkt des Graphen. Daher entspricht die Ausbreitung dem Radius des Graphen und hat den Wert 2.

Definition 4.1.1.3 (Abstand) *Der Abstand zweier Musterinstanzen zweier (nicht notwendigerweise gleicher) Regeln in einem Arbeitsgraphen ist definiert als die minimale Entfernung zwischen zwei Zentren der Musterinstanzen.*

Der Abstand lässt sich somit mittels einer Wegfindung von einem Zentrum zum anderen ermitteln. Ist der Abstand kleiner, als die Summe der Ausbreitung der beiden Regeln, so können die beiden Musterinstanzen keine gemeinsamen Elemente besitzen und sind somit parallel anwendbar.

Problematisch ist jedoch das Bestimmen des Abstandes: bekannte Algorithmen für den kürzesten Pfad zwischen zwei Knoten in einem Graphen sind Dijkstra sowie der schnellere A*-Algorithmus. A* bezieht seinen Geschwindigkeitsvorteil jedoch durch Heuristiken, die auf zusätzlichen Informationen über den Graphen basieren. Diese sind hier im Allgemeinen nicht vorhanden, so dass nur der Algorithmus von Dijkstra übrig bleibt. Dieser hat jedoch eine Laufzeit von $O(n * \log(n + m))$, wobei n die Anzahl der Knoten und m die Anzahl der Kanten des Graphen ist. Hinzu kommt genau wie beim elementweisen Vergleich, dass alle Musterinstanzen, die gleichzeitig abgearbeitet werden sollen, paarweise bezüglich ihres Abstandes überprüft werden müssen.

4.1.1.3 Höhere Skalierbarkeit durch Partitionierung

Das Problem der oben vorgestellten Ansätze ist der hohe Aufwand, um zu bestimmen, ob die gefundenen Vorkommen eines Musters gemeinsame Elemente besitzen oder nicht. Um dieses Problem schneller zu lösen, kann eine Partitionierung des Arbeitsgraphen verwendet werden. Hierbei wird der Graph in mehrere Partitionen zerteilt. Bei der Mustersuche wird dann protokolliert, in welcher Partition die Graphelemente einer gefundenen Musterinstanz liegen. Somit lässt sich überprüfen, ob zwei Instanzen eines Musters in der selben Partition liegen. Ist dies der Fall, besteht die Gefahr, dass sie gemeinsame Elemente besitzen. Die Ausführung der Regeln wird in diesem Fall sequentiell durchgeführt. Sind allerdings beide Vorkommen in verschiedenen Partitionen, so besitzen sie keine gemeinsamen Elemente und können gefahrlos gleichzeitig ausgeführt werden. Dieses Verfahren weist eine weit höhere Skalierbarkeit auf: Die Anzahl der nötigen Vergleiche, um festzustellen, ob zwei Musterinstanzen in derselben Partition liegen, ist unabhängig von der Größe des Musters. Eine detailierte Beschreibung der Regelausführung unter Verwendung von Partitionen wird in den folgenden Abschnitten dieses Kapitels vorgenommen.

4.1.2 Erzeugen von Graphpartitionen

Die Abschnitte 4.1.2.1 und 4.1.2.2 gehen zunächst auf einige Punkte bezüglich der Strukturierung einer Partitionierung ein. In 4.1.2.3 bis 4.1.2.5 werden anschließend verschiedene Methoden zum Erzeugen der Partitionen vorgestellt. Änderungen am Arbeitsgraphen sind bei der Graphersetzung nur natürlich: 4.1.3 beschreibt anschließend, wie eine Partitionierung hiermit umgehen kann.

4.1.2.1 Struktur der Partitionierung

Grundlegend stellt sich die Frage, ob lediglich die Knoten, die Kanten oder beides partitioniert werden sollen. Da zum Löschen und Erzeugen von Kanten bei den meisten Implementierungen Schreibzugriff auf beide Knoten notwendig ist, kann auf eine Partitionierung der Kanten verzichtet werden. Das Sperren beider Endknoten einer Kante kommt dem Sperren der Kante selbst gleich.

Der Verzicht auf eine expliziten Partitionszuordnung der Kanten kann den Speicherbedarf deutlich reduzieren, lediglich die Anzahl der Knoten beeinflusst den Speicherbedarf der Partitionsverwaltung.

Einen Spezialfall stellt die Suche nach einer Kante ohne Berücksichtigung der Randknoten dar: Dies würde nicht zum Sperren einer Partition führen. Wird die Kante gelöscht, müssen die Partitionen der Endknoten gesperrt werden. Liegt die Kante lediglich im Klebegraphen der Regel, so ist das Sperren überflüssig: Auf die Kante wird dann nur lesend zugegriffen. Einen Problemfall stellt allerdings das Bearbeiten der Eigenschaften einer solchen Kante dar: Dies könnte von mehreren Regeln gleichzeitig geschehen, ohne die Endknoten mitzusperren. Somit würde eine Wettlaufsituation entstehen. Dies kann vermieden werden, indem auch zur Bearbeitung der Kanteneigenschaften beide Endknoten (bzw. deren Partitionen) gesperrt werden müssen.

Es zeigt sich somit, dass mit wenigen Anpassungen auf die Partitionierung der Kanten verzichtet werden kann. Im Folgenden wird davon ausgegangen, dass lediglich die Knoten partitioniert werden.

4.1.2.2 Datenstruktur und Zugriff auf Graphpartitionen

Um Graphpartitionen zu implementieren, muss zunächst deren Struktur festgelegt werden. In diesem Abschnitt werden verschiedene Möglichkeiten zur Umsetzung aufgezeigt. Es wird davon ausgegangen, dass Knoten jeweils als separate Objekte einer gemeinsamen (Basis-)Klasse vorliegen. Andere, häufig verwendete Darstellungen wie Adjazenzmatrizen werden ausgeschlossen. Dies geschieht auch in Hinblick auf die spätere Umsetzung für GrGen.NET, wo Knoten wie auch Kanten als eigenständige .NET-Objekte umgesetzt werden.

Partitions-IDs

Eine Lösung, um Graphpartitionen zu repräsentieren, besteht darin, die Knotenklasse um ein ID-Feld zu erweitern, dass die ID der Partition speichert, zu der der Knoten gehört. Somit kann für einen gegebenen Knoten in $O(1)$ ermittelt werden, zu welcher Partition er gehört. Allerdings steigt der Speicherbedarf der Knoten um `k * sizeof(ID)`. Möchte man die Menge aller zu einer gegebenen Partition gehörenden Knoten bestimmen, so ist hierzu das Iterieren über alle Knoten notwendig.

Partitions-Objekte

Alternativ kann für jede Graphpartition ein Objekt angelegt werden, welches Referenzen auf alle enthaltenen Knoten enthält. Der Speicherbedarf ist hierbei nahezu identisch, hinzu kommt lediglich der Speicherbedarf der Partitionsobjekte[2]. Somit lassen sich leicht alle Elemente finden, die zu einer gegebenen Partition gehören. Das ermitteln der Partition, zu der ein gegebener Knoten gehört, ist dafür jedoch nur noch in $O(n)$ durchzuführen, da alle Knoten in allen Partitionen überprüft werden müssen.

Getrennte Knotenverwaltung

Die beiden obigen Ansätze gehen davon aus, dass alle Knoten unabhängig von ihrer Partition in einer gemeinsamen Datenstruktur abgelegt sind. Allerdings könnte solch eine Struktur auch pro Partition vorgehalten werden. Somit wären keine zusätzlichen Elementzeiger im Partitions-Objekt notwendig. Zu welcher Partition ein gegebener Knoten gehört, liese sich dann schneller feststellen: Werden die Elemente einer Partition beispielsweise in einer doppelt-verketteten Liste verwaltet, so könnte der Listenkopf die zugehörige Partitions-ID speichern. Somit liese sich für einen gegebenen Knoten in $O(\frac{n}{p})$ ermitteln, zu welcher Partition dieser gehört, wobei n die Anzahl der Knoten und p die Anzahl der Partitionen ist[3]. Dies ist in Abbildung 4.2 dargestellt.

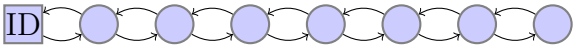

Abbildung 4.2: Eine Liste, wobei der Listenkopf die Partitions-ID speichert. Wird für ein Element der Liste die Parititions-ID benötigt, kann über die verkettete Liste der Kopf gefunden werden.

4.1.2.3 Zufällige Partitionen

Die einfachste Möglichkeit, einen Graphen zu partitionieren, besteht darin, die Knoten des Graphen zufällig auf p Partitionen zu verteilen. Dieses Vorgehen ist in Abbildung 4.3 dargestellt:

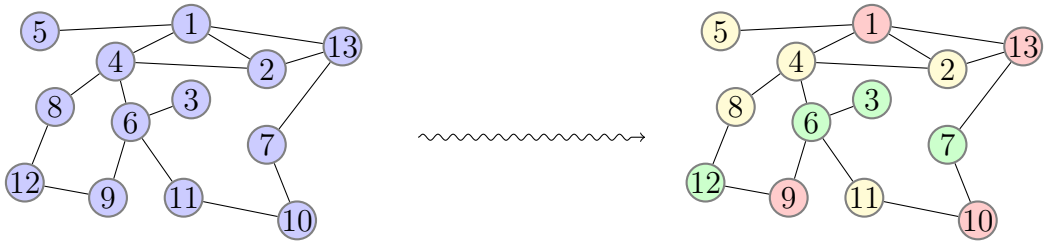

Abbildung 4.3: Zufällige Einteilung eines Graphen in Partitionen. Jede der drei Knotenfarben im rechten Graphen entspricht einer Partition.

[2]Unter der Annahme, dass das Speichern einer Partitions-ID im Knoten genauso viel Platz beansprucht, wie eine Referenz auf einen Knoten im Partitions-Objekt.

[3]Wobei davon ausgegangen wird, dass die n Knoten gleichmäßig auf die p Partitionen verteilt sind

Dieses Verfahren ist in $O(n)$ durchführbar, n ist hierbei die Anzahl der Knoten des Graphen. Allerdings ist diese Aufteilung für die praktische Graphersetzung meist ungenügend: Suchmuster bestehen meist aus mehreren, zusammenhängenden Knoten. Besteht ein Suchmuster S aus k Knoten und wurde der Arbeitsgraph in p Partitionen aufgeteilt, so liegt die Wahrscheinlichkeit, dass sich alle Knoten in derselben Partition befinden, bei $p_S = (\frac{1}{p})^{k-1}$. Die meisten Musterinstanzen werden sich daher über mehrere Partitionen erstrecken, was die Anzahl der gleichzeitig ausführbaren Musterinstanzen deutlich einschränkt.

4.1.2.4 Das Rubini-Verfahren

Damit Musterinstanzen über möglichst wenige Partitionen verteilt sind, kann deren lokaler Zusammenhang ausgenutzt werden. Hierzu müssen Partitionen erzeugt werden, in denen möglichst viele Knoten miteinander verbunden sind, so dass eine Partition einem zusammenhängenden Teilgraphen des Arbeitsgraphen entspricht.

In Rahmen dieser Arbeit wurde ein Partitionierungsverfahren entwickelt, dass diese Eigenschaft berücksichtigt. Es leitet sich von den *Rubini-Graphen* ab. Diese übertragen das Prinzip der Raumpartitionierung der *Voronoi-Diagramme*[4] auf eine endliche Menge von Objekten.

Gegeben sind zwei Mengen von Punkten in einem euklidischen Raum, wovon die Elemente der ersten Menge als *Ankerpunkte* bezeichnet werden. Zur Erzeugung des Rubini-Graphen wird nun von jedem Nicht-Ankerpunkt aus eine Kante zu demjenigen Ankerpunkt erzeugt, zu dem er den geringsten euklidischen Abstand besitzt. Dies ist in Abbildung 4.4 dargestellt.

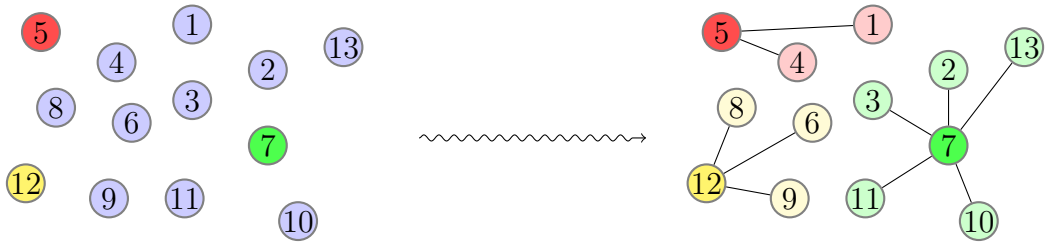

Abbildung 4.4: Partitionierung einer Punktwolke nach dem Rubini-Verfahren mit euklidischem Maß. Die Ankerpunkte sind farblich hervorgehoben.

Es ergibt sich ein Graph, welcher aus einer Menge von unverbundenen Teilgraphen besteht und somit eine natürliche Partitionierung besitzt.

Die Rubini-Graphen stammen aus dem Bereich der geographischen Informationssysteme (GIS)[gfk]. Hier werden sie verwendet, um Kundenagglomerationen und die Effizienz von Vertriebsstrukturen graphisch darzustellen. Hierbei werden meist die Positionen von Filialen oder Außendienstmitarbeitern als Ankerpunkte verwendet, die übrigen Punkte entsprechen den (potentiellen) Kunden.

Dieses Prinzip der Objektpartitionierung lässt sich auf Graphen übertragen: Hierbei wird eine Teilmenge der Knoten als Ankerpunkte ausgezeichnet. Nun werden

[4]Ein Voronoi-Diagramm ist eine Gebietseinteilung eines kontinuierlichen Raumes. Es sind n Ankerpunkte in diesem Raum gegeben. Der Raum wird in n Partitionen aufgeteilt, die jeweils den einzelnen Ankerpunkten zugeteilt sind. Jeder Punkt in diesem Raum wird derjenigen Partition zugeordnet, zu deren Ankerpunkt er den geringsten Abstand hat.

die übrigen Knoten jeweils genau demjenigen Ankerpunkt zugeordnet, der mit der geringsten Anzahl an Schritten über Kanten erreichbar ist. Jeder Ankerpunkt bildet nun zusammen mit den ihm zugeordneten Knoten eine Partition des Graphen. Diese Partitionierungsmethode wird im Folgenden *Rubini-Verfahren* genannt. Auf der linken Seite von Abbildung 4.5 ist ein unpartitionierter Graph zu sehen. Die rechte Seite zeigt das Ergebnis einer Partitionierung nach dem Rubini-Verfahren.

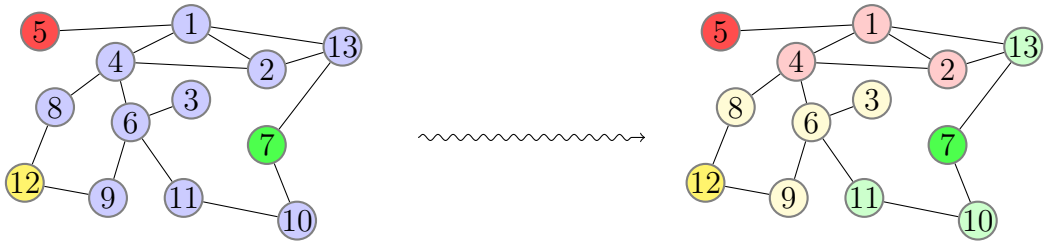

Abbildung 4.5: Partitionierung eines Graphen nach dem Rubini-Verfahren.

Erzeugen von Rubini-Graphen

Wie in Abbildung 4.5 zu sehen ist, bieten Rubini-Graphen eine günstigere Partitionierung als die in 4.1.2.3 beschriebene zufällige Einteilung. Nun soll geklärt werden, wie ein Rubini-Graph effizient erzeugt werden kann. Hierbei wird vorausgesetzt, dass die Ankerpunkte bereits bekannt sind.

Die Lösung besteht darin, von jedem Punkt aus eine Wegfindung zu allen Ankerpunkten durchzuführen und somit den nächsten Ankerpunkt zu bestimmen. Dieses Verfahren wäre praktisch jedoch nicht tragbar: In Abschnitt 4.1.1.2 wurde bereits gezeigt, dass Wegfindungsalgorithmen einen zu hohen Aufwand besitzen, um in der Graphersetzung angewendet zu werden. Es lässt sich jedoch ein Algorithmus konstruieren, der keine Wegfindung benötigt. Hierbei wird von jedem Ankerpunkt aus eine Breitensuche nach den Knoten durchgeführt. Der Algorithmus ist im Folgenden in Pseudocode-Notation angegeben:

1. Lege für jeden Ankerpunkt zwei Mengen an: `open` und `closed`.
2. Füge jeden Ankerpunkt in seine `open`-Liste ein.
3. Solange nicht alle `open`-Listen leer sind:
   (a) Führe für jede `open`-Liste aus:
      i. Führe für jedes Element das aktuell in der `open`-Liste liegt durch:
         A. Verschiebe das Element in die zugehörige `closed`-Liste.
         B. Lege alle mit dem Element verbundenen Knoten, die noch nicht einem Ankerpunkt zugeordnet wurden, in die `open`-Liste und markiere sie als zugeornet.

Sobald dieser Algorithmus durchgeführt wurde, ist jeder Knoten seinem nächsten Ankerpunkt zugeordnet. Die `closed`-Listen enthalten dann jeweils alle Knoten einer Partition. Diese simultane Breitensuche lässt sich parallelisieren, indem Schritt 3.(a) für alle `open`-Listen gleichzeitig durchgeführt wird. Hierbei muss allerdings beachtet

werden, dass dann mehrere Fäden diejenigen Knoten gleichzeitig finden werden, die von den zugehörigen Ankerpunkten den gleichen Abstand haben. Somit muss das „in Besitz nehmen" eines Knotens synchronisiert werden. Die einzelnen Schritte des so parallelisierten Verfahrens sind in Abbildung 4.6 dargestellt. Im zweiten Schritt von 4.6 ist zu sehen, dass Knoten Nummer 4 sowohl vom roten als auch vom gelben Ankerpunkt gleich weit entfernt ist und beide diesen Knoten im nächsten Schritt in ihre Knotenmenge aufnehmen könnten. In welche Menge der Knoten eingefügt wird, ist indeterministisch und hängt von der genauen Ausführungsreihenfolge der Fäden ab. Somit ist durch die Parallelisierung auch der erzeugte Rubini-Graph indeterministisch.

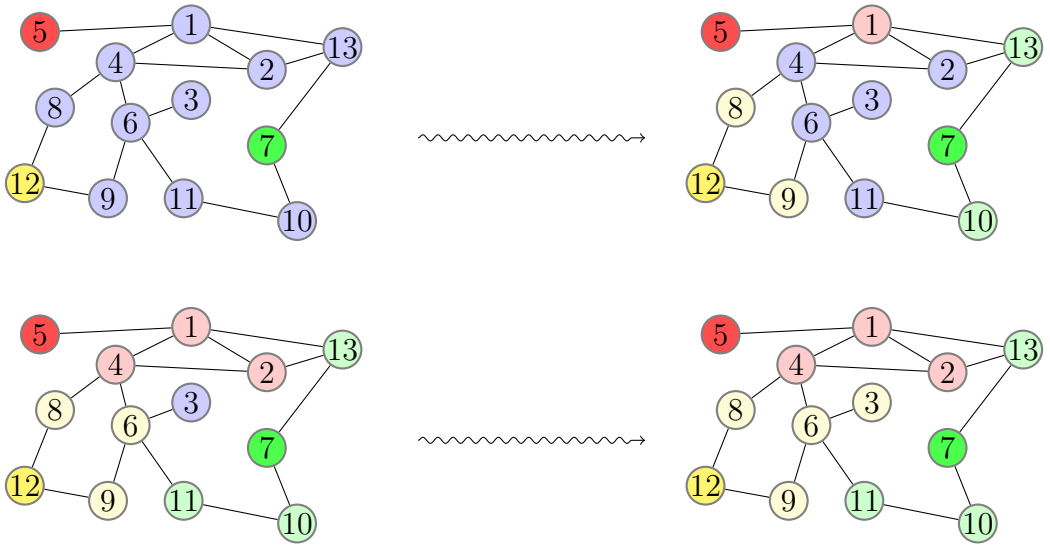

Abbildung 4.6: Partitionierung eines Graphen nach dem parallelen Rubini-Verfahren.

Auswahl der Ankerpunkte

Die Qualität der durch das Rubini-Verfahren erzeugten Partitionierung wird durch die Wahl der Ankerpunkte beeinflusst. Ideal wäre eine Ankerpunktbelegung, die einen „großen" Abstand zwischen die einzelnen Punkte bringt. Hierzu wären allerdings strukturelle Informationen über den Arbeitsgraphen notwendig, die im Normalfall jedoch nicht zur Verfügung stehen. Eine Möglichkeit, solche Informationen zu gewinnen, wäre die Verwendung von Graph-Layout-Algorithmen, die allerdings selbst sehr komplex sind und somit dem gewünschten Effekt, der Verkürzung der Ausführungszeit durch Parallelisierung, entgegenwirken.

Durch eine zufällige Auswahl können Ankerpunkte mit geringem Rechenaufwand erzeugt werden. Dies ist vertretbar, da die Anzahl der Partitionen (und somit auch der Ankerpunkte) klein im Verhältnis zur Zahl der Knoten sein sollte. Somit ist unwahrscheinlich, dass mehrere Ankerpunkte besonders nahe aneinander liegen. Natürlich ist der Fall einer ungünstigen Knotenbelegung nicht auszuschließen und kann daher zu negativen Ausreißern bei der Ausführungsgeschwindigkeit führen.

Eine Alternative zu einer rein zufallsbasierten Belegung wäre die Verwendung leicht berechenbarer Heuristiken. Beispielsweise könnten Knoten, die außergewöhnlich viele Kanten besitzen, bevorzugt als Ankerpunkt verwendet werden. Auch Knotentypen, die nur wenige Instanzen besitzen, kommen in Frage. Um hier konkrete Aussagen

treffen zu können, sind weitergehende Untersuchungen nötig, die im Rahmen dieser Arbeit nicht geleistet werden können. Daher wird auch bei der Implementierung ausschließlich die zufallsbasierte Bestimmung der Ankerpunkte verwendet.

Nicht-zusammenhängende Graphen

Die zufällige Belegung der Ankerpunkte kann bei nicht-zusammenhängenden Graphen zu Problemen führen: Es ist nicht gewährleistet, dass solch eine Belegung einen Ankerpunkt auf jedem Teilgraphen des Arbeitsgraphen erzeugt. Der oben angegebene Algorithmus würde in diesem Fall nicht alle Knoten erfassen. Ein Algorithmus für die Breitensuche, der alle Teilgraphen erfasst, ist in [Sedg88] gegeben. Hierbei werden in einer Schleife alle Knoten abgelaufen und für jeden Knoten eine Breitensuche gestartet, sofern dieser nicht bereits in einer vorhergehenden Suche als gefunden markiert wurde[5].

Dieser Ansatz lässt sich bei der Erzeugung von Rubini-Graphen in veränderter Form anwenden: Nach der Generierung des Rubini-Graphen wird sequentiell für alle Knoten überprüft, ob sie sich in einer Partition befinden. Ist dies für einen Knoten nicht der Fall, wird dieser als zusätzlicher Ankerpunkt markiert und eine Rubini-Suche an diesem Knoten gestartet. Nachdem diese Suche beendet ist, wird die Überprüfung fortgesetzt.

Ist bekannt, wie viele Knoten der Arbeitsgraph insgesamt besitzt, so lässt sich berechnen, ob alle Knoten einer Partition zugeteilt wurden. Dann gilt:

$$k = \sum_{i=1}^{p} k_i$$

Wobei k die Anzahl der Knoten im Arbeitsgraphen, p die Anzahl der Partitionen und k_i die Anzahl Knoten der i-ten Partition ist. Wenn diese Bedingung nach der Rubini-Suche gilt, muss nicht nach unerfassten Knoten gesucht werden und der Arbeitsgraph enthält keine unentdeckten Subgraphen.

4.1.2.5 Weitere Partitionierungsmethoden

Es existieren verschiedene Algorithmen, um Graphen zu partitionieren. Vor allem die Datenverarbeitung auf verteilten Systemen hat die Entwicklung solcher Algorithmen motiviert: Teilprobleme und deren Abhängigkeiten werden als Graph dargestellt. Eine Partitionierung dieses Graphen wird dann verwendet, um Teilprobleme den einzelnen Rechenknoten zuzuordnen. Diese Verfahren basieren jedoch üblicherweise auf einem detailierten Wissen über die Struktur des Graphen: geometrische Algorithmen beispielsweise machen Gebrauch von Koordinatensystemen, auf die der Graph abgebildet wird. Ein häufig verwendeter Algorithmus wird in [KeLi70] beschrieben. In weiterführenden Arbeiten kann untersucht werden, wie sich solche Algorithmen zur Partitionierung in der Graphersetzung verwenden lassen.

[5] Die Knoten müssen hierfür in einer sequentiellen Datenstruktur wie einem Feld oder einer Liste vorliegen.

4.1.3 Wartung der Partitionen

Durch die Anwendung von Graphersetzungsregeln verändert sich der Arbeitsgraph bezüglich Anzahl und Anordnung der Knoten und Kanten. Dies wirkt sich auch auf die Partitionen des Arbeitsgraphen aus: Sind bestimmte Partitionen von den Änderungen mehr betroffen als andere, ergibt sich ein Ungleichgewicht, was die Leistungsfähigkeit der Partitionierung negativ beeinflussen kann. In Extremfällen können einzelne Partitionen komplett „verschwinden" oder sämtliche Aktionen nur noch in einer einzigen Partition stattfinden.

In diesem Abschnitt werden einige Möglichkeiten aufgezeigt, wie diesem Problem begegnet werden kann.

Erzeugen von Knoten

Werden neue Knoten erzeugt, so stellt sich die Frage, welcher Partition diese zugeordnet werden. Neue Knoten werden meist direkt nach dem Erstellen mit anderen Knoten verbunden. In diesem Fall kann der neue Knoten in die Partition eines verbundenen, älteren Knotens eingefügt werden. Dies ist in Abbildung 4.7 dargestellt.

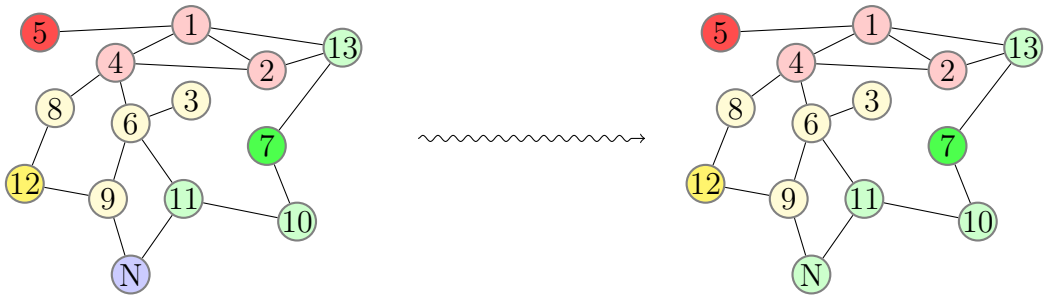

Abbildung 4.7: Einfügen eines neuen Knoten N. Dieser wird einer Partition zugeordnet, zu der er eine Verbindung besitzt.

Wie in Abbildung 4.7 zu sehen ist, wird der neue Knoten N der grünen Partition zugeordnet. Allerdings besitzt er auch eine Kante zur gelben Partition und liegt näher an deren Ankerpunkt. Daher wäre es sinnvoll, den neuen Knoten der gelben Partition zuzuordnen. Um diese Information zur Laufzeit zu erhalten, ist jedoch eine Wegfindung zu den Ankerpunkten der mit dem neuen Knoten verbundenen Partitionen notwendig. Dies ist während eines laufenden Graphersetzungsvorgangs jedoch zu teuer. Da das entstehende Ungleichgewicht an der Datenstruktur nur gering ist, bietet es eine hinreichende Annäherung, neue Knoten zufällig einer verbundenen Partition zuzuordnen. Falls die Anzahl der Elemente einer Partition in $O(1)$ ermittelt werden kann, ist es auch möglich, den neuen Knoten der kleinsten Partition zuzuordnen.

Wird der neue Knoten nicht direkt mit einem anderen Knoten (oder nur mit ebenfalls neu erzeugten Knoten) verbunden, so kann dieser in eine eigens für solche, nichtzuordenbare Knoten angelegte Partition eingefügt werden. Diese spezielle Partition enthält dann im Laufe der Zeit viele echte Subgraphen, die in späteren Konsolidierungsschritten auf Verbindungen zu „echten" Partitionen untersucht und dann neu zugeordnet werden können.

Teilen von Partitionen

Wächst eine Partition zu stark an, so muss sie in mehrere kleinere Partitionen unterteilt werden. Hierzu kann eine *lokale Rubini-Suche* verwendet werden: Das Verfahren funktioniert genauso wie oben beschrieben, allerdings wird nicht der ganze Graph, sondern ausschließlich die zu große Partition als Eingabe verwendet. Die alte Partition wird aufgelöst, innerhalb deren Elemente neue Ankerpunkte definiert und dann die Rubini-Suche gestartet. Knoten, die bereits eine Partition besitzen, dienen als Grenze der Suche und werden nicht in die neuerliche Suche miteinbezogen. Darüber hinaus ist es möglich, solch eine Neuaufteilung parallel zu normalen Graphersetzungsaktionen in anderen Partitionen des Graphen durchzuführen, so dass der grundlegende Fluss der Abarbeitung nicht unterbrochen werden muss. Das Verfahren ist in Abbildung 4.8 dargestellt.

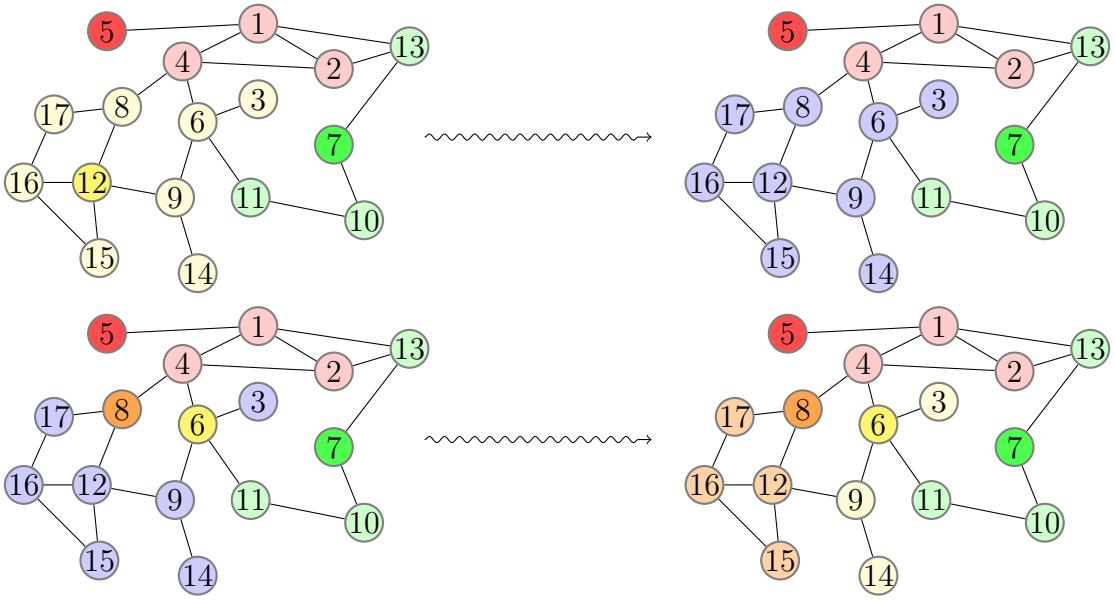

Abbildung 4.8: Aufteilen einer Partition durch eine lokale Rubini-Suche. Zunächst wird die betroffene Partition aufgelöst. Anschließend werden neue Ankerpunkte gewählt. Nach einigen (parallelen) Arbeitsschritten sind die neuen Partitionen gefunden.

Zusammenführen von Partitionen

Werden Partitionen durch das Löschen von Knoten sehr klein, kann es vorkommen, dass Instanzen von Suchmustern verhältnismäßig viele Partitionen belegen. Dann kann es von Vorteil sein, solche kleinen Partitionen zu vereinigen. Hierzu kann die Anzahl der Elemente herangezogen werden[6]: fällt diese unter eine vorgegebene Grenze, so ist diese Partition ein Kandidat für eine Vereinigung.

Das Zusammenführen zweier Partitionen A und B kann geschehen, indem alle Elemente von B als Elemente der Partition A markiert werden und eventuell vorhandene Datenstrukturen der Partition B aufgelöst werden.

Genau wie das Aufteilen von Partitionen kann auch das Zusammenführen parallel zu den Graphersetzungsprozessen in anderen Partitionen stattfinden.

[6]Zusätzlich kann die Anzahl der Verbindungen zwischen den Knoten einer Partition verwendet werden.

Neuberechnung

Sowohl das Aufteilen als auch das Zusammenführen von Partitionen sind gut geeignet, um schnell die Struktur der Partitionierung an lokalen Brennpunkten zu verbessern. Dies kann im Laufe der Zeit jedoch zu einer unausgeglichenen Partitionierung führen, da die Partitionierung nur auf den Verknüpfungen der lokalen Elemente basiert. Daher kann es sinnvoll sein, die Partitionierung in gewissen Abständen komplett aufzulösen und neu zu erzeugen. Zeiten, in denen der Anwender Ergebnisse beobachtet, wären optimal hierfür, da die Regelanwendung in diesem Fall nicht verzögert wird. Dies kann nur geschehen, wenn die Anwendung mit einer graphischen Benutzerschnittstelle ausgestattet ist und somit durch den Anwender hervorgerufene Ruhephasen besitzt.

4.1.4 Mustersuche auf Graphpartitionen

In diesem Abschnitt wird erläutert, wie eine parallele Suche nach Instanzen von Mustern auf einem partitionierten Arbeitsgraphen durchgeführt werden kann.

Eine naheliegende Variante der Parallelisierung bei einer vorhandenen Partitionierung ist die getrennte Suche pro Partition: Jeder Thread führt den Suchplan auf seiner lokalen Partition aus. Dieses Vorgehen ist in Abbildung 4.9 dargestellt.

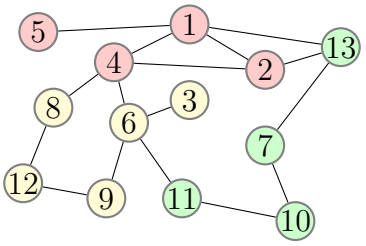

Abbildung 4.9: Ein partitionierter Graph. Jeder Partition ist ein eigener Thread zugeordnet.

Problematisch ist diese Form der Suche an den Übergängen[7] zwischen den Partitionen. Hierzu gehören beispielsweise Kanten, bei denen die Endknoten in verschiedenen Partitionen liegen. Dieses Problem wird in Abbildung 4.10 veranschaulicht. Auf der linken Seite der Abbildung ist die linke Seite einer Regel zu sehen, die einen Knoten der Klasse A sucht, der mit einem Knoten der Klasse B verbunden ist. Auf der rechten Seite ist ein Ausschnitt eines Arbeitsgraphen zu sehen. Die Partitionen sind farblich hervorgehoben.

Wird die Suche über die Partitionen verteilt ausgeführt, so kann die in Abbildung 4.10 sichtbare Musterinstanz nicht gefunden werden, da die Elemente über zwei Partitionen verteilt sind. Dies wird erst möglich, wenn die Suche dahingehend erweitert wird, dass ein Faden mit der Suche nur in einer ihm zugewiesenen Partition beginnt, sich aber durchaus Elemente der gefundenen Musterinstanz in anderen Partitionen befinden dürfen. Beispielsweise könnte für das Muster aus Abbildung 4.10 der Suchplan aus Abbildung 4.11 vorgesehen sein.

[7] Die Übergänge werden im Folgenden als zweidimensionale Grenze dargestellt, die genannten Eigenschaften gelten jedoch unabhängig von der Art und Weise der Darstellung und insbesondere auch bei höheren Dimensionen.

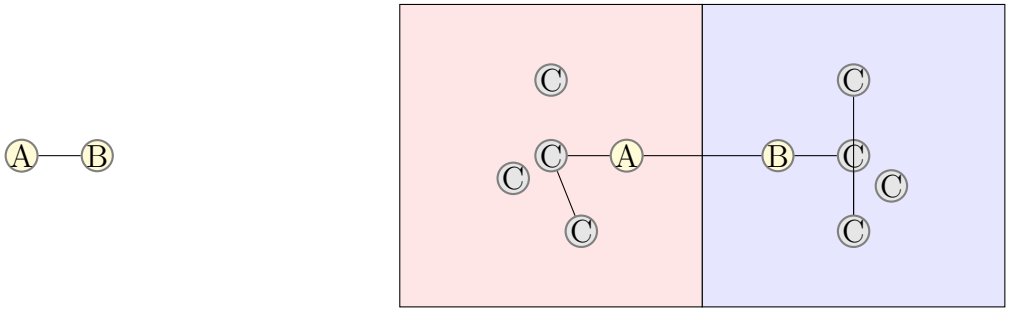

Abbildung 4.10: Die linke Seite zeigt ein Suchmuster. Rechts ist ein aus zwei Partitionen bestehender Arbeitsgraph zu sehen, der eine partitionsübergreifende Instanz des Musters enthält.

```
Finde eine Instanz a der Knotenklasse A
Finde eine Kante, die a mit einem Knoten der Klasse B verbindet
```

Abbildung 4.11: Suchplan für das Muster aus Abbildung 4.10.

Ermöglicht ein Knoten den Zugriff auf alle Kanten (und deren Endknoten) des Musters partitionsübergreifend, so würde die Musterinstanz trotz der Partitionierung gefunden werden. Dieser Suchansatz besitzt jedoch selbst wiederum ein Problem: Angenommen, ein Suchmuster hat einen symmetrischen Aufbau wie in Abbildung 4.12:

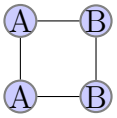

Abbildung 4.12: Ein Suchmuster mit symmetrischem Aufbau.

Überschreitet eine Instanz dieses Suchmusters nun die Grenze zweier (oder mehr) Partitionen, so kann diese Instanz mehrfach gefunden werden. Dies ist in Abbildung 4.13 dargestellt.

Ein doppeltes Auffinden einer einzigen Instanz eines Suchmusters ist nur schwer zu verhindern[8]. Einfacher ist das spätere Auffinden und Aussortieren solcher Doppelgänger. Hierzu muss für jeden Treffer eines Suchmusters vermerkt werden, in welchen Partitionen sich dessen Elemente befinden. Ergeben sich mehrere Treffer, die Elemente in mehreren, paarweise identischen Partitionen besitzen, so müssen die einzelnen Elemente der Musterinstanz auf Elementgleichheit überprüft werden. Besitzen beide Instanzen dieselben Elemente, so sind sie identisch.

Ein weiteres Problem besteht bei Suchmustern, die nicht zusammenhängend sind. Ein Beispiel ist in Abbildung 4.14 zu sehen. Bei diesem Muster wird nach zwei jeweils durch Kanten verbundenen Knotenpaaren gesucht, ein Paar hat dabei den Typ A, das andere den Typ B.

Falls sich die beiden Knotenpaare in verschiedenen Partitionen befinden, kann diese Instanz des Suchmusters nicht aufgefunden werden. Insbesondere hilft hier auch nicht

[8] Bei der *ApplyAll*-Regelanwendung in GrGen.NET beispielsweise werden solche doppelten Treffer nicht speziell behandelt und normal weiterverarbeitet.

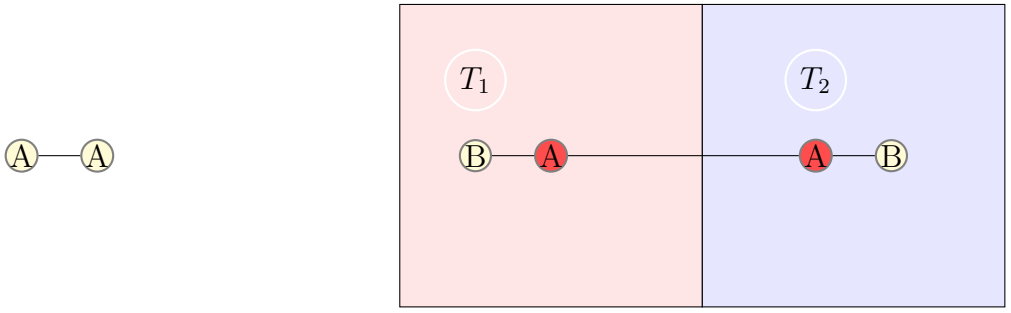

Abbildung 4.13: Das links dargestellte Muster hat im (partitionierten) Arbeitsgraphen rechts eine Musterinstanz. Diese wird bei einer parallelen Suche jedoch mehrfach gefunden.

Abbildung 4.14: Ein Suchmuster, das aus zwei unverbundenen Teilen besteht.

die Möglichkeit, in „fremde" Partitionen einzudringen, dies funktioniert nur, wenn eine Kante als „Brücke" in die andere Partition dient. Das Problem ist in Abbildung 4.15 veranschaulicht.

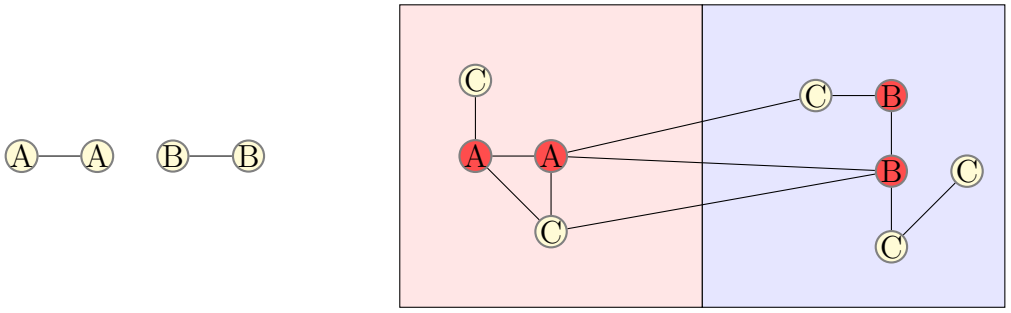

Abbildung 4.15: Suche nach dem nicht-zusammenhängenden Muster auf der linken Seite. Beide den Partitionen zugeteilte Threads können hier nur die Hälfte des Musters erkennen.

Solche Instanzen von Suchmustern können bei einer Suche auf Partitionen nur aufgefunden werden, wenn das Suchmuster vorher analysiert wurde: Ist bekannt, dass das Muster nicht-verbundene Elemente enthält, so lässt sich das Muster dementsprechend aufteilen und eine Suche nach den partiellen Mustern durchführen. Besteht das Suchmuster aus n unverbundenen Teilen, so werden n partielle Ergebnismengen gefunden. Die tatsächliche Ergebnismenge besteht dann aus dem n-fachen Kreuzprodukt der partiellen Ergebnismengen.

4.1.5 Elementaroperationen bei Graphpartitionen

Im vorhergehenden Abschnitt wurde gezeigt, wie auf einem partitionierten Graphen Instanzen von Suchmustern aufgefunden werden können. Diese sollen anschließend auch parallel gegen die rechte Seite der entsprechenden Regel ersetzt werden. In diesem Abschnitt werden die einzelnen Elementaroperationen aufgelistet, mit denen die linke Seite einer Graphproduktion in die rechte Seite überführt wird. Betrachtet werden deren Auswirkungen auf die Partitionen eines Graphen, beziehungsweise ihre Ausführbarkeit unter der Verwendung von Partitionen.

Kanten erzeugen

Beim Erzeugen von Kanten zwischen zwei Knoten muss beachtet werden, dass beide Knoten gesperrt sein müssen. Hierzu müssen die Partitionen beider Knoten erkannt und gesperrt werden. Neue Kanten können Partitionen verbinden, die zuvor keine Verbindung hatten. Werden viele Kanten erzeugt, kann somit das Neuberechnen der Partitionierung von Vorteil sein.

Kanten löschen

Das Löschen von Kanten wird in den meisten Implementierungen Schreibzugriff auf beide Knoten der Kante erfordern. Daher müssen zur Ausführung einer Musterinstanz die Partitionen beider Knoten gesperrt werden. Das Löschen von Kanten kann innerhalb von Partitionen echte Teilgraphen erzeugen oder komplette Partitionen vom restlichen Graphen separieren.

Knoten erzeugen

Das Erzeugen von Knoten kann ohne Sperren übriger Objekte erfolgen. Welcher Partition ein neuer Knoten hinzugefügt wird, wird in 4.1.3 behandelt. Häufige Knotenerzeugungen können Partitionen unverhältnismäßig erweitern und ein Aufspalten dieser Partition erfordern. Auch dies wird in 4.1.3 beschrieben.

Knoten löschen

Das Löschen von Knoten ist eine der problematischsten Elementaroperationen: Insbesondere sticht hier der SPO-Ansatz hervor, der auch bei GrGen.NET verwendet wird. Hierbei ist es erlaubt, Knoten zu löschen, die noch Kanten zu anderen Knoten besitzen. Insbesondere müssen solche Kanten nicht erfasst werden, also nicht in der linken Seite der Regel gebunden werden. Benötigt das Löschen einer Kante, wie oben beschrieben, Schreibzugriff auf beide Knoten, so müssen beim Löschen eines Knotens alle verbundenen Knoten ebenfalls gesperrt werden, um die Kanten zu entfernen. Da ein Knoten Kanten zu beliebig vielen anderen Knoten besitzen kann, können bei Knoten mit hohem Kantengrad mehrere zusätzliche Partitionen betroffen sein, die nicht Teil der Musterinstanz sind. Diese Partitionen müssen für den Löschvorgang ebenfalls gesperrt werden, was den maximalen Parallelitätsgrad solcher Operationen deutlich einschränken kann. Abbildung 4.16 veranschaulicht die Problematik.

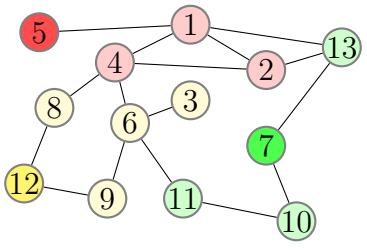

Abbildung 4.16: Soll in diesem Graphen der Knoten Nummer 6 gelöscht werden, müssen zusätzlich zur gelben Partition auch noch die rote und die grüne gesperrt werden, da der Knoten Kanten zu beiden Partitionen besitzt.

Graphelemente umtypisieren

Beim Umtypisieren von Graphelementen ändert sich die Struktur des Graphen nicht. Je nach gewählter Implementierung können trotzdem Operationen anfallen, die eine Synchronisierung erzwingen: Werden Graphelemente in einer Datenstruktur nach ihrem Typ geordnet abgelegt, so entspricht das Umtypisieren dem Entfernen und Einfügen eines Graphelementes. Wird der Typ lediglich in einem Feld des Objektes gespeichert, welches das Graphelement repräsentiert, ist keine objektübergreifende Synchronisierung notwendig. Der Aufwand des Umtypisierens hängt somit stark von der gewählten Datenstruktur ab.

Eigenschaften von Graphelementen neu auswerten

Die Neuberechnung der Eigenschaften von Graphelementen kann ohne größeren Aufwand vorgenommen werden. Ausdrücke, die hierzu ausgeführt werden, beeinflussen ausschließlich Elemente der Musterinstanz. Daher müssen keine weiteren Partitionen gesperrt werden. Da die Änderungen nur lokal am Graphelement vorgenommen werden, ist auch keine weitere gegenseitige Beeinflussung möglich, die synchronisiert werden müsste.

4.2 Weitere Methoden zur Parallelisierung

Im letzten Abschnitt wurde gezeigt, wie ein Graphersetzungssystem durch eine Partitionierung des Arbeitsgraphen profitieren kann. In diesem Abschnitt werden Parallelisierungsansätze aufgezeigt, die ohne eine Partitionierung des Arbeitsgraphen auskommen.

In Abschnitt 4.2.1 wird erklärt, wie die Suche nach einer Instanz eines Suchmusters parallel durchgeführt werden kann. In 4.2.2 und 4.2.3 wird die Suche nach mehreren Instanzen eines, beziehungsweise mehrerer Muster thematisiert.

4.2.1 Parallele Suche nach einem Treffer eines einzelnen Suchmusters

In diesem Abschnitt wird zunächst beschrieben, wie eine sequentielle Suche nach Musterinstanzen durchgeführt werden kann. Ob dieser Prozess bei der Suche nach genau einer Musterinstanz parallelisiert werden kann, hängt von der Struktur des Musters ab. Daher werden im Anschluss verschiedene Klassen von Mustern auf ihre Parallelisierbarkeit untersucht.

Die Implementierung eines Suchmusters wird im Folgenden als *Suchprogramm* bezeichnet. Wie ein Suchprogramm aussieht, hängt von verschiedenen Entwurfsentscheidungen ab. Im Folgenden wird davon ausgegangen, dass Graphelemente in verketteten Listen abgelegt sind. Pro Knoten- und Kantentyp existiert eine Liste. Soll ein Suchprogramm einen Knoten des Typs A finden, der eine bestimmte Eigenschaft aufweist, so müssen solange Elemente der zum Typ A gehörenden Liste überprüft werden, bis ein Knoten mit den gewünschten Eigenschaften gefunden wurde, oder aber die komplette Liste verarbeitet wurde und somit gefolgert werden kann, dass kein passendes Element existiert. Ein Suchprogramm für das Suchmuster aus Abbildung 4.17 ist in Abbildung 4.18 in Pseudocode notiert. Hierbei fällt auf, dass

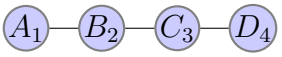

Abbildung 4.17: Ein Suchmuster. Die Knotenbezeichnungen geben den Knotentyp an, die Nummerierung dient der Übersicht.

```
Finde einen Knoten vom Typ A, der...
...über eine Kante mit einem Knoten vom Typ B verbunden ist, der...
...über eine Kante mit einem Knoten vom Typ C verbunden ist, der...
...über eine Kante mit einem Knoten vom Typ D verbunden ist.
```

Abbildung 4.18: Pseudo-Code-Darstellung eines Suchprogramms für das Muster aus Abbildung 4.17.

ein Suchprogramm von einem vorgegebenen Startpunkt aus die zusammenhängenden Komponenten des Musters erforscht. Ob dies in Form einer Breitensuche oder Tiefensuche geschieht, ist implementierungsabhängig.

Ob ein Suchmuster eine parallele Verarbeitung ermöglicht, hängt von dessen Aufbau ab. Im Folgenden werden verschiedene Strukturen untersucht und deren Parallelisierungsmöglichkeiten aufgezeigt.

Muster: Baum mit Kantengrad 1

Entspricht das Suchmuster einem Baum mit maximalem Kantengrad 1, also ohne Verzweigungen, ist eine Parallelisierung nur schwer möglich. Das Auffinden jedes einzelnen Knoten hängt vom vorherigen Knoten ab: Für das Muster aus Abbildung 4.17 kann eine Instanz des Knotens B erst dann gesucht werden, wenn Knoten A gefunden ist, da sonst nicht sichergestellt ist, dass eine Kante von A zu einem Knoten vom Knotentyp B verläuft. Eine Möglichkeit, dennoch parallel zu arbeiten, wird in Abbildung 4.19 veranschaulicht. Die gewählte Instanz von A besitzt eine große Anzahl von ausgehenden Kanten. Welche Kante einen Endknoten vom selben Knotentyp wie B besitzt, muss iterativ ermittelt werden. Diese Überprüfung aller Kanten kann von mehreren Ausführungsfäden gleichzeitig vorgenommen werden. Eine günstige Implementierung würde hierbei erlauben, vor der Kanteniteration die Anzahl der Kanten in O(1) zu ermitteln, um abhängig von der Kantenzahl mehrere Ausführungsfäden einzusetzen.

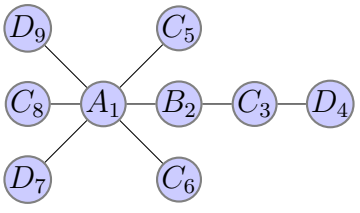

Abbildung 4.19: Ein Arbeitsgraph, der das Suchmuster aus Abbildung 4.17 einmal enthält.

Muster: Baum mit Kantengrad > 1

Ein Muster wie in Abbildung 4.20 entspricht einem Baum mit einem maximalen Ausgangskantengrad > 1. An den Abzweigungen des Baumes kann die Suche nach den

Ästen jeweils von einem eigenen Ausführungsfaden durchgeführt werden. Innerhalb der sequentiellen Teilgraphen kann dann, wie in 4.2.1 beschrieben, der Rechenaufwand bei der Kantenüberprüfung auf weitere Ausführungsfäden verteilt werden. Dies entspricht dann einer verschachtelten, parallelen Ausführung.

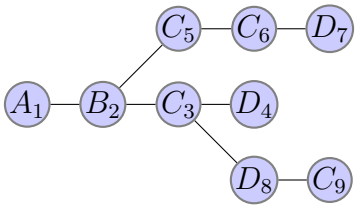

Abbildung 4.20: Ein Suchmuster in der Form eines Baumes.

Diese Methode ist auch möglich, wenn der Graph nicht einen Baum, sondern einen „Baum mit verwachsenen Zweigen" darstellt. Dies ist in Abbildung 4.21 dargestellt.

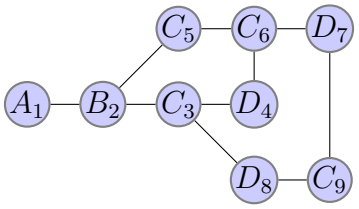

Abbildung 4.21: Ein Suchmuster in der Form eines „Baumes mit verwachsenen Zweigen".

Nicht-zusammenhängende Muster

In Abbildung 4.22 ist ein nicht-zusammenhängendes Muster zu sehen. Hier kann die Suche nach den beiden Teilgraphen von zwei Ausführungsfäden gleichzeitig durchgeführt werden, wobei jeder Faden für einen der Teilgraphen zuständig ist. Bei der Suche nach den Teilgraphen kann der Suchaufwand dann wieder entsprechend den oben beschriebenen Methoden auf weitere Ausführungsfäden verteilt werden.

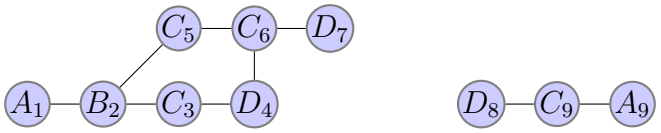

Abbildung 4.22: Ein Suchmuster, das aus mehreren unverbundenen Teilen besteht.

Vorsicht ist jedoch bei der Identifizierung von Knoten im Arbeitsgraphen geboten: In Abbildung 4.22 enthalten beide Teilgraphen einen Knoten vom Typ A. Graphersetzungssysteme wie GrGen.NET ermöglichen es, in solchen Mustern anzugeben, ob beide Instanzen von A im Arbeitsgraphen auf den selben Knoten abgebildet werden dürfen[9]. Ist dies nicht der Fall, muss zwischen den Ausführungsfäden ein Kommunikationsmechanismus definiert werden, der verhindert, dass beide Fäden dieselbe Instanz von A als Teil ihres Suchergebnises verwenden.

[9] Vergleiche „homomorphe Passung" in 2.2.

V-Strukturen

Bei der Suche nach einem Treffer eines Suchmusters können im Arbeitsgraphen Muster auftreten, bei denen der Suchalgorithmus sich für eine Option „entscheiden" muss. Betrachtet man beispielsweise das Suchmuster aus Abbildung 4.17 sowie den Arbeitsgraphen in Abbildung 4.23, so sieht man, dass der Knoten mit der Knotenklasse B mit mehreren Instanzen der Knotenklasse C verbunden ist. Da der Algorithmus keine zusätzlichen Informationen über den Graphen hat, müssen beide möglichen Pfade bezüglich einer Instanz des Suchmusters überprüft werden. Solche „Gabelungen" bei der Suche nach einem Muster werden in [Batz05] *V-Strukturen* genannt[10]. V-Strukturen können parallel überprüft werden, indem jeder Pfad von einem eigenen Ausführungsfaden weiterverfolgt wird. Findet ein Ausführungsfaden eine vollständige Instanz des Suchmusters, können die übrigen Ausführungsfäden terminiert werden[11]. Es ist möglich, dass mehrere V-Strukturen aufeinander folgen. Hier kann ein Arbeitsfaden bei Bedarf erneut unterteilt werden, so dass auch hier ein verschachtelter Parallelismus vorliegt.

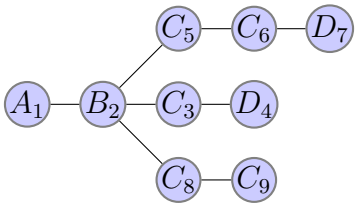

Abbildung 4.23: Ein Arbeitsgraph. Für das Suchmuster aus Abbildung 4.17 ist bei Knoten B_2 eine V-Struktur zu erkennen.

Aufteilung von V-Strukturen

Eine Unterteilung von V-Strukturen auf mehrere Suchfäden lohnt sich nur bei großen Suchmustern oder V-Strukturen mit sehr vielen Verzweigungen, so dass beispielsweise bei 20 Verzweigungen zwei Ausführungsfäden jeweils 10 Pfade überprüfen. Ansonsten überwiegt der Aufwand, die Suche auf die Fäden zu verteilen, den Geschwindigkeitsvorteil durch die parallele Ausführung. Systeme, die das parallele Suchen auf V-Strukturen unterstützen sollen, benötigen daher heuristische Methoden, um abschätzen zu können, ob sich der Aufwand lohnt.

Um die Suche zu beschleunigen, kann auch direkt die Suche nach dem ersten Graphelement parallelisiert werden, in Abbildung 4.17 beispielsweise die Suche nach einem Knoten mit dem Knotentyp A. Wenn mehrere Instanzen dieses Typs im Arbeitsgraphen existieren, kann an jeder Instanz gleichzeitig mit der Suche nach einer Instanz des Suchmusters begonnen werden. Viele vorhandene Instanzen dieses ersten zu suchenden Elements entsprechen einer V-Struktur. Der Unterschied zum oben beschriebenen Fall ist, dass die V-Struktur auftritt, ohne dass bereits ein Teil des Musters gefunden wurde. Dieser Ansatz wird auch bei der Implementierung für GrGen.NET verwendet. Die Implementierung wird in Abschnitt 5.3 beschrieben.

[10] Dabei ist es unerheblich, ob die V-Struktur zwei oder mehr mögliche Verzweigungen aufweist.

[11] Dies gilt nur dann, wenn nur eine Instanz des Musters gesucht wird. Werden mehrere oder alle Instanzen benötigt, so führen die Arbeitsfäden ihre Suche fort.

4.2.2 Suche nach mehreren Instanzen von Suchmustern

Die Suche nach mehreren Instanzen der linken Seite einer oder mehrerer Regeln besitzt eine inhärente Parallelität: Da ein Suchvorgang keine Änderung am durchsuchten Objekt vornimmt, können mehrere Ausführungsfäden ohne die Verwendung von Synchronisierungsmechanismen auf dem Graphen arbeiten. Allerdings muss entschieden werden, wie die Suche auf mehrere Fäden verteilt werden soll. Im Folgenden wird unterschieden, ob nach Musterinstanzen derselben oder verschiedener Regeln gesucht wird.

Suche nach mehreren Instanzen des selben Suchmusters

Die Suche nach mehreren Treffern funktioniert wie die Suche nach einzelnen Treffern der Regel. Allerdings wird die Suche nicht nach dem ersten Treffer beendet. Dies schließt auch die Suche nach allen Treffern der Regel mit ein. Dieser Fall wurde bereits oben im Rahmen der V-Strukturen besprochen: Die parallele Suche nach mehreren Vorkommen derselben Regel zerlegt den Suchraum, der durch die Menge der Graphelemente vom Typ des ersten Graphelements des Suchplans definiert ist, in mehrere Teile, die gleichzeitig auf Instanzen des Suchmusters überprüft werden. Dies ist allerdings nur dann sinnvoll, wenn zwischen den Instanzen des Musters keine Abhängigkeiten bestehen, beziehungsweise wenn der Anwender des Graphersetzungssystems sich über die Folgen bei einer späteren Verarbeitung der Treffer im Klaren ist. Andernfalls muss zuerst eine Instanz des Musters verarbeitet werden, bevor nach weiteren gesucht wird. Dieser Fall wird in den folgenden Abschnitten behandelt.

Suche nach mehreren Instanzen von verschiedenen Suchmustern

Wird nach den Instanzen verschiedener Suchmuster gesucht, kann jeder Suchauftrag in einem eigenen Ausführungsfaden stattfinden. Dabei wird der maximale Parallelitätsgrad durch die Anzahl der gleichzeitig vorliegenden Suchanfragen beschränkt. Dieser Parallelisierungsansatz skaliert daher schlecht mit der Anzahl an verfügbaren Prozessoren: Stehen nur zwei Suchaufträge zur Verfügung, so können maximal zwei Prozessoren gleichzeitig eingesetzt werden. Anders sieht die Situation jedoch bezüglich einer großen Anzahl an gleichzeitig verfügbaren Suchvorgängen aus: hier wird die Anzahl der verfügbaren Prozessoren zum limitierenden Faktor. Allerdings haben die meisten Anwendungen von Graphersetzungssystemen eine begrenzte Anzahl von Regeln, von denen aus semantischen Gründen auch nur eine begrenzte Anzahl gleichzeitig ablaufen können. Geht man von einer stetig steigenden Anzahl an Rechenkernen pro System aus, so scheint es unwahrscheinlich, dass praktische Anwendungen genug Regeln aufweisen werden, um auf Dauer alle verfügbaren Kerne auszulasten. In diesem Fall muss die parallele Suche nach Instanzen verschiedener Regeln mit den oben beschriebenen Methoden kombiniert werden, um durch verschachtelte Anwendungsfäden für einen höheren Parallelitätsgrad zu sorgen.

Verschiedene Suchanfragen besitzen (Zufälle ausgenommen) verschieden lange Laufzeiten. Daher legt die Laufzeit der komplexesten Suchanfrage die Gesamtlaufzeit fest: Während die komplexeste Suchanfrage noch andauert, sind die anderen Ausführungsfäden bereits fertig. Insbesondere ist bei diesem Ansatz kein „Workstealing" möglich: Die wartenden Fäden können dem beschäftigten Faden nicht helfen. Dieses

Problem kann sich in ungünstigen Szenarien drastisch auf die erzielbare Beschleunigung auswirken: Während die Beschleunigung im Idealfall von Anfragen mit identischer Laufzeit linear von der Anzahl der gleichzeitig ausgeführten Suchvorgänge abhängt, lässt sich bei ungünstigen Szenarien, also wenn eine Suchanfrage deutlich länger benötigt als die Summe der Laufzeiten der anderen Anfragen, nur eine geringe Beschleunigung erzielen. Auch hier gilt, dass verschachtelter Parallelismus zu einem besseren Laufzeitverhalten führen kann.

4.2.3 Verknüpfte Graphersetzungsbefehle

Die Regelanwendung ist bei Graphersetzungssystemen indeterministisch. Der Anwendungsentwickler gibt eine Menge von Regeln sowie einen Arbeitsgraphen vor. Das Graphersetzungssystem wählt dann immer wieder zufällig eine Regel aus und versucht diese auf den Arbeitsgraphen anzuwenden. Eine Alternative hierzu bieten Ablaufpläne: viele Graphersetzungssysteme bieten die Möglichkeit, die Reihenfolge der Regelanwendung zu beeinflussen. Hierzu gehört auch die logische Verknüpfung von Regeln. Hierbei ist die wiederholte Ausführung mehrerer Regeln an einen regulären Ausdruck geknüpft. Eine Übersicht über die gebräuchlichsten logischen Ausdrücke in Graphersetzungssystemen bietet Tabelle 4.1. Diese sind darüber hinaus in GrGen.NET-Syntax angegeben.

Tabelle 4.1: Verknüpfungen von Regeln in GrGen.NET

$S*$	S, solange erfolgreich, wiederholt ausführen.
$S\{n\}$	S, solange erfolgreich, wiederholt ausführen, aber n-mal höchstens.
$[S]$	Erst alle Treffer von S finden, anschließend ersetzen (*Apply-All*-Semantik).
$S;T$	Erst S, anschließend T ausführen. Ausführung erfolgreich, falls S oder T erfolgreich.
$S\|T$	S ausführen. Schlägt S fehl, T ausführen. Erfolgreich, wenn S oder T erfolgreich.
$S\&T$	Erst S, dann T ausführen. Erfolgreich wenn S und T erfolgreich.
$S\&\&T$	S ausführen. Ist S erfolgreich, T ausführen. Erfolgreich, wenn S und T erfolgreich.
$S\$ < op >$	Operator $< op >$ kommutativ auswerten.

In diesem Abschnitt soll für diese Verknüpfungen untersucht werden, wie eine parallele Ausführung ermöglicht werden kann und welche Einschränkungen die Semantik der verschiedenen Befehlssequenzen aufzeigen. Die folgenden Unterabschnitte beschreiben die Parallelisierungsmöglichkeiten der gebräuchlichsten Verknüpfungen. Hierbei wird auch beschrieben, wie diese Verknüpfungen sich für eine Anwendung auf partitionierten Graphen eignen.

4.2.3.1 Wiederholte Ausführung einer Regel

Die mehrfache Ausführung einer Regel wird in GrShell durch den Befehl `A*` ausgedrückt, wobei `A` der Bezeichner der Regel ist. Ziel dieser Regelsequenz ist es, `A` so oft wie möglich auf den Arbeitsgraphen anzuwenden. Hierbei ist zu beachten, dass die Anwendung einer sequentiellen Semantik folgen muss: Ein zweites Vorkommen der linken Seite von `A` soll erst gesucht werden, nachdem die rechte Seite des ersten Vorkommens vollständig ausgeführt wurde. Dies ist von Bedeutung, da die Ausführung

einer Regel das Auffinden weiterer Vorkommen beeinflussen kann. Ein Beispiel für solch eine Regel zeigt Abbildung 4.24. Wird diese Regel auf den Arbeitsgraphen in Abbildung 4.25 angewandt, so können (ohne direkte Anwendung der rechten Seite) zwei Vorkommen der Regel festgestellt werden. Wird die Regel jedoch direkt nach dem Auffinden des ersten Vorkommens angewandt, verschwindet das zweite, potentielle Vorkommen und die Regel kann nur einmal angewendet werden. Die beiden unterschiedlichen Ergebnisse sind in Abbildung 4.26 zu sehen.

Abbildung 4.24: Eine Graphersetzungsregel. Die Buchstaben geben die Knotentypen an. Die Indizes dienen der Identifizierung der Knoten.

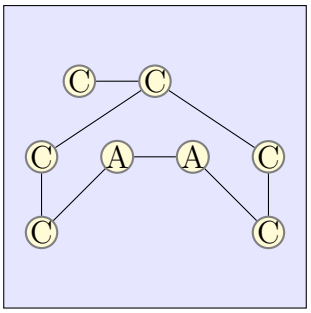

Abbildung 4.25: Ein Arbeitsgraph. Die Knotenbezeichnungen geben den Knotentyp an.

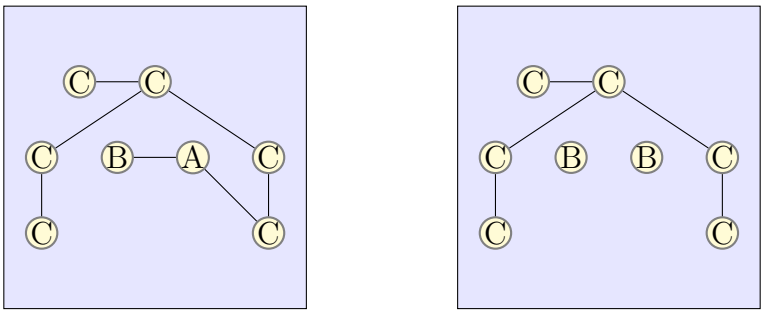

Abbildung 4.26: Der Arbeitsgraph aus Abbildung 4.25 nach Anwendung der Regel 4.24. Links wurde die Regel einmal gefunden und ausgeführt. Anschließend kann die Regel nicht erneut ausgeführt werden. Rechts wurde die Regel zweimal gefunden und anschließend zweimal ausgeführt.

Dennoch kann diese sequentielle Regelausführung parallelisiert werden. Die Voraussetzungen hierfür liefert das Parallelisierungstheorem 3.1.1.1. Bei obigem Beispiel haben beide Vorkommen der Regel gemeinsame Elemente außerhalb des Klebegraphen, daher können sie nicht parallel verarbeitet werden. Unter Verwendung einer Partitionierung des Arbeitsgraphen lässt sich allerdings leicht sicherstellen, dass keinerlei gemeinsame Elemente, also insbesondere nicht außerhalb des Klebegraphen, bei zwei Vorkommen auftreten.

Die gleichzeitige Ausführung zweier Vorkommen, die keinerlei Elemente gemeinsam besitzen, entspricht der Parallelisierung nach dem Parallelisierungstheorem. In Ab-

bildung 4.27 ist zu sehen, dass das Muster der Regel 4.24 in 4 verschiedenen Partitionen auftritt. Diese können somit alle gleichzeitig ausgeführt werden, ohne dass sich die Semantik im Vergleich zu einer sequentiellen Ausführung verändert.

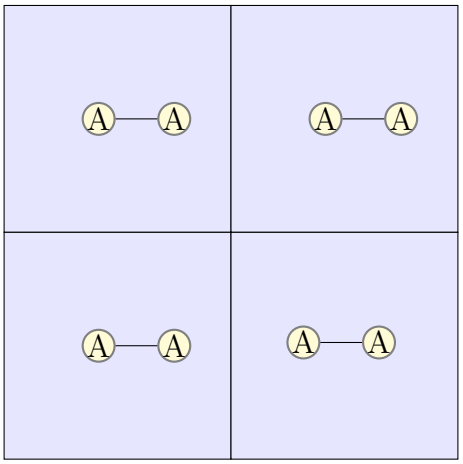

Abbildung 4.27: Der Arbeitsgraph besteht aus 4 Partitionen. Jede enthält das Muster der Regel aus 4.24 genau einmal.

4.2.3.2 Konkatenation

Eine Konkatenation wird in GrShell durch `A ; B` ausgedrückt. Hierbei soll zunächst `A` und anschließend `B` einmalig ausgeführt werden. Da in diesem Fall sicher ist, dass beide Regeln ausgeführt werden müssen, ist es möglich, den Graphen zu halbieren und in einer Hälfte nach einem Vorkommen von Regel A, in der anderen nach Regel B zu suchen. Wurde jeweils ein Treffer gefunden, können diese parallel ausgeführt werden, da sie keine gemeinsamen Teile beinhalten. Wurden keine Treffer gefunden, wird auf der jeweils anderen Hälfte nach A und B gesucht. Dies lässt sich sehr gut mittels Partitionen ausführen, da diese bereits eine Partition des Graphen vorgeben und somit nur noch ermittelt werden muss, in welchen Partitionen nach Muster A und in welchen nach Muster B gesucht werden soll.

Wird nur eine Instanz der linken Seite von Regel B gefunden, so wird hier die rechte Seite ausgeführt und der Vorgang ist beendet. Wird hingegen ausschließlich eine Musterinstanz von Regel A gefunden, so muss nach der Ausführung der rechten Seite von Regel A die entsprechende Partition erneut nach Regel B durchsucht werden, die Anwendung von A die Anwendung von B ermöglichen kann. In diesem Fall kann somit zwar die Suche parallelisiert werden, nicht aber die Ausführung der Regeln, da eine kausale Abhängigkeit zwischen beiden besteht.

Der Rückgabewert einer Konkatenation ist `true`, wenn mindestens eine der beiden Regeln erfolgreich war, ansonsten `false`. Die Konkatenation entspricht somit der logischen Funktion „Or".

4.2.3.3 Logische „Und"-Verknüpfung

Die logische Und-Operation, in GrShell durch `A & B` ausgedrückt, kann genauso wie die Konkatenation durchgeführt werden. Beide Regeln werden ausgeführt, wobei geprüft werden muss, ob die Ausführung von A eine anschließende Ausführung von B ermöglicht. Die Und-Verknüpfung unterscheidet sich von einer Konkatenation somit nur im Rückgabewert, dieser ist genau dann `true`, wenn beide Regeln erfolgreich angewendet wurden, ansonsten `false`.

4.2.3.4 Konkatenation („faule" Auswertung)

Eine Konkatenation mit „fauler" Auswertung wird in GrShell durch `A | B` ausgedrückt. Sie enthält eine sequentielle Abhängigkeit: B wird nur dann ausgeführt, wenn keine Instanz der linken Seite von A im Arbeitsgraphen gefunden werden konnte. Dies schränkt die Möglichkeit einer parallelen Ausführung ein: Da zunächst eine Musterinstanz von A gefunden werden muss, kann eine parallele Suche nach Instanzen von B überflüssig sein, wenn keine Instanz von A gefunden wird. Eine parallele Suche nach A und B ist somit nur dann sinnvoll, wenn mehr Rechenkerne zur Verfügung stehen, als für die Suche nach einem Vorkommen von A eingesetzt werden können. Ist eine Instanz von A gefunden, so kann während der Anwendung von A bereits in anderen Partitionen nach Instanzen von B gesucht werden. Verlief diese Suche nach Instanzen von B nicht erfolgreich, so wird nach der Ausführung von A auch diese Instanz nach B durchsucht. Der Rückgabewert ist `true`, falls A oder B erfolgreich ausgeführt wurden.

4.2.4 Unabhängige Regeln

Das Parallelisierungstheorem gibt Auskunft, wann zwei Vorkommen einer Regel parallel angewendet werden können. Allerdings gibt es auch eine Situation, in der die parallele Anwendbarkeit zweier Regeln unabhängig von den konkreten Vorkommen festgestellt werden kann. Seien A und B zwei Graphersetzungsregeln, T die Menge aller Typen von Graphelementen im Graphmodell und T_A sowie T_B die Menge der Graphelementtypen, die in den Regeln A beziehungsweise B vorkommen. Gilt nun $T_A \cap T_B = \emptyset$, so kann daraus gefolgert werden, dass ein Vorkommen der Regel A und ein Vorkommen der Regel B nie gemeinsame Elemente besitzen. Das Parallelisierungstheorem ist somit für zwei Instanzen dieser Regeln immer erfüllt. Zwei Instanzen dieser Regeln können somit immer ohne weitere Überprüfung parallel ausgeführt werden.

Ob zwei Regeln die Bedingung $T_A \cap T_B = \emptyset$ erfüllen, kann vor der eigentlichen Graphersetzung für alle Kombinationen der verfügbaren Graphersetzungregeln überprüft und abgespeichert werden. Somit kann im Verlauf der Graphersetzung bei möglicher gleichzeitiger Anwendung zweier Regeln in O(1) überprüft werden, ob der hier vorgestellte Fall vorliegt und entsprechend verfahren werden.

Als unabhängig gelten des Weiteren Regeln, die höchstens im Klebegraphen Graphelemente des gleichen Typs haben. In diesem Fall können auch Musterinstanzen solcher Regeln ohne weitere Prüfung auf gemeinsame Elemente parallel ausgeführt werden. Abhängig von der gewählten Datenstruktur kann jedoch ein gemeinsamer Zugriff auf Klebeelemente erfolgen. Abbildung 4.28 veranschaulicht diesen Fall: Die beiden oben dargestellten Regeln haben nur im Klebegraphen Elemente einer gemeinsamen Klasse (A). Von beiden Regeln kann im unten abgebildeten Arbeitsgraphen eine Musterinstanz gefunden werden. Beide Musterinstanzen teilen sich hierbei den Knoten vom Typ A. Da beide Knoten eine Kante, die zu diesem A-Knoten führt, löschen müssen, muss an dieser Stelle der Zugriff auf die Kantenliste synchronisiert werden.

4.3 Zusammenfassung

In diesem Kapitel wurde beschrieben, wie sich Graphersetzungssysteme parallelisieren lassen. Um Graphersetzungsregeln gleichzeitig auszuführen, muss das Paralleli-

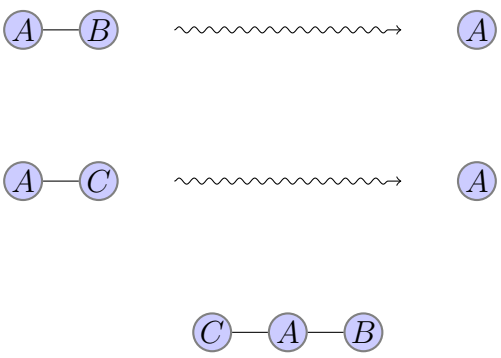

Abbildung 4.28: Zwei Graphersetzungsregeln, die nur im Klebegraphen gemeinsame Typen besitzen. Unten ein Arbeitsgraph, der jeweils eine Instanz der beiden Regeln besitzt.

sierungstheorem beachtet werden. Durch Partitionierung des Arbeitsgraphen wurde eine skalierbare Methode zur Überprüfung der parallelen Unabhängigkeit aufgezeigt. Hierfür wurde mit dem Rubini-Verfahren des Weiteren ein Partitionierungsverfahren entwickelt. Auch ohne Partitionierung lässt sich die Mustersuche in verschiedenen Varianten auf mehrere Arbeitsfäden verteilen. Welcher Ansatz hierbei geeignet ist, hängt vom Aufbau des Suchmusters ab. Im folgenden Kapitel wird eine Implementierung der Partitionierung sowie einiger Varianten der Mustersuche ohne Partitionierung vorgestellt.

5. Implementierung

Im Rahmen dieser Arbeit wurden die in Kapitel 4 vorgestellten Verfahren für das GrGen.NET-Graphersetzungssystem umgesetzt. Dieses Kapitel beschreibt die Implementierung sowie die nötigen Änderungen, die an GrGen.NET hierzu vorgenommen werden mussten.

Abschnitt 5.1 beschreibt den Aufbau des GrGen.NET-Systems, sowie die darin verwendeten Datenstrukturen. Graphpartitionen und die Verarbeitung von Graphersetzungsregeln bei partitionierten Graphen werden in 5.2 erläutert. Die parallele Mustersuche wird in 5.3 thematisiert. Abschließend wird in Kapitel 5.4 angegeben, welche Änderungen am Code-Generator nötig waren, um eine Parallelisierung zu ermöglichen.

5.1 Strukturelle Grundlagen von GrGen.NET

In diesem Abschnitt wird der Aufbau des GrGen.NET-Systems vorgestellt. Hierbei wird insbesondere auf die verwendeten Datenstrukturen eingegangen. Dies ist nötig, um die im Rahmen der Parallelisierung durchgeführten Änderungen und Erweiterungen nachzuvollziehen.

In 5.1.1 werden die Komponenten von GrGen.NET beschrieben. 5.1.2 geht auf die verwendeten Datenstrukturen ein.

5.1.1 Subsysteme von GrGen.NET

GrGen.NET ist keine monolithische Anwendung, sondern ein System von separat verwendbaren Modulen. Diese werden, wie auch ihr Zusammenhang, in diesem Abschnitt betrachtet.

Ein typischer Anwendungsfall von GrGen.NET kann wie folgt beschrieben werden: Der Anwender möchte ein Graph-Modell sowie Regeln, die auf diesem Modell basieren, definieren. Anschließend soll ein Arbeitsgraph geladen werden, auf dem die Regeln in vom Anwender vorgegebener Reihenfolge angewendet werden. Danach soll der Ergebnis-Graph visuell dargestellt werden und auf Konformität zum Graphmodell geprüft werden. Diese Aufgaben lassen sich den einzelnen Komponenten von

GrGen.NET zuordnen. *GrGen.exe* kompiliert Graphmodelle und Regeln, die in einer GrGen-eigenen Sprache verfasst wurden. Eine Übersicht der verwendeten Dateitypen ist in Tabelle 3.1 zu sehen. Die *GrShell* kann diese Modelle anschließend laden. Hier können Arbeitsgraphen definiert und Regelsequenzen angewendet werden. Des Weiteren ist auch eine Validierung des Arbeitsgraphens bezüglich Modell-Zusicherungen möglich. Der erzeugte Graph kann anschließend mit der Java-Anwendung *yComp* (3.4.3) betrachtet werden. Dies ist in Abbildung 5.1 schematisch dargestellt.

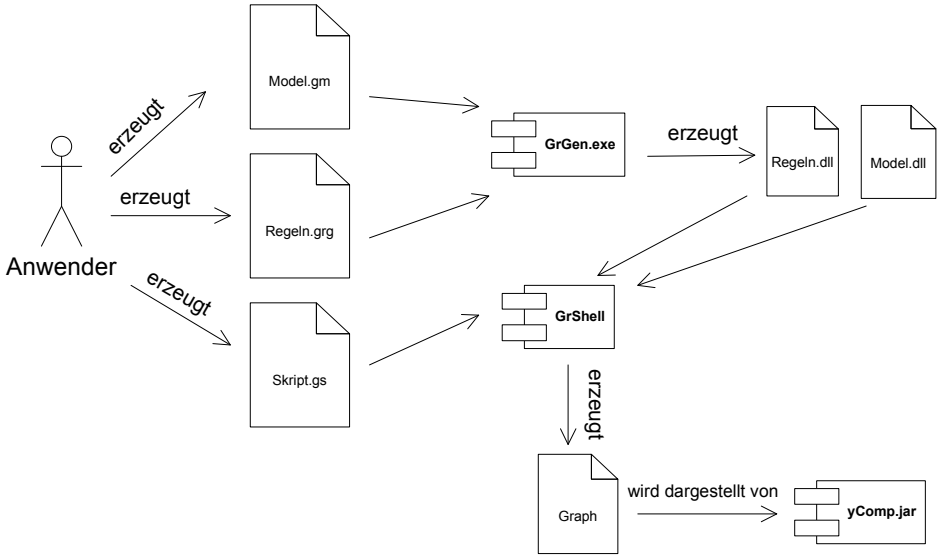

Abbildung 5.1: Die Komponenten und Artefakte von GrGen.NET.

Im Folgenden werden die einzelnen Komponenten genauer beschrieben.

GrGen.exe

GrGen.exe ist ein Compiler, der die Eingabemenge aus *.gm- und *.grg-Dateien in zwei .NET-Bibliotheken übersetzt: <Name>Modell.dll und <Name>Actions.dll, wobei <Name> der Dateiname der *.gm-Datei ist[1]. <Name>Modell.dll enthält hierbei das definierte Graphmodell, <Name>Actions.dll die auf diesem Modell definierten Regeln.

GrGen.exe besteht aus zwei Teilen: Einem Java-Teil und einem .NET-Teil. Zunächst werden die Eingabedateien (*.gm und *.grg) an den Java-Teil übergeben. Dieser besteht aus dem Java-Programm *grgen.jar*, welches anhand einer ANTLR-Grammatik [Parr] .NET-Klassen für das Graphmodell und die Graphregeln erzeugt. Die Klassen für die Graphregeln enthalten bereits Code, um eine gefundene Instanz der linken Seite einer Regel durch die rechte Seite zu ersetzen. Der Code für das Auffinden von Instanzen der linken Seite fehlt jedoch.

Die erzeugten C#-Quellcode-Dateien werden an den .NET-Teil der GrGen.exe-Implementierung übergeben. Dieser fügt den fehlenden Code für die Mustersuche hinzu und kompiliert den nun vollständigen C#-Quellcode zu einer .NET-Bibliothek.

[1] Ohne das *.gm-Suffix des Dateinamens.

GrShell.exe

Bei *GrShell.exe* handelt es sich um eine Konsolenanwendung. Von hier aus können durch eine Eingabeaufforderung die von *GrGen.exe* erzeugten .NET-Bibliotheken geladen werden. Anschließend kann interaktiv ein Arbeitsgraph erstellt und die in den geladenen Bibliotheken vorhandenen Graphregeln ausgeführt werden. Bei Bedarf kann auch direkt aus der Eingabeaufforderung das Anzeigeprogramm *yComp* gestartet werden. Auch eine interaktive Schritt-für-Schritt-Ausführung ist möglich. Die Veränderung des Arbeitsgraphen kann mit *yComp* auch in Einzelschritten betrachtet werden.

Um den bei einer Konsolenanwendung anfallenden Schreibaufwand abzukürzen, können Befehlssequenzen für GrShell auch in Skript-Dateien (*.grs) abgelegt und durch einen einzigen Befehl ausgeführt werden.

5.1.2 Verwaltung von Graphelementen

5.1.2.1 Die GrGen.NET-Typbibliotheken

Jeder Graphelementtyp wird in <Name>Modell.dll als eigene .NET-Klasse dargestellt, wobei im Graphmodell definierte Typhierarchien durch die Implementierung von .NET-Schnittstellen umgesetzt werden[2].

In <Name>Actions.dll wird jede Graphregel als eigene .NET-Klasse umgesetzt. Diese bietet Methoden, um nach dem Muster der Regel in einem Graphen zu suchen und um die rechte Seite der Regel auf eine bereits gefundene Instanz des Musters anzuwenden.

5.1.2.2 Knoten und Kanten

Knoten und Kanten werden in GrGen.NET als .NET-Objekte repräsentiert. Jedes Knoten-Objekt besitzt zwei Referenzen auf Kanten: eine für ausgehende Kanten und eine für eingehende Kanten. Die Kanten-Objekte eines Knotens werden in einer doppelt-verkettete Liste verwaltet: Kanten-Objekte enthalten daher vier Referenzen auf andere Kanten-Objekte. Zwei dieser Referenzen navigieren vorwärts und rückwärts durch die ausgehenden Kanten eines Knotens, die anderen beiden durch die eingehenden. Für den Graphen aus Abbildung 5.2 ist in Abbildung 5.3 die passende Datenstruktur dargestellt. Des Weiteren besitzt jede Kante noch eine Referenz auf ihre Quell- und Zielknoten.

5.1.2.3 Der Arbeitsgraph

Die Knoten und Kanten des Arbeitsgraphen werden mit einem System von Ringlisten verwaltet. Für jeden Knoten- und Kantentyp wird eine eigene Ringliste verwendet. Alle Knoten und Kanten des Graphen werden als eigenes Objekt repräsentiert und in der Ringliste des passenden Typs abgelegt. Jede Ringliste besitzt einen Listenkopf, der selbst Element der Ringliste ist und kein konkretes Graphelement repräsentiert[3].

[2] Eine direkte Umsetzung durch Subklassenbildung ist nicht möglich, da .NET keine Mehrfachvererbung erlaubt, GrGen.NET im Graphmodell hingegen schon. Somit werden alle Klassenhierarchien über Schnittstellen abgebildet, um so auch eine Art der Mehrfachvererbung unter .NET zu realisieren.

[3] Existiert von einem bestimmten Typ keine Instanz in einem Graphen, so enthält die Ringliste des Typs genau ein Element: den Listenkopf.

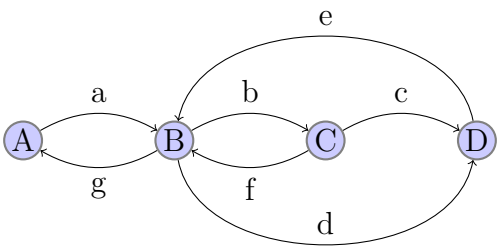

Abbildung 5.2: Ein Graph, der aus 4 Knoten und 7 Kanten besteht. Die Kantenbezeichnungen werden in Abbildung 5.3 wiederverwendet.

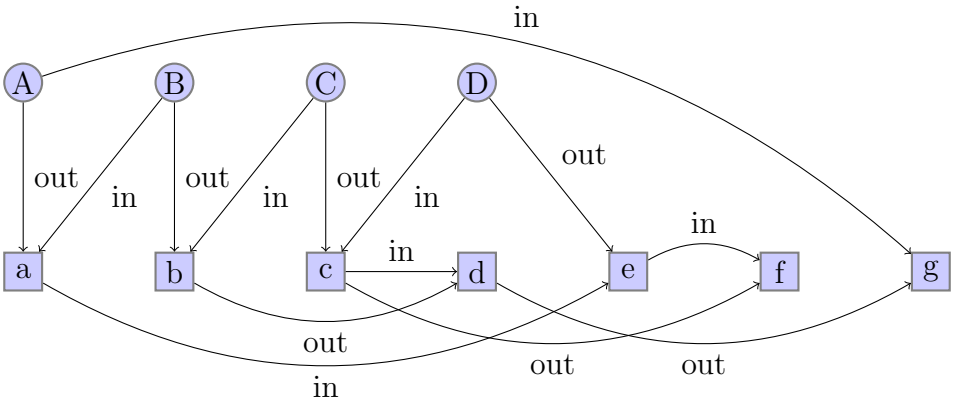

Abbildung 5.3: Graphische Darstellung der Datenstruktur des Graphen aus Abbildung 5.2.

Die Ringlisten werden in zwei Feldern verwaltet: eines für Knoten und eines für Kanten. Für jede Ringliste ist genau ein Feld-Element reserviert, welches eine Referenz auf den Listenkopf beinhaltet. Dies ist in Abbildung 5.4 graphisch dargestellt.

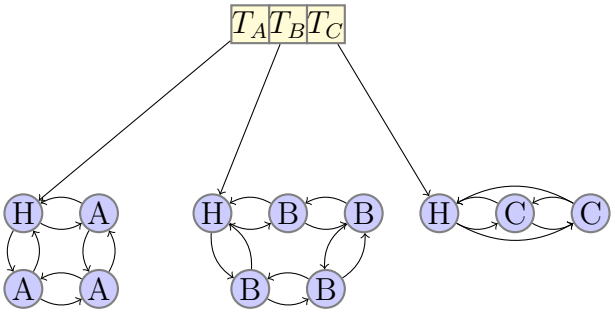

Abbildung 5.4: Graphische Darstellung der Typ-Ringlisten. Die Elemente jedes Knoten- und Kantentyps sind in einer eigenen Ringliste abgelegt. Ein Feld speichert die Referenzen der Listenköpfe.

GrGen.NET bietet Methoden, um für einen gegebenen Graphelement-Typen eine ID zu erhalten. Diese ID dient als Index in das entsprechende Ringlisten-Feld. Somit lässt sich anhand der Klasse eines Knotens oder einer Kante auf seine konkreten Objekte im aktuellen Arbeitsgraphen zugreifen. Möchte man einen Befehl für alle Elemente einer Klasse durchführen, so kann man beginnend beim Listenkopf so lange über die Elemente iterieren, bis der Listenkopf wieder „auftaucht": Dann wurde der komplette Ring einmal abgearbeitet. In Abbildung 5.5 wird dies in Pseudocode verdeutlicht.

```
Setze Knoten kopf = Listenkopf
Wenn (kopf.Next != NULL)
  Setze Knoten elem = kopf.Next
  Solange (elem != kopf)
    elem verarbeiten
    elem = elem.Next
  Schleifenende
Sonst
  Ringliste ist leer
```

Abbildung 5.5: Pseudocode: Iterieren einer Ringliste

5.1.2.4 Verwalten von Musterinstanzen

Wird bei der Mustersuche eine Instanz gefunden, müssen die zum Muster gehörenden Graphelemente bis zum Ersetzen des Musters abgespeichert werden[4]. Hierzu wird die .NET-Klasse *LGSPMatch* verwendet. Die Knoten und Kanten werden jeweils in einem Feld abgespeichert. Da die benötigte Feldlänge vom Suchmuster abhängt, wird diese im Objekt-Konstruktor von *LGSPMatch* festgelegt.

[4]Es ist auch möglich nur nach Instanzen eines Musters zu suchen ohne diese direkt zu ersetzen. Beispielsweise könnte man so zunächst weitere Kriterien, die von der Regel nicht erfasst werden, überprüfen, um anschließend nur einige Vorkommen des Suchmusters zu ersetzen.

Um alle gefundenen Instanzen des Suchmusters zu verwalten, werden die *LGSPMatch*-Instanzen in einer verketteten Liste abgelegt. Die Liste ist durch eine innere Verzeigerung der *LGSPMatch*-Objekte realisiert, jedes *LGSPMatch*-Objekt besitzt somit eine Referenz auf das nächste *LGSPMatch*-Objekt.

Weitere Informationen zu GrGen.NET sind in [GrG] zu finden.

5.2 Implementierung von Graphpartitionen

In diesem Abschnitt wird beschrieben, wie die Partitionierung des Arbeitsgraphen für GrGen.NET umgesetzt wurde. In 5.2.1 werden zunächst die für Partitionen erzeugten Datenstrukturen erläutert. Anschließend zeigt 5.2.2, wie die Partitionierung selbst vorgenommen wird. Abschnitt 5.2.3 stellt die Suche auf Partitionen dar. In 5.2.4 wird gezeigt, wie parallele und sequentielle Methoden gemeinsam auf einen Graphen angewendet werden können. Zuletzt wird in 5.2.5 aufgezeigt, wie die parallele Ausführung verschiedener Regeln erläutert.

5.2.1 Die Klasse Partition

Eine Partition entspricht einem vollständigen Graphen und kann in GrGen.NET wie ein gewöhnlicher Graph als Struktur zur Verwaltung von Knoten und Kanten verstanden werden. Graphen werden in GrGen.NET durch die Klasse `LGSPGraph` dargestellt. Grundlegende Mechanismen zum Einfügen und Löschen von Knoten sind in dieser Klasse ebenso zu finden wie die Listen zum Speichern von Knoten und Kanten verschiedener Typen. Daher wird für die Darstellung von Partitionen eine Klasse `Partition` von `LGSPGraph` abgeleitet. Da nach einer Partitionierung einem Graphen mehrere Partitionen zugeordnet sind, wurde die Klasse `LGSPGraph` um eine Liste vom Typ `Partition` erweitert. Dies wird in Abbildung 5.6 veranschaulicht.

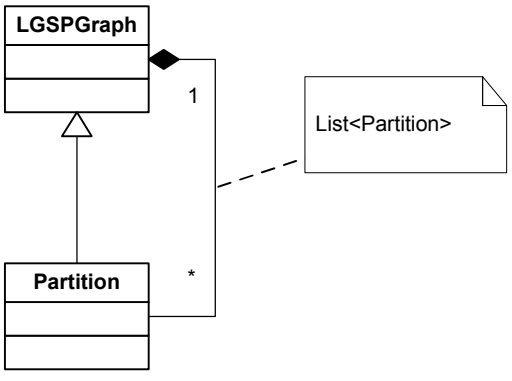

Abbildung 5.6: Die Klasse Partition.

Durch den in Abbildung 5.6 dargestellten Zusammenhang zwischen `LGSPGraph` und `Partition` besitzt jede Partition selbst wieder eine Liste von Partitionen. Somit ergibt sich eine rekursive Partitionsstruktur. Partitionen werden in dieser Arbeit allerdings nur in der ersten Stufe berücksichtigt. „Partitionen von Partitionen" werden

nicht verwendet. Allerdings bilden diese einen interessanten Ansatz für weitere Arbeiten: In 4.1.3 wurde beispielsweise gezeigt, dass Partitionen, die zu viele Graphelemente enthalten, in mehrere Partitionen zerteilt werden können. Da eine Partition selbst ein `LGSPGraph` ist, kann die Teilung mit den `LGSPGraph`-eigenen Methoden durchgeführt werden. Eine weitere mögliche Anwendung wäre die Verwendung von rekursiven Partitionen, um mit verschachteltem Parallelismus zu arbeiten.

5.2.2 Erzeugen von Graph-Partitionen

Das Erzeugen von Graphpartitionen ist ein aus zwei Schritten bestehender Prozess:

1. Ordne allen Knoten eine Partition zu.
2. Verschiebe jeden Knoten in die ihm zugewiesene Partition.

In Schritt 1 werden die in 4.1.2 beschriebenen Partitionierungsverfahren (zufällig und Rubini) verwendet. Diese besuchen alle Knoten des Graphen und setzen jeweils das `PartitionID`-Feld der Knoten. In Schritt 2 wird anschließend erneut über alle Knoten iteriert. Hierbei werden die Knoten dann aus den Typlisten des Hauptgraphen entfernt und in die entsprechenden Listen der Partitionen eingefügt. Im Folgenden werden beide Schritte genauer beschrieben.

Schritt 1: Partitionierung

Um das verwendete Partitionierungs-Verfahren austauschbar zu halten, wurde eine Schnittstelle erzeugt. Sie wird von drei Strategien[5] implementiert: Die Klasse `RandomDistribution` besucht die Knoten des Graphen mittels Tiefensuche und „würfelt" jeweils eine PartitionsID für jeden besuchten Knoten. Für die Partitionierung mittels Rubini-Verfahren stehen zwei Klassen zur Verfügung: `RubiniSingle` und `RubiniMulti`. Beide Klassen können wahlweise selbst Ankerpunkte erwürfeln, oder eine übergebene Knotenmenge als Ankerpunkte verwenden. `RubiniSingle` implementiert das Rubini-Verfahren mit einem Ausführungsfaden, während `RubiniMulti` mehrere Ausführungsfäden verwendet.

Schritt 2: Verschieben der Knoten

Sobald jeder Knoten einer Partition zugeteilt wurde, können diese in die Knotenlisten der Partitionen verschoben werden. Für jeden Knotentyp des Graphens wird ein eigener Ausführungsfaden verwendet. Somit können alle Knotenlisten des Hauptgraphen gleichzeitig verarbeitet werden. Da die Elemente einer Typliste zwar in verschiedenen Partitionen, aber jeweils immer in der Liste des gleichen Typs eingefügt werden, kann diese Verarbeitung ohne Sperren durchgeführt werden. Abbildung 5.7 veranschaulicht diesen Zusammenhang.

[5] *Strategie* ist ein Verhaltensmuster, um Algorithmen austauschbar zu halten. Dieses wird, wie viele andere Entwurfsmuster, in [GHJV95] beschrieben.

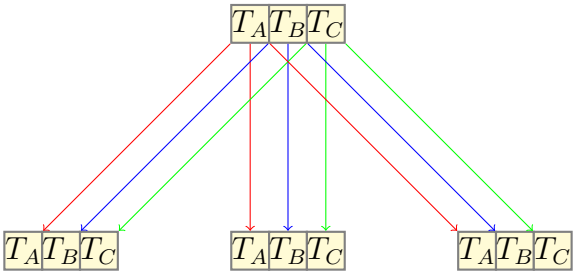

Abbildung 5.7: Die Knoten jeder Typliste des Hauptgraphen werden auf die entsprechenden Typlisten der Partitionen verteilt.

5.2.3 Mustersuche auf Graphpartitionen

Die Suche nach Musterinstanzen auf Partitionen funktioniert wie die ursprüngliche Suche in GrGen.NET (vgl. 5.3.1.1). Es wird der komplette im Teilgraphen befindliche Suchraum von einem Arbeitsfaden durchsucht. Möglich wäre auch eine Aufteilung des Suchraums wie in 4.2.2 (zusätzlich zur Partitionierung). Somit wäre eine verschachtelte Verarbeitung möglich: Mehrere Arbeitsfäden verarbeiten die vorhandenen Partitionen, die jeweils wieder von mehreren Fäden durchsucht werden. Um dies zu ermöglichen, sind jedoch weitere Änderungen am Code-Generator nötig (vergleiche 5.4), die jedoch den zeitlichen Rahmen dieser Arbeit überschritten hätten.

5.2.4 Interoperabilität zwischen parallelen und sequentiellen Methoden

Partitionierte Arbeitsgraphen haben keine Knoten mehr in ihren eigenen Datenstrukturen. Nicht-partitionsfähige Operationen können solche Arbeitsgraphen nicht korrekt verarbeiten, sie stellen sich für sie wie leere Graphen dar. Um die ursprünglichen, nicht-parallelen Befehle von GrGen.NET mit partitionierten Graphen verwenden zu können, ist es daher nötig, diese anzupassen. Die sequentiellen Suchfunktionen müssten beispielsweise so abgeändert werden, dass sie nicht nur einen Graphen, sondern auch dessen Partitionen nach Knoten absuchen. Dies konsistent für GrGen.NET umzusetzen, bleibt eine der Aufgaben, welche die Parallelisierung in der Folge nach sich zieht. Im Rahmen dieser Arbeit konnte solch eine umfassende Bearbeitung des ursprünglichen GrGen.NET-Quellcodes aufgrund des zeitlichen Rahmens nicht vorgenommen werden.

In diesem Abschnitt wird beschrieben, wie parallele und sequentielle GrGen.NET-Befehle dennoch abwechselnd verwendet werden können.

Auflösen von Partitionen

Um die ursprünglichen Methoden von GrGen.NET bei einem bereits partitionierten Graphen weiterverwenden zu können, ist es nötig, die Knoten wieder in den Hauptgraphen zu verschieben. Hierzu wurde der `Graph`-Klasse die Methode `ClaimPartitions` hinzugefügt. Diese durchsucht alle Partitionen eines Graphen und verschiebt deren Knoten in den Hauptgraphen. Nach dem Methodenaufruf besitzen die Partitionen somit keine Knoten mehr, diese sind nun wieder direkt im Hauptgraphen zu finden und können von den ursprünglichen Methoden von GrGen.NET verarbeitet werden.

Da Knoten in GrGen.NET in Listen abgespeichert sind, können diese in die Listen des Hauptgraphen umgehängt werden. Ein physikalisches Kopieren der Knoten ist nicht notwendig. Das Verfahren ist in Abbildung 5.8 in Pseudocode formuliert.

```
Für alle Partitionen P des Graphen
  Für alle Knotentypen T des Graphmodells
    Füge den ersten Knoten vom Typ T an
      die T-Knotenliste des Hauptgraphen G an
    Definiere den letzten T-Knoten von
      P als letzten T-Knoten von G
    Leere die T-Knotenliste von P
  Nächster Knotentyp T
Nächste Partition P
```

Abbildung 5.8: Pseudocode-Darstellung der `ClaimPartitions`-Methode.

Um anschließend wieder mit Methoden zu arbeiten, die Partitionen benötigen, können die Knoten wie gewohnt auf die Partitionen verteilt werden. Es ist nicht nötig, erneut einen Graphpartitionierungsalgorithmus wie das Rubini-Verfahren auszuführen, da die Knoten ihre Partition immer noch im Feld `PartitionsID` abgespeichert haben. Die Knoten können daher nacheinander wieder ihren Partitionen zugeordnet werden. Hat eine sequentielle Methode neue Knoten erzeugt, so wurde diesen noch keine Partition zugeordnet. Somit muss bei einer Partitionsverteilung der Knoten nach einem Aufruf von `ClaimPartitions` geprüft werden, ob jedem Knoten eine gültige Partition zugeordnet ist. Werden Knoten ohne gültige Partition gefunden, so kann diesen die Partition eines Nachbarknotens zugeordnet werden.

Fork-Join-Parallelismus

Die Methode `ClaimPartitions` ermöglicht es, partitionsbezogene und nicht-partitionsbezogene Methoden abwechselnd zu verwenden. Daher können insbesondere auch neuere GrGen.NET-Techniken, wie beispielsweise Musteralternativen [Jaku08], zusammen mit parallelen Methoden verwendet werden. Dieser Ansatz erinnert an den *Fork-Join-Parallelismus*: Hier werden bei parallelen Bereichen in Anwendungen mehrere Anwendungsfäden erzeugt, die einen Codeabschnitt parallel ausführen. Anschließend werden diese Fäden beendet und die Anwendung wird sequentiell fortgesetzt. So kann auch hier der Graph bei Bedarf in Partitionen aufgeteilt und anschließend wieder in einen einzigen Graph zusammengeführt werden.

5.2.5 Parallele Ausführung verschiedener Regeln

In diesem Abschnitt wird beschrieben, wie die parallele Ausführung verschiedener Regeln implementiert wurde. Die parallele Ausführung ist hierbei immer mit der Suche nach Instanzen der entsprechenden Regeln im Arbeitsgraphen verbunden, da geprüft werden muss, welche Instanzen parallel unabhängig sind. Die parallele Ausführung arbeitet des Weiteren auf den oben beschriebenen Graphpartitionen. Eine parallele Operation wird durch eine Instanz einer von der abstrakten Klasse *Repetition* abgeleiteten Klasse dargestellt. Die einzelnen Implementierungen von *Repetition* repräsentieren jeweils eine eigene semantische Verknüpfung paralleler Regeln. Tabelle 5.1 gibt eine Übersicht.

Tabelle 5.1: Die abstrakte Klasse Repetition und ihre Implementierungen

Dateityp	Verwendung	
Repetition	-	Repräsentiert die wiederholte Ausführung von verknüpften Regeln.
RepetitionSingle	$R*$	Die Regel R wird so lange angewendet, wie ihre Ausführung erfolgreich war.
RepetitionOr	$(R_1\|..\|R_{n-1}\|R_n)*$	Die Regeln R_1 bis R_n werden mit Oder-Semantik ausgeführt. Dies wird wiederholt, solange die Anwendung erfolgreich ist.
RepetitionAnd	$(R_1\&R_2)*$	Die Regeln R_1 und R_2 werden mit einer Und-Verknüpfung ausgeführt. Wiederholung solange Anwendung erfolgreich.

Dem Konstruktor der *Repetition*-Klasse wird die Anzahl der für die Ausführung zu verwendenden Fäden übergeben. Diese werden von *Repetition* verwaltet und sind für den Aufrufer transparent. Für jeden Faden wird eine Instanz der Klasse *ThreadData* angelegt, der zusätzlich zwei Sperren übergeben werden. Diese dienen der Signalisierung und Kommunikation mit dem Hauptfaden. *ThreadData* startet den Ausführungsfaden, welcher die Methode *ThreadMainLoop* ausführt. Bekommt ein Faden das Signal zu arbeiten, führt er die Methode *DoJob* einer Instanz der Klasse *ThreadJob* aus. Somit kann durch sukzessives Zuweisen unterschiedlicher *ThreadJob*-Klassen (beziehungsweise davon abgeleiteter Klassen) derselbe Ausführungsfaden für verschiedene Aufgaben wiederverwendet werden[6].

Dies ist insbesondere wichtig, da sich bei der Ausführung der durch *Repetition*-Objekte dargestellten Operationen Suche und Anwendung von Regeln abwechseln, aber auch die zu verwendende Regel selbst wechselt. Eine Übersicht der beschriebenen Klassen ist in Abbildung 5.9 zu sehen. Des Weiteren verteilt *Repetition* auch die vorhandenen Partitionen auf die zur Verfügung stehenden Ausführungsfäden. Hierzu speichert *ThreadData* eine Liste von Partitionen.

Jede von *Repetition* abgeleitete Klasse repräsentiert wie oben beschrieben eine bestimmte Form der parallelen Abarbeitung von Graphersetzungsregeln. Im Folgenden werden die jeweiligen *Repetition*-Klassen der Reihe nach beschrieben.

RepetitionSingle

RepetitionSingle repräsentiert das wiederholte Ausführen einer einzelnen Regel. Die Regel wird solange ausgeführt, bis keine Instanzen des Musters mehr im Arbeitsgraphen vorhanden sind, oder eine zusätzlich übergebene maximale Menge von Anwendungen durchgeführt wurde.

Der Regelanwendungsprozess wird durch Aufruf der Methode *Run* gestartet. *Run* sucht in einer Schleife nach Instanzen des Suchmusters im Arbeitsgraphen. Hierbei durchsucht jeder Ausführungsfaden die in seinem *ThreadData*-Objekt hinterlegten Partitionen des Graphen. Sobald jeder Faden eine Instanz des Suchmusters gefunden hat (beziehungsweise meldet, dass keine Instanzen vorhanden sind), werden die

[6]Das beschriebene Vorgehen entspricht dem Master/Worker-Entwurfsmuster. Parallele Entwurfsmuster sind in [Matt04] beschrieben.

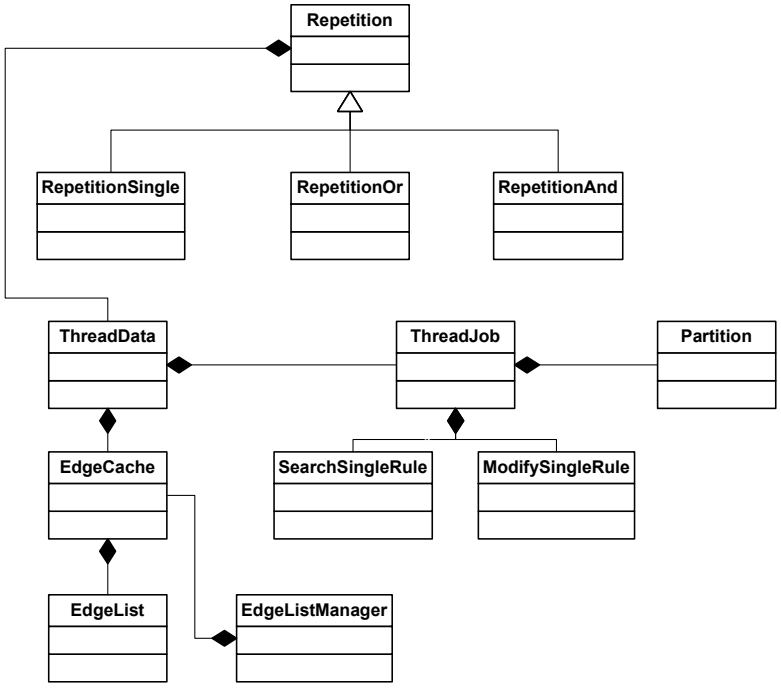

Abbildung 5.9: Klassendiagram von Repetition und verwandten Klassen.

Instanzen in der *Run*-Methode eingesammelt. Jede Instanz des Suchmusters wird wie bei der sequentiellen Variante von GrGen.NET durch eine Instanz der Klasse *Match* repräsentiert. In *Match* sind zusätzlich die IDs der Partitionen vermerkt, in denen sich die Elemente der Instanz befinden[7]. Nun werden die *Match*-Objekte der Reihe nach auf ihre Partitions-Belegung untersucht. Hierbei werden sukzessive alle Partitionen als gesperrt markiert, die ein *Match*-Objekt belegt. Liegt ein *Match*-Objekt in einer Partition, die bereits zuvor als gesperrt markiert wurde, so wird es verworfen. Sobald genauso viele *Match*-Objekte ihre Partitionen gesperrt haben, wie Ausführungsfäden vorhanden sind, werden diese *Match*-Objekte an die Fäden verteilt. Die Ausführungsfäden führen dann den Code zur Modifikation des Musters aus. Belegen mehrere *Match*-Objekte dieselben Partitionen, so kann nur eines der *Match*-Objekte ausgeführt werden, da beide potentiell gemeinsame Objekte besitzen und somit die Gefahr einer Wettlaufsituation besteht. Dieser Prozess wird nun wiederholt, bis keine *Match*-Objekte mehr gefunden werden können oder eine vom Aufrufer vorgegebene Anzahl von *Match*-Objekten ausgeführt wurde. Das beschriebene Verwerfen von *Match*-Objekten erhält die sequentielle Semantik: Wenn keine Elemente des Objektes verändert wurden, so wird das *Match* in einem späteren Durchlauf erneut gefunden und ausgeführt.

RepetitionOr

Die Klasse *RepetitionOr* repräsentiert eine logische Oder-Verknüpfung mehrerer Regeln, die solange wiederholt wird, wie die Ausführung erfolgreich ist, beziehungsweise bis eine vom Aufrufer angegebene maximale Anzahl an Iterationen durchgeführt

[7]Eine Instanz eines Suchmusters kann mehrere Partitionen überlappen, insbesondere auch solche, die anderen Ausführungsfäden zugeordnet sind.

wurde. Hierbei wird eine faule Auswertung angewendet, so dass eine einzelne erfolgreiche Regelanwendung die Ausführung als erfolgreich kennzeichnet und abbricht. *RepetitionOr* sucht mit der vorgegebenen Anzahl an Arbeiterfäden nach Instanzen des ersten Suchmusters auf dem Graphen. Die Auswahl der auszuführenden Instanzen erfolgt wie zuvor beschrieben. Wurde mindestens eine Instanz gefunden und erfolgreich ausgeführt, startet die Suche erneut. Das gleichzeitige Ausführen von n Instanzen des Suchmusters entspricht somit n sequentiellen Durchgängen der Oder-Verknüpfung. Die Semantik bleibt hierbei erhalten. Kann keine Instanz des ersten Suchmusters mehr gefunden werden, so wird in gleicher Weise nach Instanzen des zweiten Musters gesucht, bis auch hiervon keine Treffer mehr zu finden sind. So werden alle Regeln der Reihe nach abgearbeitet. Da eine erfolgreiche Anwendung der Regel $n + 1$ weitere Instanzen der zuvor gesuchten Regel n erzeugen kann, wird solange über die Menge aller Regeln iteriert, wie irgendeine Regel erfolgreich angewendet wurde. Erst sobald einmal keine der verknüpften Regeln mehr angewendet werden konnte, ist der Vorgang abgeschlossen. Die Ausführungsreihenfolge weicht von einer mehrfachen Oder-Verknüpfung in Programmiersprachen ab: Wurde keine Instanz der ersten Regel mehr gefunden, so wird solange nach Mustern der zweiten Regel gesucht, bis auch hiervon keine Instanzen mehr vorhanden sind, anstatt nach einer erfolgreichen Anwendung wieder nach der ersten Regel zu suchen. Dies ist erlaubt, da die Reihenfolge der Regelanwendung in einem Graphersetzungssystem indeterministisch ist.

RepetitionAnd

`RepetitionAnd` führt eine wiederholte Und-Verknüpfung von zwei Regeln durch. Der Ausdruck $A\&B$ wird solange wiederholt auf den Arbeitsgraphen angewendet, bis A und B nicht mehr nacheinander erfolgreich ausgeführt werden konnten. Hierzu wird der Graph in zwei Gebiete unterteilt, die Partitionen des Graphen also in zwei Mengen unterteilt. Der erste Teilgraph wird dann nach Regel A, der zweite gleichzeitig nach Regel B durchsucht[8]. Sind beide Suchvorgänge erfolgreich, werden beide gefundenen Regelinstanzen ausgeführt. Konnte für eines der beiden Muster keine Instanz gefunden werden, wird der andere Teilgraph durchsucht. Wird dann eine Instanz gefunden, muss geprüft werden, ob sich beide Instanzen in Ihrer Anwendung behindern. Konnte auf diese Weise keine Übereinstimmung gefunden werden, wechseln die durchsuchten Gebiete zwischen den Regeln.

Löschen und Erzeugen von Kanten

Wie in 5.1.2.2 beschrieben wurde, werden Kanten in GrGen.NET in drei doppelt verketteten Listen gespeichert: In der Liste der eingehenden Kanten des Zielknotens, der Liste der ausgehenden Kanten des Quellknotens, sowie in der globalen, nach Kantentyp geordneten Liste. Ein Kantenlöschvorgang muss dementsprechend auch alle drei Kantenverweise entfernen.

Das Entfernen der ein-/ausgehenden Kantenverweise in den Knoten kann parallel durchgeführt werden: Da die Partitionen, in denen die zu einer Kante gehörenden Knoten liegen bereits gesperrt sind, greift auch nur der zugehörige Ausführungsfaden auf die Objekte der Partition zu und hat somit exklusiven Zugriff auf die Knoten und deren Verweislisten.

[8] Die Suche kann in jedem der beiden Teilgraphen auf mehrere Arbeitsfäden verteilt werden.

Anders liegt die Situation bei der globalen, nach Kantentyp aufgeteilten Kantenliste: Da Kanten in der vorliegenden Implementierung nicht auf Partitionen verteilt werden, sondern beim übergeordneten Graph-Objekt verbleiben, muss der Zugriff auf diese Kantenlisten synchronisiert werden. Die Synchronisation des Zugriffs auf diese Kantenlisten kann sich jedoch negativ auf die Leistung des Systems auswirken: Wenn mehrere Ausführungsfäden gleichzeitig Kanten des gleichen Typs entfernen möchten, staut sich die Verarbeitung beim notwendigerweise sequentiellen Zugriff auf die Kantenlisten.

Die Lösung dieses Problems nutzt aus, dass die globalen Kantenlisten bei der Verarbeitung mit Partitionen nicht benötigt werden: Kanten werden nie durch Iteration über einen Kantentyp besucht, sondern immer durch Überprüfung der ein-/ausgehenden Kantenlisten von Knoten. Daher ist es akzeptabel, dass das Entfernen von Kanten aus der globalen Liste verzögert geschieht. Genauso verhält es sich bei neu erzeugten Kanten.

Damit gelöschte beziehungsweise neue Kanten verzögert in der globalen Liste aktualisiert werden können, besitzt jede Instanz von *ThreadData*, die einen Ausführungsfaden kapselt, eine Referenz auf ein Objekt vom Typ *EdgeCache*. *EdgeCache* besitzt zwei Listen: Eine für neu erstellte Kanten, eine für gelöschte Kanten. Die neuen Kanten werden gleich in Form einer dynamisch verketteten Liste abgespeichert, da diese später dann in $O(1)$ in die globale Kantenliste eingehängt werden können. Sobald die Listen in *EdgeCache* eine gewisse Größe erreicht haben, wird das Objekt an die als Singleton implementierte Klasse *EdgeListManager* übergeben. *EdgeListManager* verwaltet selbst einen Ausführungsfaden, der empfangene *EdgeCache*-Objekte verarbeitet und die darin enthaltenen Kanten in die globale Liste einfügt, beziehungsweise entfernt. Diese Operationen laufen somit ab, ohne den Arbeitsablauf der übrigen Ausführungsfäden zu verzögern. Der Zusammenhang der Klassen ist in Abbildung 5.9 zu sehen.

Löschen von Knoten

Das Löschen von Knoten kann parallel ausgeführt werden, da die Partition, in der Knoten liegt, zum Löschzeitpunkt immer gesperrt ist und somit nur von einem Ausführungsfaden verwendet wird. Komplexer wird die Situation allerdings, wenn der zu löschende Knoten noch Kanten besitzt, die zu Knoten anderer Partitionen führen. Um die Kanten aus den Listen der verbundenen Knoten zu entfernen, müssen deren Partitionen gesperrt werden. Geschieht dies erst direkt vor dem Löschen, also während des bereits gestarteten Ersetzungsprozesses, so sind Verklemmungen möglich, wenn zwei Ausführungsfäden jeweils versuchen, eine Partition zu sperren, die der jeweils andere bereits blockiert. Um dies zu vermeiden, müssen die Partitionen bereits während der Mustersuche im *Match*-Objekt vermerkt werden. Hierzu muss die Mustersuche so angepasst werden, dass bei Knoten, die später gelöscht werden, alle Kanten untersucht werden und die Partitionen der Endknoten in die Liste der für das *Match* zu sperrenden Partitionen aufgenommen werden. Somit können bei den oben beschriebenen Verfahren zur parallelen Graphersetzung mittels Partitionen keine Verklemmungen auftreten.

Für die Implementierung der beschriebenen Lösung ist es notwendig, dass der Codegenerator weiß, welche Knoten später gelöscht werden. Da der Generator des Codes

für die Mustersuche getrennt vom Generator der Musterersetzung operiert, weiß dieser nicht, welche Knoten später gelöscht werden sollen. Allerdings wird der Code für die Musterersetzung zuerst generiert. Somit konnte dieser Generator so angepasst werden, dass er diese Informationen für den Generator der Mustersuche zugänglich macht.

5.3 Umsetzung der parallelen Mustersuche

In diesem Abschnitt wird die Parallelisierung der Mustersuche von GrGen.NET behandelt. Zunächst werden die sequentiellen Suchmethoden von GrGen.NET beschrieben. Anschließend wird erläutert, wie diese parallelisiert wurden. Hierbei wird die Suche auf einem unpartitionierten Graphen behandelt. Abschnitt 5.3.1 erläutert hierzu die Implementierung der sequentiellen Mustersuche, 5.3.2 beschreibt die vorgenommene Parallelisierung.

5.3.1 Sequentielle Suchmethoden in GrGen.NET

Inhalt dieses Abschnitts ist die sequentielle Mustersuche in GrGen.NET. 5.3.1.1 beschreibt zunächst die Schnittstelle, die ein Entwickler verwenden kann, um die Mustersuche in eigenen Programmen zu nutzen. 5.3.1.2 zeigt anschließend, wie diese Instrumente implementiert sind.

5.3.1.1 Anwendung

In GrGen.NET wird jede Graphersetzungsregel durch eine eigene .NET-Klasse dargestellt. Alle Regelklassen erben von der gemeinsamen Oberklasse *LGSPAction*. Diese bietet eine Reihe verschiedener Methoden, mit denen einer Mustersuche gestartet werden kann. Eine Übersicht gibt Tabelle 5.2.

Tabelle 5.2: Graphersetzungsmethoden von LGSPAction

Match	Finde eine Musterinstanz.
Modify	Wende eine Musterinstanz auf den Arbeitsgraphen an.
ModifyAll	Wende alle übergebenen Musterinstanzen auf den Arbeitsgraphen an.
Apply	Finde eine Musterinstanz und wende Sie auf den Arbeitsgraphen an.
ApplyAll	Finde zuerst alle Musterinstanzen, wende diese anschließend auf den Arbeitsgraphen an.

Die Methoden aus Tabelle 5.2 rufen die Mustersuch-Methode *Match* auf, um die gefundenen Instanzen des Musters anschließend weiterzuverarbeiten. *Match* übernimmt als Parameter eine Referenz auf den zu durchsuchenden Arbeitsgraphen, sowie die Anzahl der maximal zu liefernden Musterinstanzen.

Die in Tabelle 5.2 gezeigten Methoden sind Teil der Programmierschnittstelle von GrGen.NET, werden allerdings auch bei der Verarbeitung eines GrShell-Skriptes verwendet. Somit bietet eine Parallelisierung dieser Methoden sowohl für Anwender der API als auch der GrShell potentiell Vorteile.

5.3.1.2 Umsetzung

Die Suchmethode *Match* ist in der Basisklasse *LGSPAction* implementiert und ruft einen Delegierer[9] auf. Dieser wird in den konkreten Klassen auf eine von GrGen.NET generierte Methode festgelegt. Diese generierte Methode ist das eigentliche Zentrum der Suchfunktion: Sie stellt die Implementierung eines Suchplanes dar. Die Implementierung eines Suchplans für das in Abbildung 5.10 zu sehende Muster könnte die Suche bei Knoten A beginnen. Da alle Knoten vom Typ A im Arbeitsgraphen als Ausgangspunkt für eine Instanz des Musters in Frage kommen, besitzt die Implementierung zunächst einen Schleife, die über alle Knoten vom Typ A iteriert. In dieser Schleife wird dann geprüft, ob der aktuelle A-Knoten ausgehende Kanten des Typs *K* besitzt. Ist dies der Fall, wird über diese iteriert, da wiederum jede dieser Kante zum Muster gehören könnte. Somit ergibt sich für die Implementierung eines Suchplans je nach Größe des Musters eine Methode, die aus vielen verschachtelten Schleifen besteht. Ein Beispiel für das Muster aus Abbildung 5.10 ist in Abbildung 5.11 zu sehen.

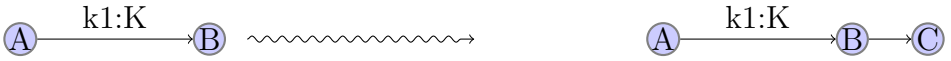

Abbildung 5.10: Eine einfache Graphersetzungsregel.

```
foreach(Node node_a in Nodes[TypeOf(A)])
{
  foreach(Edge e_k in node_a.EdgesOutgoind)
  {
    if (e_k.EdgeType == TypeOf(K) &&
        e_k.Target.NodeType == TypeOf(B))
    {
       // Regelinstanz gefunden
    }
  }
}
```

Abbildung 5.11: Pseudocode-Darstellung eines Suchprogramms für die Regel aus Abbildung 5.10.

5.3.2 Parallelisierte Suchmethoden

In diesem Abschnitt wird erläutert, wie die Suche in GrGen.NET parallelisiert wurde. In 5.3.2.1 wird beschrieben, wie sich diese Veränderung für den Anwender der API von GrGen.NET auswirkt. Anschließend wird in 5.3.2.2 erläutert, wie die Parallelisierung umgesetzt wurde.

5.3.2.1 Anwendung

Nach der Parallelisierung besitzen von *LGSPMatch* abgeleitete, generierte Klassen eine Methode *myMatchMT*. Diese hat die gleiche Funktion wie *myMatch* und ist somit eine weitere Implementierung des zur Regel gehörenden Suchplans. *myMatchMT*

[9]Ein Delegierer ist ein .NET-Objekt, welches einen Funktionszeiger kapselt. Somit kann eine erst zur Laufzeit festgelegte Methode aufgerufen werden.

wurde dabei so verändert, dass eine parallele Suche ausgeführt werden kann. Auch für die in Tabelle 5.2 gezeigten Methoden wurden parallele Versionen erstellt, beispielsweise *ApplyAllMT* für *ApplyAll*. Somit wurde die Anwendungsschnittstelle um parallele Methoden erweitert.

Eine neue Methode, die keine sequentielle Entsprechung hat, ist *MatchParallel*. Diese Methode kann mehrere Suchpläne auf demselben Arbeitsgraphen parallel abarbeiten.

Damit Anwendungen von GrGen.NET die parallelisierten Methoden verwenden können, müssen diese hierfür angepasst werden. Wie in Kapitel 6 zu sehen sein wird, kann es durchaus vorkommen, dass einzelne von Graphersetzungsregeln nicht parallel ausgeführt werden können. Daher wurde hier dem Anwender die Wahl gelassen, welche Methoden er für sein Zielsystem einsetzen möchte.

5.3.2.2 Umsetzung

Für die Implementierung der parallelisierten Version von GrGen.NET mussten einige Änderungen an der Klasse *LGSPAction* vorgenommen werden. Um dennoch zu alten Anwendungen von GrGen.NET kompatibel zu sein, wurden diese Änderungen allerdings in der Klasse *LGSPActionMT* zusammengefasst, die von *LGSPAction* erbt. Der Klassengenerator von GrGen.NET wurde so abgeändert, dass generierte Klassen nicht mehr von *LGSPAction*, sondern von *LGSPActionMT* erben. Die parallelen Versionen der Methoden aus Tabelle 5.2 werden in *LGSPActionMT* definiert und sind somit nicht generiert. Diese Methoden übernehmen zusätzlich zu den Befehlen ihrer sequentiellen Vorbilder die folgenden Aufgaben:

- Erzeugen von Ausführungsfäden für die parallele Suche
- Lastverteilung auf die Ausführungsfäden
- Verwalten der Suchergebnisse
- Beenden der Ausführungsfäden

Jedes Suchprogramm beginnt mit einer Schleife, die über alle Instanzen des Typs des zuerst gesuchten Graphelements iteriert. Die hier vorgenommene Parallelisierung entspricht Abschnitt 4.2.2 und versucht diese Menge von Graphelementen auf mehrere Ausführungsfäden zu verteilen. Hierfür wird die Methode *myMatchMT* generiert: Während die Methode *myMatch* über alle Elemente dieser äußersten Schleife iteriert, iteriert *myMatchMT* nur über einen Bereich dieser Menge.

Jeder arbeitende Ausführungsfaden führt somit während der Suche die Methode *myMatchMT* für eine Teilmenge der zu durchsuchenden Graphelemente aus. Dies ist in Abbildung 5.12 graphisch dargestellt. Hier ist insbesondere zu sehen, dass die Menge der zu überprüfenden Graphelemente genau den Elementen einer Typ-Ringliste entsprechen.

Der verwendete Algorithmus, um Elemente der Typ-Ringliste an mehrere Ausführungsfäden zu verteilen, folgt dem dynamischen Master/Worker-Entwurfsmuster: Zunächst werden vom Hauptfaden der Anwendung (*Master*) n weitere Ausführungsfäden (*Worker*) erzeugt und gestartet. Diese melden sich beim Hauptfaden und fordern weitere Blöcke von Graphelementen an, die dann jeweils von *myMatchMT* abgesucht werden. Jeder Arbeiterfaden speichert von ihm gefundene Ergebnisse lokal.

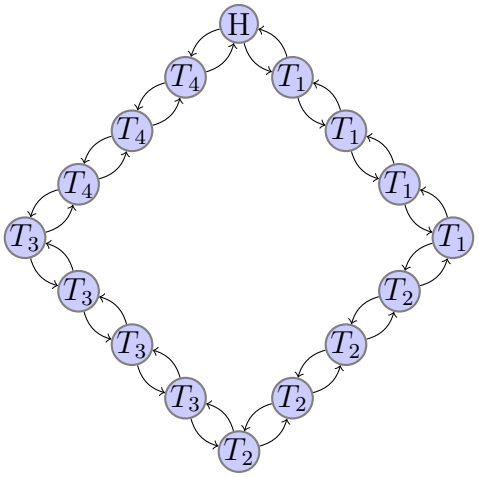

Abbildung 5.12: Die Elemente einer Typ-Ringliste werden auf mehrere Fäden verteilt. Die Knoten-Bezeichnungen entsprechen den zugewiesenen Fäden. H ist der Kopf der Ringliste.

Sobald alle Graphelemente verarbeitet wurden, teilt der Hauptfaden den Arbeiterfäden mit, dass sie sich beenden können und sammelt die jeweils lokal vorgehaltenen Ergebnisse ein. Diese werden dann zu einer einzigen Ergebnisliste zusammengefasst und zurück gegeben.

Um einem Ausführungsfaden eine Teilmenge k der zu verteilenden Graphelemente zukommen zu lassen, muss diese Menge zunächst bestimmt werden. Würden für das Speichern der Graphelemente Felder mit elementweisem Zugriff verwendet, könnte einem Ausführungsfaden jeweils der Start- und Endindex für seinen Bereich im Feld übergeben werden. Da jedoch dynamisch verkettete Listen eingesetzt werden, ist die Datenverteilung komplexer: Der Hauptfaden kennt lediglich den Kopf der zu verwendenden Ringliste und muss daher einen Zeiger auf das zuletzt an einen Arbeiterfaden vergebene Kettenglied halten, der bei der nächsten Anfrage um k Elemente nach vorne versetzt wird. Dies ist in Abbildung 5.13 veranschaulicht.

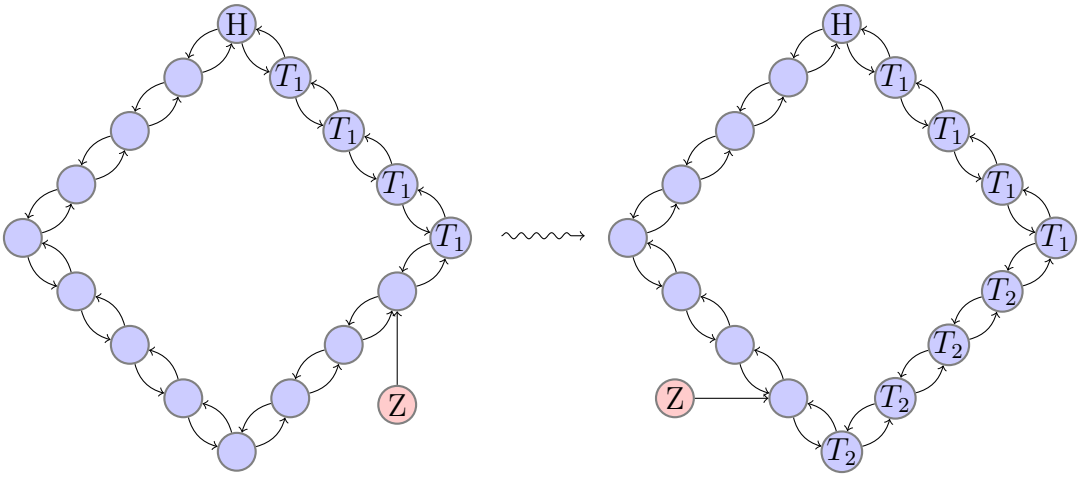

Abbildung 5.13: Die Elemente einer Typ-Ringliste werden auf mehrere Fäden verteilt. Hierzu wird ein Zeiger auf das zuletzt vergebene Element verwaltet.

Es entsteht somit im Vergleich zum sequentiellen Fall ein zusätzlicher Aufwand von $O(g)$ für den Hauptfaden, wobei g die Anzahl der zu verteilenden Graphelemente ist. Insgesamt wird die benötigte Typ-Ringliste nicht einmal, wie im sequentiellen Fall, sondern zweimal abgearbeitet: Einmal bei der Verteilung der Graphelemente und einmal segmentweise in den Arbeitsfäden bei der Suche.

Um diesen Aufwand zu minimieren, wurde pro Graphelement-Typ ein Feld eingeführt, welches n Referenzen auf Graphelemente beinhaltet, wobei n die Anzahl der Suchfäden repräsentiert. Diese Elemente dienen als Zeiger in die einzelnen Typ-Ringlisten, so dass beim Verteilen der Graphelemente auf die Suchfäden die Ringliste nicht mehr abgearbeitet werden muss, sondern vielmehr bereits eine vorgefertigte Einteilung der Ringliste zur Verfügung steht. Abbildung 5.14 verdeutlicht den Ansatz.

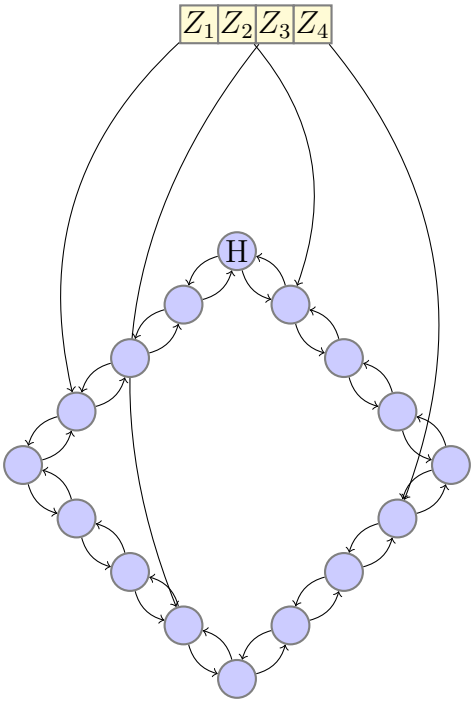

Abbildung 5.14: In einem Feld werden Zeiger auf Elemente der Knotenliste gespeichert. Somit können die an Arbeitsfäden zu vergebenden Bereiche leichter aufgefunden werden.

Damit dieser Ansatz funktionieren kann, sollten die Zeiger die Ringlisten in gleich große Abschnitte einteilen, um eine gerechte Verteilung auf die Suchfäden sicher zu stellen. Des Weiteren muss das Zeigerfeld aktuell gehalten werden, da Lösch- und Einfüge-Operationen die Qualität der Zeigerverteilung beeinflussen. Dies kann auf verschiedene Weise geschehen. Beispielsweise könnten neue Elemente abwechselnd hinter den Listenzeigern eingefügt werden. So wird jeder Abschnitt einer Liste in gleichem Maße erweitert. Eine andere Option wäre es, neue Elemente an einer Stelle einzufügen. Somit wird ein Abschnitt deutlich anwachsen. Nach einer vorgegebenen Anzahl neuer Elemente müssten die Zeiger dann komplett neu angelegt werden. In der Implementierung wurde diese Variante gewählt, da das Einfügen neuer Elemente so der bisherigen Strategie entspricht und keine unverhältnismäßigen Änderungen am System erforderte.

Der „Listen-Trick"

Eine in GrGen.NET verwendete Optimierungstechnik beruht auf der Annahme, dass Elemente, die bei einem Suchdurchgang gefunden wurden, auch bei späteren Suchvorgängen wiederverwendet werden. Beispielsweise könnte eine Regel an einem gefundenen Knoten eine Änderung durchführen, so dass dieser Knoten potentiell von einer anderen Regel weiterverarbeitet werden kann. Um diesen Zusammenhang auszunutzen, schiebt GrGen.NET die Köpfe der Graphelement-Typlisten, nachdem ein Match gefunden wurde, direkt vor die einzelnen gefundenen Graphelemente. Wird später wieder über alle Elemente eines zuvor gefundenen Graphelement-Typs iteriert, wird das bei der letzten Suche gefundene Element zuerst untersucht und somit potentiell früher ein Match gefunden. Dies kann die Laufzeit zum Auffinden eines Matches in günstigen Fällen von $O(n)$ auf $O(1)$ kürzen. Dieses Vorgehen wird von den GrGen.NET-Entwicklern als „Listen-Trick" bezeichnet.

Der Listen-Trick stellt jedoch ein Hindernis für die Parallelisierung dar: Da der Kopf der Ringlisten ein echtes Element der Liste ist, und nicht einfach ein Zeiger auf ein Element der Liste, wird beim Verschieben des Kopfes eine Änderung an der Liste vorgenommen. Da neue Elemente direkt vor dem Listenkopf eingefügt werden, verschiebt sich innerhalb der Liste die Position, an der neue Elemente eingefügt werden. Dies führt dazu, dass die oben beschriebenen Zeigerfelder nicht immer konsistent sind. Insbesondere ist beim Verteilen einer Ringliste anhand der Zeigerfelder auf mehrere Ausführungsfäden nicht bekannt, wo sich der Listenkopf aktuell befindet. Somit müssen alle Fäden bei jedem zu verarbeitenden Element prüfen, ob es sich um den Listenkopf handelt.

Um dieses Problem zu vermeiden, wäre eine Neuimplementierung der Ringlisten nötig, die keinen Listenkopf mehr innerhalb der Listen führt, sondern lediglich einen Zeiger auf ein Element der Liste verwaltet. Somit könnten beispielsweise auch der Einfügeort für neue Elemente und der Start einer Iteration durch zwei Zeiger unabhängig voneinander gehalten werden. Diese Änderung ist im Rahmen dieser Arbeit aufgrund des hohen Aufwandes nicht möglich. Daher wurde der Listen-Trick deaktiviert. Bei einer günstigeren Implementierung der Listenköpfe könnte auch die parallele Version die Geschwindigkeitsvorteile des Listen-Tricks ausnutzen. Durch das Entfernen des Tricks werden die effektiven Ausführungszeiten der sequentiellen wie auch der parallelen Version verlängert, das Verhältnis bleibt davon jedoch unbeeinflusst.

5.4 Modifikation des Code-Generators

Um die oben beschriebenen Parallelisierungsstrategien für GrGen.NET umzusetzen, war es nötig, Änderungen am Code-Generator (*grgen.exe*) vorzunehmen. Die Änderungen sollten dabei jedoch so vorgenommen werden, dass keine Funktionalität verloren geht, insbesondere sollten die sequentiellen Methoden von GrGen.NET weiterhin verwendbar sein. In diesem Abschnitt wird beschrieben, wie diese Änderungen vorgenommen wurden. Die beschriebenen Ansätze dürften sich bei späteren Erweiterungen des Code-Generators als hilfreich erweisen.

Der Code-Generator besteht aus zwei Subsystemen: Das in C# geschriebene „Backend" des Generators erzeugt den Code, der einen Arbeitsgraphen nach einem Muster durchsucht. Dieses System wird in 5.4.1 beschrieben. 5.4.2 zeigt anschließend,

welche Änderungen vorgenommen wurden. Das „Frontend", welches in Java entwickelt wurde, generiert Klassen zum Ersetzen von gefundenen Musterinstanzen. Die daran vorgenommenen Änderungen werden in 5.4.3 erläutert.

5.4.1 Aufbau des C#-Code-Generators

Der in C# geschriebene Teil des Code-Generators, als „Backend" bezeichnet, hat die Aufgabe, C#-Code für die Suche nach Mustern in Arbeitsgraphen auszugeben. Das Gegenstück des Backends ist das „Frontend": Es emittiert C#-Code, der eine gegebene Musterinstanz durch die rechte Seite der zugehörigen Regel ersetzt. Das Frontend wurde in Java geschrieben. Obwohl bei der Graphersetzung zunächst eine Musterinstanz aufgefunden werden muss, um diese anschließend zu ersetzen, wird beim Code-Generator zunächst das Frontend und anschließend das Backend ausgeführt. Der Ersetzungscode wird somit vor dem Suchcode erzeugt.

Um Suchcode zu erzeugen, verwendet das Backend den vom Frontend bereits erzeugten Code. Dieser beinhaltet neben dem Ersetzungscode auch die für das Graphmodell nötigen Knoten- und Kantenklassen; jede in einer Modelldatei (*.gm) definierte Knoten- oder Kantenklasse wird vom Frontend in eine .NET-Klasse übersetzt. Darüber hinaus benötigt das Backend die Regeldatei (*.grg), welche die Beschreibung der zu suchenden Muster enthält.

Der vom Backend erzeugte Code besteht aus vielen verschiedenen C#-Ausdrücken und Anweisungen: Um über alle Knoten eines gegebenen Typs zu iterieren, ist ein Schleifenkonstrukt nötig, während die Prüfung eines Knotenattributes eine if-Anweisung erfordert. Um diese verschiedenen Aufgaben zu verwalten, wurde für jede benötigte Elementaroperation eine passende Klasse in GrGen.NET angelegt. Alle diese Klassen erben von `SearchProgram`. Eine Menge von (verschiedenen) C#-Anweisungen kann als Menge von Objekten der Subklassen von `SearchProgram` ausgedrückt werden.

Da manche Befehle wiederum untergeordnete Befehle enthalten, wie beispielsweise eine if-Anweisung ihren Rumpf, muss auch dieser Zusammenhang ausgedrückt werden. Zu diesem Zweck wird die Klasse `SearchProgramList` verwendet, welche die selbe Basisklasse wie `SearchProgram` besitzt. Hat ein Befehl untergeordnete Befehle, so enthält er eine `SearchProgramList`, welche diese Befehle kapselt. Der Zusammenhang ist in Abbildung 5.15 dargestellt.

Eine zu erzeugende Suchfunktion wird als (Syntax-)baum von `SearchProgramOperation`-Objekten dargestellt. Ein Beispiel ist in Abbildung 5.16 zu sehen. Soll nun die Suchfunktion ausgegeben werden, wird die Methode `Emit` des Wurzelelements aufgerufen. Diese emittiert ein Codefragment. Des Weiteren ruft sie die Emit-Methode des nächsten Elements im Baum auf, oder, falls vorhanden, eines untergeordneten Elements. Somit wird der Baum sukzessive von der Wurzel bis zu den Blättern durchlaufen und der gesamte Quellcode erzeugt.

5.4.2 Modifikation des Backends

In 5.4.1 wurde gezeigt, wie ein Baum von `SearchProgramOperation`-Objekten verwendet wird, um den Code für ein Suchmuster zu erzeugen. Für jede Graphersetzungsregel wird einmal ein solcher Baum erstellt und auch einmal (beim Erzeugen

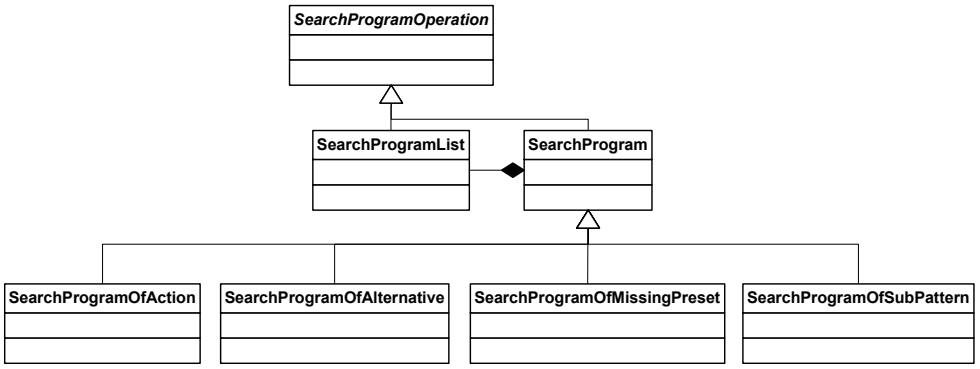

Abbildung 5.15: SearchProgram und verwandte Klassen.

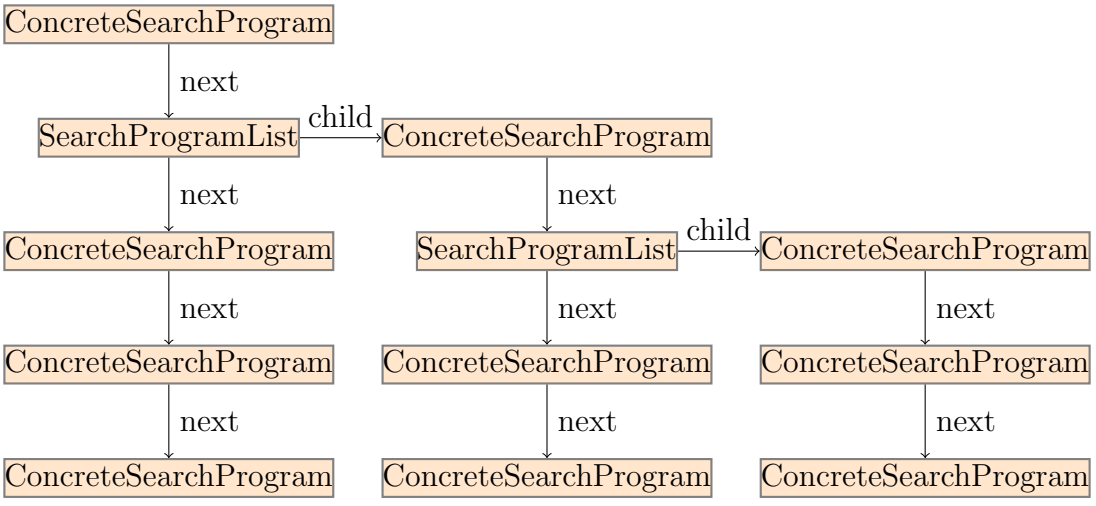

Abbildung 5.16: Beispiel für einen Baum aus SearchProgramOperations.

Tabelle 5.3: Alte und Neue Implementierungen von SearchProgram.

ursprüngliche Klasse	neue Klasse
AbandonCandidate(Global)	AbandonCandidate(Global)MT
AcceptCandidate(Global)	AcceptCandidate(Global)MT
CheckForIsomorphy(Global)	CheckForIsomorphy(Global)MT
GetCandidateByIteration	GetCandidateByIterationMT
GetCandidateByIteration	GetCandidateByPartitionIteration
SearchProgramOfAction	SearchProgramOfActionMT
SearchProgramOfAction	SearchProgramOfActionPartition
SearchProgramPrintLocalVars	SearchProgramPrintLocalVarsMT
-	SearchProgramDoNothing

des Codes) traversiert. Für eine parallelisierte Suche sind Änderungen an diesem Code nötig: Solche Änderungen sind möglich, indem der Baum bearbeitet wird, was jedoch erst nach der Traversierung durchgeführt wird: Somit wurde der sequentielle Code zum Zeitpunkt der Änderungen bereits emittiert, eine sequentielle Suche ist daher auch mit der modifizierten GrGen.NET-Version möglich. In dieser Arbeit werden zwei verschiedene parallele Suchstrategien implementiert: Parallele Suche auf partitionierten und unpartitionierten Graphen. Daher sind auch zwei parallele Suchmethoden nötig. Hierzu wird der Baum zweimal verändert und nach jeder Änderung traversiert. Insgesamt sind somit für jede Graphersetzungsregel drei Suchmethoden vorhanden.

Zunächst wurde geprüft, welche Subklassen von `SearchProgram` für die parallelen Suchmethoden verändert werden müssen. Für jede dieser Klassen wurde eine neue Subklasse von `SearchProgram` angelegt, welche den parallelisierten Code-Abschnitt emittiert. Eine Übersicht gibt Tabelle 5.3. Der Syntaxbaum wird nun traversiert. Wird hierbei eine Instanz der „alten" Klassen gefunden, so wird diese durch eine Instanz der zugehörigen „neuen" Klasse ersetzt. Die Konstruktoren der neuen Klassen wurden so entworfen, dass diese eine Instanz der alten Klassen als Eingabe erhalten können und die alten Eigenschaftswerte übernehmen. So bleiben die genauen Parameter, die zur Codeerzeugung nötig sind, erhalten. Anschließend kann durch Aufruf von `Emit` der parallele Code erzeugt werden.

5.4.3 Modifikation des Frontends

Um neue Knoten direkt in Partitionen einzufügen, anstatt in den Hauptgraphen, sind Änderungen am Frontend des Code-Generators von GrGen.NET nötig. Diese werden in diesem Abschnitt beschrieben.

Das in Java geschriebene Frontend verzichtet auf das Erzeugen eines Syntaxbaumes zur Code-Erzeugung: Dies ist möglich, da die einzelnen Elementaroperationen nur in geringem Maße Abhängigkeiten zwischen einander besitzen. Beispielsweise ist zwischen den Löschoperationen, um mehrere Knoten zu entfernen, keine feste Reihenfolge definiert. Das Frontend besitzt daher eine Methode `genModifyRuleOrSubRule`, welche für jede Elementaroperation eine Subroutine aufruft. Diese Subroutinen schreiben dann den Code für die zugehörige Elementaroperation so oft wie nötig: Beispielsweise ist solch eine Subroutine für die Elementaroperation „Lösche Knoten" vorhanden. Müssen in einem Muster 5 Knoten gelöscht werden, wird diese Methode 5 mal den passenden Code emitieren.

Wie im Backend muss auch im Frontend der emittierte Code für einige Elementaroperationen verändert werden. Hierzu wurde die Methode `genModifyRuleOrSubRule` kopiert (und umbenannt), so dass durch deren Aufruf eine zweite Ersetzungsmethode generiert wird. Für die zu verändernden Elementaroperationen wurden ebenfalls neue Versionen angelegt, so dass ein zweiter Codegenerierungspfad zur Verfügung steht.

6. Evaluierung

In diesem Abschnitt werden die Ergebnisse der Implementierung beschrieben. Hierzu gehört auch der im Rahmen dieser Arbeit entwickelte Genexpression-Benchmark. Er wird in 6.1.1 beschrieben. Die Leistungsmessung ist in 6.1.2 zu finden. 6.2 zeigt zusätzlich die Messergebnisse eines Standard-Beispiels von GrGen.NET.

6.1 Der Genexpression-Benchmark

6.1.1 Beschreibung des Genexpression-Benchmarks

Der Genexpression-Benchmark simuliert die in Abschnitt 2.5 beschriebenen biologischen Prozesse. Hierzu zählen die Transkription von mRNA anhand von DNA, sowie das Erzeugen der passenden Aminosäure-Ketten aus der mRNA (Translation). Die verwendete DNA besteht aus einem 55.000 Basen langen Abschnitt der DNA von Escherichia Coli (Stamm K12). Das *E.coli*-Genom kann unter [ecoc] eingesehen werden. Die erzeugten Aminosäure-Ketten entsprechen den korrekten Ketten, die auch in lebenden *E.coli* Zellen vorkommen. Eine Liste der in diesem Benchmark exprimierten Gene ist in Tabelle 6.1 zu finden.

Das GrGen.NET-Modell

Der Kern des Benchmarks besteht aus einer Modelldatei und einer Datei für die grundlegenden Graphersetzungsregeln. Das Modell wird in Abbildung 6.1 dargestellt.

Die Knotenklasse *Nucleotide* bildet die Basis für Nucleotide. Von Ihr leiten sich die Knotenklassen *Adenin*, *Guanin*, *Cytosin*, *Thymin* sowie *Uracil* ab. *Nucleotide* ist mit einem Attribut *NucType* ausgestattet, dass den konkreten Typ eines Nucleotids angibt. Dies ist in den Graphersetzungsregeln nötig, um den konkreten Typ eines Nucleotid-Knotens zu bestimmen. *NucType* wird bereits beim Erzeugen eines Nucleotid-Knotens korrekt initialisiert.

Innerhalb einer DNA-Kette sind Abschnitte zu finden, die als Gene bezeichnet werden. Diesen vorangestellt sind die Promotor-Regionen des entsprechenden Gens. Die-

Tabelle 6.1: Gene des Benchmarks

Gen	Länge (bp)	Startposition	Endposition	Promotorlänge (bp)
PyrH	726	191,855	192,580	91
frr	558	192,872	193,429	76
dxr	1197	193,521	194,717	144
ispU	762	194,903	195,664	120
cdsA	858	195,677	196,534	143
bamA/YaeT	2433	197,928	200,360	145
hlpA	486	200,482	200,967	85
lpxD	1026	200,971	201,996	131
fabZ	456	202,101	202,556	105
lpxA	789	202,560	203,348	63
lpxB	1149	203,348	204,496	97
rnhB	597	204,493	205,089	75
dnaE	3483	205,126	208,608	91
accA	960	208,621	209,580	140
ldcC	2142	209,679	211,820	120
yaeR	390	211,877	212,266	68
tilS	1299	212,331	213,629	127
yaeQ	546	214,291	214,836	68
yaeJ	423	214,833	215,255	128
nlpE	711	215,269	215,979	77
gmhB	576	222,833	223,408	64
dkgB	804	229,167	229,970	105
yafD	801	231,122	231,922	110
yafE	624	231,926	232,549	135
yafS	723	234,816	235,538	100
dnaQ	732	236,067	236,798	95
yafT	786	237,335	238,120	111
ivy	474	240,343	240,816	121
lpcA	579	243,543	244,121	68
yafJ	768	244,327	245,094	77

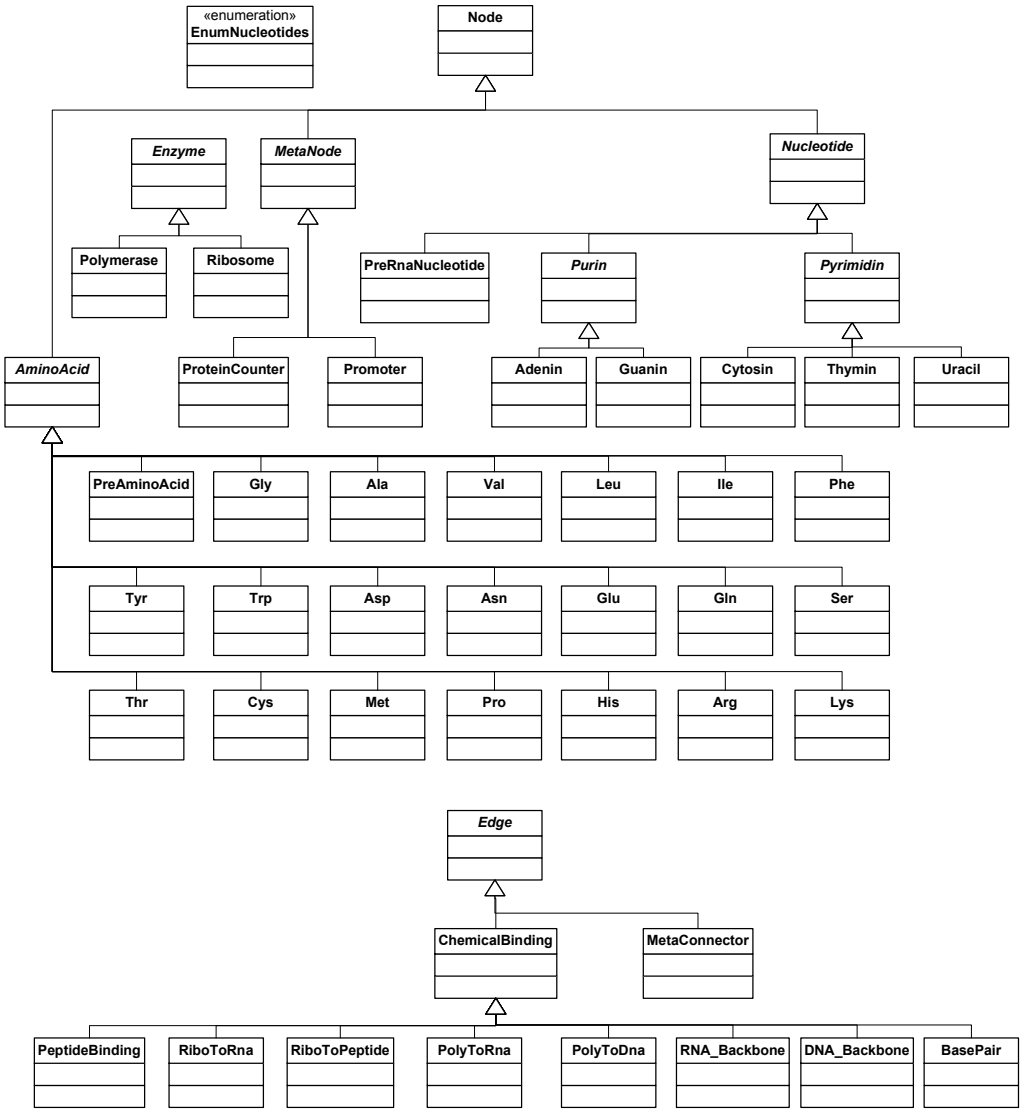

Abbildung 6.1: Das Modell des Genexpressions-Benchmarks als Klassendiagramm.

se werden in der Natur durch eine komplexe chemische Reaktion mit diversen Translationsfaktoren, dem Polymerase-Komplex und weiteren Proteinen identifiziert. Diesen Prozess exakt zu simulieren ist im Rahmen dieser Arbeit nicht möglich, insbesondere, da die Promotoren einem sehr unscharfen Nucleotid-Muster folgen, dass mit den gegenwärtigen Fähigkeiten von GrGen.NET nicht erfasst werden kann.

Um Promotoren innerhalb des Benchmarks zu identifizieren, werden diese im DNA-Graphen durch zusätzlichen Knoten markiert. Solche Marker-Knoten befinden sich jeweils an Nucleotiden, die dem ersten Nucleotid eines Gens vorhergehen. Sie besitzen ein Feld, welches die Länge des Gens angibt. Diese Längenangabe wird durch entsprechende Regeln in Polymerase-Knoten übertragen, so dass diese bei der Translation das Ende des Gens nicht überschreiten. In der Biologie wird dies durch die in 2.5.3.1 beschriebenen Beendigungsmechanismen umgesetzt, die den Rahmen dieser Arbeit jedoch überschreiten und zusätzliche Erweiterungen an GrGen.NET voraussetzen würden. Wie die Promotoren in der DNA aufgefunden werden, wird im nächsten Abschnitt erläutert.

Beim Erzeugen der RNA ist ein Zwischenschritt notwendig: Aufgrund von Einschränkungen der GrGen.NET Sprache kann eine Übersetzung von DNA- zu RNA-Nucleotiden nicht in einem einzigen Schritt erfolgen. Vielmehr werden zunächst Knoten des Typs PreRnaNucleotide erzeugt, welche in einem Attribut ihren konkreten Nucleotid-Typ vermerken. Durch weitere Regelanwendungen werden diese Knoten dann in konkrete Nucleotide umgewandelt. Der gleiche Prozess findet bei der Erzeugung von Aminosäuren anhand von RNA statt.

Ersetzungsregeln in GrGen.NET

Der Genexpressions-Benchmark beinhaltet 39 Graphersetzungsregeln. Hinzu kommt pro verwendetem Promotor eine weitere Regel zum Auffinden und Markieren des Promotors innerhalb der DNA. In dieser Arbeit wurden 30 Promotoren verwendet. Eine Übersicht der Regeln gibt Tabelle 6.2.

Die Regeln lassen sich in drei Gruppen einteilen: Erzeugen von RNA anhand eines DNA-Stranges, Erzeugen von Aminosäure-Ketten anhand von RNA sowie das Auffinden von Promotoren. Die wesentlichen Werkzeuge des Graphersetzungssystems, die zur Umsetzung nötig waren, ist die Möglichkeit, Attribute von Knoten auszulesen und zu verändern sowie die Möglichkeit, Knoten einen neuen Typ zu geben. Die Regeln IncProteinCount sowie AttachProteinCounterAUG/UUG/GUG ermöglichen es zu zählen, wie oft eine RNA-Kette bereits in Aminosäuren abgeschrieben wurde. Somit kann erkannt werden, wann jedes Gen 15 mal umgewandelt wurde und der Benchmark kann sich anschließend beenden.

Von besonderem Interesse sind die Regeln zum Auffinden von Promotoren: Sie suchen nach einer Folge von Nucleotiden, die innerhalb des Genoms eindeutig ist und exakt vor dem ersten Nucleotid des zugehörigen Gens endet. Die Regeln fügen dann einen Marker-Knoten an diese Stelle an, so dass die aufwendige Suche nicht wiederholt werden muss. Die Länge der Promotor-Muster liegt zwischen 60 und 150 Nucleotiden. Wie diese langen Regeln erzeugt wurden, erläutert der folgende Abschnitt.

Erzeugen der Promotor-Daten

Die exakten Nucleotidsequenzen, die vor einem Gen liegen, können von der Webseite [ecoc] bezogen werden. Da die Längen der verwendeten Promotorsequenzen sehr

Tabelle 6.2: Graphersetzungsregeln des Benchmarks

Funktionsname	Beschreibung
Insert_RNA_Polymerase	Fügt eine Polymerase in den Arbeitsgraphen.
Insert_Ribosome	Fügt ein Ribosom in den Arbeitsgraphen ein.
Attach_RNA_Pol_After_Promoter	Bindet eine freie Polymerase an einen Promotor.
RNA_Pol_Produce_Initial	Erzeugt das erste Nucleotid einer neuen RNA-Kette.
RNA_Pol_Produce_Following	Erzeugt die weiteren Nucleotide einer RNA-Kette.
RNA_Pol_Finish	Erzeugt das letze Nucleotid einer RNA-Kette und setzt die Polymerase wieder frei.
PreRnaToRna_Adenin	Wandelt Pre-Rna-Knoten vom Typ Adenin in Uracil.
PreRnaToRna_Thymin	Wandelt Pre-Rna-Knoten vom Typ Thymin in Adenin.
PreRnaToRna_Guanin	Wandelt Pre-Rna-Knoten vom Typ Guanin in Cytosin.
PreRnaToRna_Cytosin	Wandelt Pre-Rna-Knoten vom Typ Cytosin in Guanin.
AttachRibosome	Fügt ein Ribosom an eine RNA-Kette an.
AttachProteinCounterAUG	Fügt einen Instanz-Zähler-Knoten an die Folge AUG an.
AttachProteinCounterUUG	Fügt einen Instanz-Zähler-Knoten an die Folge UUG an.
AttachProteinCounterGUG	Fügt einen Instanz-Zähler-Knoten an die Folge GUG an.
IncProteinCount	Erhöht den Wert eines Instanz-Zählers.
CreateInitialAmino	Erzeugt die erste Aminosäure einer neuen Kette.
CreateFollowingAminos	Erzeugt die weiteren Aminosäuren einer Kette.
EndTranslation	Erzeugt die letzte Aminosäure einer Kette und setzt das Ribosom frei.
CreateAla/Arg/..	Erzeugt aus PreAmino-Knoten die korrekten Aminosäureknoten.

lang sind, wurde ein Tool entwickelt, welches GrGen.NET Regeln anhand von Promotorsequenzen generiert. Hiermit ist es auch möglich, weitere Gene (und nicht nur die 30 des aktuellen Benchmarks) in Graphersetzungsregeln zu überführen.

Anwendung der Graphersetzungsregeln

Um die oben beschriebenen Graphersetzungsregeln anwenden zu können, ist zunächst ein Arbeitsgraph mit geeigneter DNA notwendig. Die DNA besteht aus dem *E coli*-Genom, Abschnitt 191,855 bis 245,094[1]. Alle Regeln werden direkt aus einem C#-Programm heraus aufgerufen, die GrShell wird somit nicht verwendet. Die GrShell wäre durchaus verwendbar, allerdings sind die parallelen Ansätze, die in dieser Arbeit entwickelt wurden, noch nicht in die GrShell integriert. Daher wird auf deren Einsatz verzichtet und die Regeln über die API von GrGen.NET ausgeführt.

Zunächst muss die DNA in den Arbeitsgraphen geladen werden. Hierzu wurde eine Methode geschrieben, die eine Textdatei mit DNA-Daten im FASTA-Format ([fast]) als Argument übernimmt und die Nucleotide in entsprechende Knoten übersetzt. Anschließend können die Regeln ausgeführt werden.

Die Reihenfolge der Regelausführung orientiert sich am logischen Ablauf des Benchmarks, wobei natürlich auch eine rein zufällige Ausführung möglich wäre. Regeln, die „zu früh" ausgeführt werden, bleiben in diesem Fall ohne Wirkung. Der grundlegende Ablauf ist in Abbildung 6.2 in Pseudocode-Notation skizziert.

```
1) Lese DNA-Sequenz.
2) Finde und markiere die Promotoren.
3) Insert_RNA_Polymerase(); Insert_Ribosome();
4) while (Attach_RNA_Pol_After_Promoter() == true) {
4.1) RNA_Pol_Produce_Initial();
4.2) while (RNA_Pol_Produce_Following() == true) { }
4.3) RNA_Pol_Finish();
4.4) PreRnaToRna_Adenin(); PreRnaToRna_Guanin(); ...;
4.5) AttachProteinCounterAUG(); AttachProteinCounterGUG(); ...;
4.6) while (AttachRibosome() == true) {
4.6.1) IncProteinCount();
4.6.2) CreateInitialAmino();
4.6.3) while ( CreateFollowingAminos() == true ||
             EndTranslation() == true) {} }
4.7) while(CreateAla() == true || CreateArg() == true) { } }
```

Abbildung 6.2: Pseudocode Notation des sequentiellen Genexpression-Benchmarks.

In Abbildung 6.3 ist die parallele Variante dieses Codes zu sehen. Regeln, die parallel ablaufen, sind hierbei entsprechend markiert.

Die Messergebnisse werden im nächsten Abschnitt präsentiert.

[1] Gerechnet in Basenpaaren.

```
1)  Lese DNA-Sequenz.
2)  Finde und markiere die Promotoren. (Parallel)
3)  foreach (Promotor) { Insert_RNA_Polymerase() }
4)  while (Attach_RNA_Pol_After_Promoter() == true) { }
5)  Partitioniere den Graphen.
6)  RNA_Pol_Produce_Initial(); (Parallel)
7)  RNA_Pol_Produce_Following(); (Parallel)
8)  RNA_Pol_Finish(); (Parallel);
9)  (PreRnaToRna_Adenin() || PreRnaToRna_Guanin() || ...); (Parallel)
10) (AttachProteinCounterAUG() || AttachProteinCounterGUG()
          || ...); (Parallel)
11) Foreach (ProteinCounter) Insert_Ribosome();
12) (IncProteinCount() || CreateInitialAmino() ||
          CreateFollowingAminos() || EndTranslation()); (Parallel)
13) (CreateAla() || CreateArg() || ...) (Parallel)
```

Abbildung 6.3: Pseudocode Notation des parallelen Genexpression-Benchmarks.

6.1.2 Messergebnisse des Genetik-Benchmarks

Die Messungen wurden in drei Teilen vorgenommen: 6.1.2.1 beschreibt die Ergebnisse der Promotorsuche, in 6.1.2.2 und 6.1.2.3 wird die Synthese von RNA-Molekülen sowie das anschließende Erzeugen von Proteinen thematisiert. Die verwendeten Testsysteme sind in Tabelle 6.3 zusammengefasst.

Da zur Parallelisierung von GrGen.NET Änderungen an der sequentiellen Version vorgenommen werden mussten, werden bei den Messungen jeweils zwei sequentielle Messungen angegeben. Die erste Messung gibt jeweils den Wert der ursprünglichen Version an, die Zweite (mit „*" markiert) den Wert der veränderten Version. Beide Referenz-Versionen sind nötig, da sich die Laufzeiten der beiden Systeme durch entfernte Optimierungen wie den Listen-Trick (5.3.2.2) teilweise deutlich unterscheiden.

6.1.2.1 Promotorsuche

Bei der Promotorsuche wird ein langes Muster von Nucleotiden auf einer DNA gesucht. Im Benchmark wird ein 55.000 Nucleotide umfassendes Teilstück des Genoms von *Ecoli* verwendet. Diese Zahl wurde nach Rücksprache mit den Entwicklern einiger anderer Graphersetzungssysteme gewählt, um eine Ausführung des Benchmarks auch auf deren Systemen zu ermöglichen. Eine Veröffentlichung des Benchmarks nach dieser Diplomarbeit ist geplant.

Die Promotorsuche ist auf einer DNA-Kette mit lediglich 55.000 Nucleotiden so schnell, dass keine aussagekräftigen Ergebnisse erzeugt werden können. Daher wird die Promotorsuche auf dem vollständigen, 4,6 Millionen Nucleotide langen *Ecoli*-Genom vorgenommen. Hierbei ist bei Rechner B unter Verwendung von vier Ausführungsfäden ein Speedup von 1,79 festzustellen. Rechner A verzeichnet mit zwei Kernen einen Speedup von 1,3. Die Tests wurden jeweils sequentiell sowie parallel mit 2,3,4 und 8 Fäden durchgeführt. Die Laufzeiten und der Speedup ist in Abbildung 6.4 zu sehen. Der Speedup wurde relativ zur unveränderten Version von GrGen.NET berechnet.

Tabelle 6.3: Die verwendeten Testsysteme

Kürzel	Rechner
A	Samsung R70; Intel Mobile Core 2 Duo T5250, 1,5Ghz, 2048KByte L2-Cache; 2GB RAM; Windows Vista SP1 32bit;
B	Intel Core 2 Quad Q6600, 2,4Ghz, 2x 4096KBytes L2-Cache; 4GB RAM; Windows Vista SP1 64bit;
C	2x Intel Xeon E5320, 1,86Ghz, 2x 8192KBytes L2-Cache; 8 GB RAM; Windows Server 2003 32bit; Physical Address Extension;
D	Samsung X20; Intel Pentium M 740, 1,73Ghz, 2048 KByte L2-Cache; 2GB RAM; Windows XP SP3 32bit;

Promotorsuche (30 Gene)

System	Sequentiell	Sequentiell*	2 Threads		3 Threads		4 Threads		8 Threads	
A	8.472 s	9.410 s	7.036 s	1,2x	6529,00	1,3x	6.272 s	1,4x	6.310 s	1,3x
B	10.646 s	11.721 s	7.638 s	1,4x	6464,00	1,6x	5.935 s	1,8x	5.903 s	1,8x
C	7.679 s	8.175 s	4.481 s	1,7x	4080,00	1,9x	3.707 s	2,1x	3.623 s	2,1x

Bei den Ausführungsfäden sind jeweils die Speedup-Werte im Vergleich zur sequentiellen (nicht veränderten) Ausführung angegeben.
Sequentiell* bezeichnet die veränderte Variante von GrGen.NET.

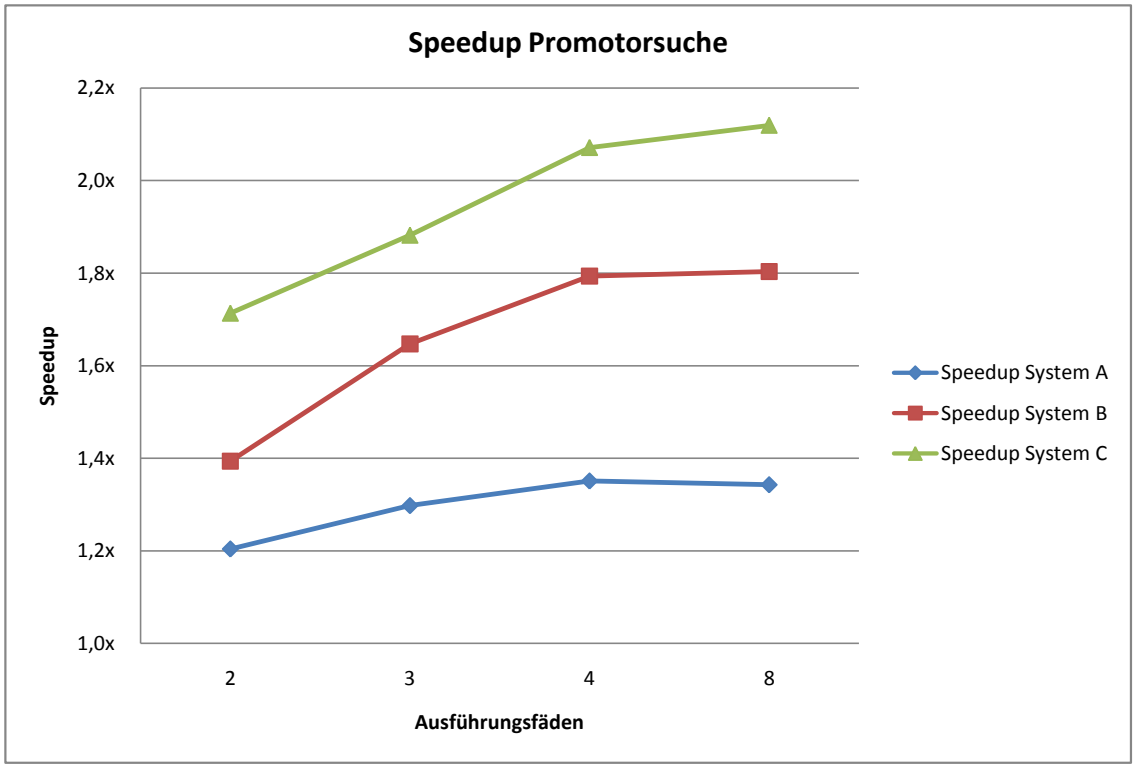

Abbildung 6.4: Messergebnisse der Promotorsuche.

Genexpression-Benchmark: RNA-Synthese (30 Gene)

System	sequentiell	sequentiell*	2 Threads		3 Threads		4 Threads		8 Threads	
A	55ms	77,9 s	54,5 s	1,4x	32,0 s	2,4x	27,9 s	2,8x	20,3 s	3,8x
B	89ms	79,9 s	56,6 s	1,4x	26,4 s	3,0x	20,7 s	3,9x	17,2 s	4,6x
C	67ms	49,5 s	25,7 s	1,9x	17,2 s	2,9x	5,9 s	8,4x	3,7 s	13,4x
D	46ms	109,0 s	92,1 s	1,2x	60,2 s	1,8x	56,4 s	1,9x	45,0 s	2,4x

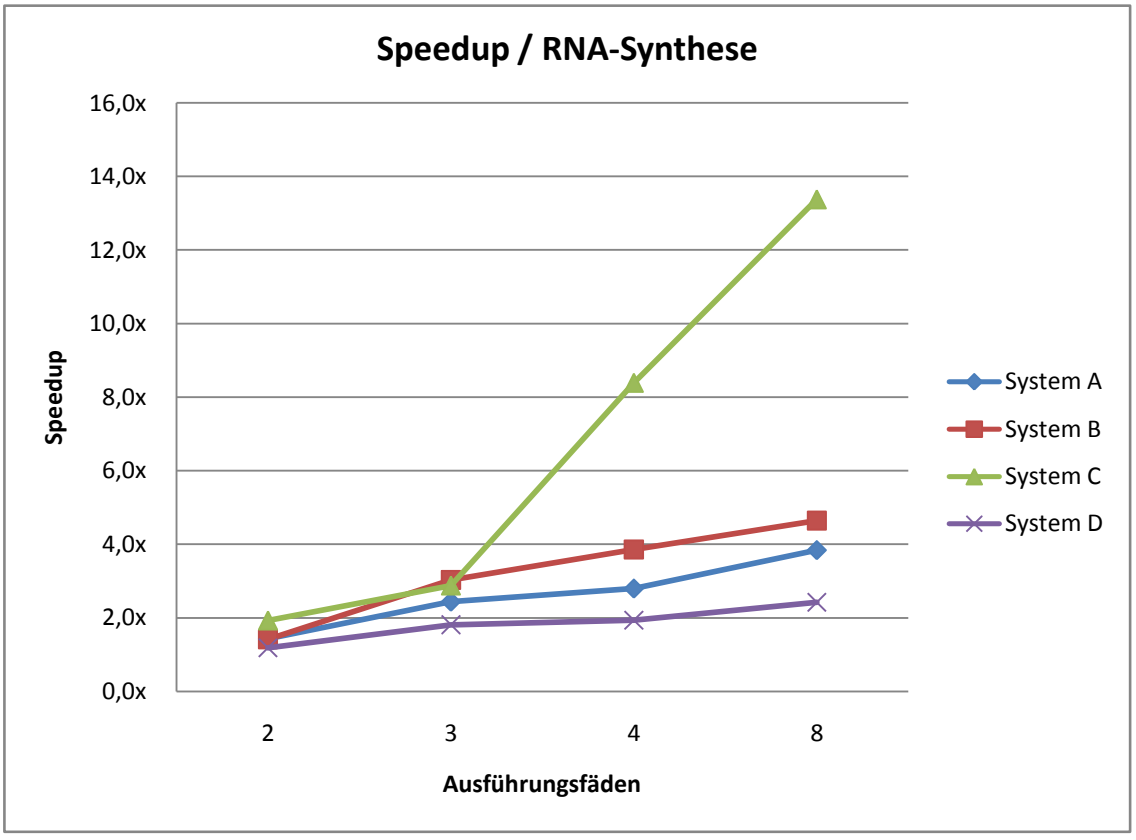

Abbildung 6.5: Messergebnisse der RNA-Synthese.

6.1.2.2 RNA-Synthese

Bei der RNA-Synthese werden Polymerasen an die zuvor markierten Promotoren angehängt. Diese bewegen sich dann entlang der DNA in Richtung des Gen-Endes und erzeugen jeweils neue Nucleotide, die zu einer RNA-Kette verknüpft werden. Ist das Ende eines Gens erreicht, löst sich die Polymerase von der DNA ab und gibt das RNA-Molekül, beziehungsweise die aus Knoten bestehende Kette, frei. Knoten, die einen Promotor markieren, werden beim Anfügen einer Polymerase gelöscht, so dass anhand jeden Gens nur ein RNA-Molekül synthetisiert wird.

Die Laufzeiten sowie Beschleunigungswerte sind in Abbildung 6.5 zu sehen. Zur Messung wurden erneut beide sequentielle Versionen von GrGen.NET verwendet. Wie den Messungen zu entnehmen ist, konnte im Vergleich zum ursprünglichen GrGen.NET keine Beschleunigung erzielt werden. Vielmehr ist die parallele Version deutlich langsamer. Der Speedup wurde jeweils relativ zur veränderten Version von GrGen.NET (in den Abbildungen mit „*" markiert) berechnet. Im Folgenden wird zunächst erläutert, weshalb die parallele Version in solchem Ausmaß langsamer ist und wie dies behoben werden kann. Anschließend werden die Ergebnisse genauer thematisiert.

Vergleich mit der sequentiellen Version von GrGen.NET

GrGen.NET wird seit mehreren Jahren regelmäßig weiterentwickelt. Ziel ist neben der Umsetzung zusätzlicher graphentheoretischer Konzepte die kontinuierliche Verbesserung der Ausführungsgeschwindigkeit. Solch ein System kann gewöhnlich nicht ohne Änderungen parallelisiert werden. Beispiele, die zeigen, dass der Überarbeitungs-Aufwand bei der Parallelisierung eine große Rolle spielt, sind in [PSJT08] zu finden. Auch an GrGen.NET waren im Rahmen dieser Arbeit Änderungen unumgänglich. Beispiele hierzu zeigt Abschnitt 5.4. Insbesondere war es im zeitlichen Rahmen dieser Arbeit nicht möglich, alle Optimierungen, die im Laufe der Zeit in GrGen.NET eingearbeitet wurden, bei der Parallelisierung beizubehalten oder erneut umzusetzen.

Eine Optimierung, die einen sehr großen Anteil an der Geschwindigkeit von GrGen hat, ist der in Abschnitt 5.3.2.2 behandelte „Listen-Trick". Dort wurde auch beschrieben, weshalb es nicht möglich war, diese Optimierung für die parallele Version von GrGen.NET zu übernehmen.

Der Listen-Trick hat große Auswirkungen auf die Ausführungsgeschwindigkeit bei der RNA-Synthese: Wird ein Treffer der RNA-Synthese-Regel gefunden, wird der Listenkopf der Nucleotid-Knotenkette direkt hinter das zuletzt gefundene Nucleotid gesetzt. Da beim nächsten Durchgang gleich das darauf folgende Element verarbeitet wird, kann dieses in $O(m+n)$ gefunden werden, wobei m der Startposition des Gens auf dem DNA-Molekül, n der Länge des Gens entspricht. Wird auf den Listen-Trick verzichtet, so muss bei jedem Verarbeitungsschritt die komplette Nucleotid-Liste bis zum aktuell zu verarbeitenden Nucleotid durchsucht werden. Der Suchaufwand kann wie folgt hergeleitet werden, wobei m und n wie oben definiert sind:

$$
\begin{aligned}
& m + (m+1) + (m+2) + ... + (m + (n-1)) + (m+n) \\
& = \textstyle\sum_{i=0}^{n} m + i \\
& = n \times m + \tfrac{n \times (n+1)}{2} \\
& = n \times (m + \tfrac{n+1}{2})
\end{aligned}
$$

Beginnt ein Gen beispielsweise an Position 1000 in einem DNA-Molekül und hat selbst eine Länge von 1000 Nucleotiden, so werden mit Listen-Trick 2000 Nucleotide, ohne jedoch mehr als 1,5 Millionen Nucleotide überprüft. Dieser Unterschied kann auf den Testsystemen von der parallelen Version nicht aufgeholt werden.

Beschleunigung bei Einkern-Systemen

Wie in Abbildung 6.5 zu sehen ist, kann eine Beschleunigung durch die Verwendung mehrerer Ausführungsfäden auch auf einem Rechner mit nur einem Kern erzielt werden. Insbesondere steigt die Ausführungsgeschwindigkeit mit höherer Anzahl an Fäden deutlich.

Ursache für die Beschleunigung auf Einkern-Rechnern ist die Partitionierung des Arbeitsgraphen: In 6.1.2.2 wurde beschrieben, dass sich die Anzahl der zu durchsuchenden Knoten bei der RNA-Synthese aus dem Weg m vom Start der Knotenliste bis zum aktuell verarbeiteten Knoten sowie der Genlänge n zusammen setzt. Durch eine Partitionierung wird der Weg m verkürzt, da die Menge der Knoten auf mehrere Listen verteilt wird. Sind weniger Ausführungsfäden als Partitionen vorhanden, so werden jedem Faden mehrere Partitionen zugeteilt. Je mehr Fäden vorhanden sind, um so geringer wird der Wert m. Die Anzahl der auszuführenden Operationen sinkt also mit steigender Anzahl an Partitionen und steigender Anzahl an Arbeitsfäden, wobei ein Optimum zu finden ist, wenn die Anzahl der Arbeitsfäden der Anzahl an Partitionen entspricht. Die Partitionierung des Arbeitsgraphen bewirkt somit ähnliche Effekte wie der Listen-Trick. Auf Einkern-Systemen ist eine Leistungssteigerung möglich, da eine höhere Zahl von Fäden bei vorgegebener Partitionierung die Anzahl der auszuführenden Operationen verringert.

Beschleunigung bei Mehrkern-Systemen

Für Mehrkern-Systeme gelten die gleichen Aussagen wie für Einkern-Systeme. Zusätzlich ist jedoch eine höhere Beschleunigung durch die Parallelverarbeitung zu sehen. Insbesondere fällt auf, dass der Leistungsgewinn mit der Anzahl an Arbeitsfäden und Kernen skaliert. Die Verbesserung durch Parallelverarbeitung sowie die Partitionen genügen jedoch nicht, um den fehlenden Listen-Trick auszugleichen. In zukünftigen Arbeiten sollte der Implementierung des Listen-Tricks daher einen hohen Stellenwert erhalten.

6.1.2.3 Erzeugen von Proteinen

Aktuell ist das Erzeugen von Proteinen mit der parallelen Version von GrGen.NET nicht lauffähig. Hierzu wären weitergehende Änderungen an GrGen.NET notwendig, die im Rahmen dieser Diplomarbeit nicht mehr durchführbar waren. Wie in

Abschnitt 5.4 beschrieben, mussten zur Codegenerierung Änderungen am Syntaxbaum von GrGen vorgenommen werden. Um das parallele Erzeugen von Proteinen zu ermöglichen, müssen einige weitere Knoten des Syntaxbaumes ausgetauscht werden, für die aktuell keine parallelen Entsprechungen existieren.

Die sequentielle Ausführung des Benchmarks ist jedoch möglich und erzeugt die korrekten Aminosäuresequenzen. Außerdem kann angegeben werden, wieviele Aminosäureketten pro vorhandenem RNA-Strang erzeugt werden sollen. Um dies zu ändern, ist allerdings eine Änderung an den Graphersetzungsregeln notwendig, die ein Neukompilieren der Bibliothek des Benchmarks nach sich zieht.

Da die Regeln wie auch der grundlegende Prozess, das Verarbeiten einer linearen Knotenkette zur Erzeugung einer neuen Kette, in Aufbau und Funktionalität große Ähnlichkeiten mit der RNA-Synthese aufweisen, sind für eine parallele Ausführung ähnliche Ergebnisse wie im RNA-Teil zu erwarten.

6.2 Die „Kochsche Schneeflocke"

Es sind verschiedene Standard-Benchmarks für Graphersetzungssysteme verfügbar. Mit GrGen.NET wurde beispielsweise ein *Busy-Beaver*-Benchmark veröffentlicht (siehe [BlGe07]). Viele dieser Benchmarks besitzen jedoch nur wenig Parallelisierungspotential und können somit kaum von einem parallelen Graphersetzungssystem profitieren. Daher wurde im Rahmen dieser Arbeit der zuvor beschriebene Genetik-Benchmark entwickelt. In diesem Abschnitt wird mit der „Kochschen Schneeflocke" zusätzlich ein Standard-Benchmark mit inhärenter Parallelität präsentiert.

Der hier vorgestellte Benchmark nutzt Graphersetzung um Fraktale nach dem Muster der „Kochschen Schneeflocke" zu generieren (siehe [BlGe07]). Der Ausgangsgraph besteht lediglich aus einem von drei Kanten und drei Knoten gebildeten Dreieck. Durch die Graphersetzungsregeln wird jeweils eine Kante des Dreiecks durch eine Kante, in die selbst wieder ein Dreieck eingebettet ist, ersetzt. Bei jedem Verarbeitungsschritt der Anwendung werden alle vorhandenen Kanten durch Kanten mit eingebettetem Dreieck ersetzt. Somit wächst die Anzahl der Graphelemente mit jedem Schritt exponentiell. Abbildung 6.6 zeigt die Entwicklung der „Kochschen Schneeflocke".

Eine Parallelisierung wird mittels Partitionierung vorgenommen: Hierzu wird nach einem gegebenen Verarbeitungsschritt eine Partitionierung vorgenommen und die Dauer des nächsten Verarbeitungsschrittes gemessen. Im nächsten Schritt können somit mehrere Kanten parallel ersetzt werden.

Da das Erzeugen der „Kochschen Schneeflocke" in hohem Maße vom Listen-Trick profitiert, konnte im Vergleich zur ursprünglichen Version von GrGen.NET keine Leistungssteigerung erzielt werden. Allerdings konnte relativ zur veränderten Version ein Speedup von bis zu 3,3 (auf System C) erzielt werden. In den Ergebnissen ist zu sehen, dass der Speedup sowohl mit der Anzahl an Arbeitsfäden, aber auch mit der Anzahl an verwendeten Partitionen steigt. Übersteigt die Anzahl der Fäden die Anzahl an Partitionen, kann keine zusätzliche Leistung erzielt werden. Die Ergebnisse sind in Tabelle 6.7 zusammengefasst.

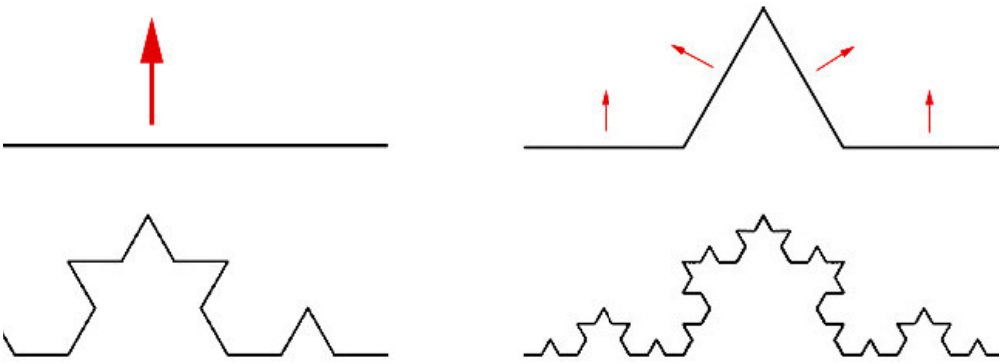

Abbildung 6.6: Entwicklung der „Kochschen Schneeflocke”.

Kochsche Schneeflocke

System	Messung	Partitionen	sequentiell	sequentiell*	2 Threads		3 Threads		4 Threads		8 Threads	
A	1	3	30ms	9,3 s	25,7 s	0,4x	8,6 s	1,1x	8,2 s	1,1x	8,4 s	1,1x
B	1	3	44ms	13,6 s	30,4 s	0,4x	9,3 s	1,5x	6,4 s	2,1x	7,5 s	1,8x
C	1	3	39ms	7,9 s	19,0 s	0,4x	3,2 s	2,5x	3,1 s	2,5x	3,4 s	2,3x
A	2	3	291ms	143,3 s	451,0 s	0,3x	147,0 s	1,0x	146,0 s	1,0x	146,0 s	1,0x
B	2	3	261ms	224,0 s	597,0 s	0,4x	186,0 s	1,2x	185,0 s	1,2x	185,0 s	1,2x
C	2	3	386ms	162,0 s	529,0 s	0,3x	146,0 s	1,1x	142,0 s	1,1x	141,0 s	1,1x
A	2	6	291ms	143,3 s	267,0 s	0,5x	165,0 s	0,9x	179,0 s	0,8x	98,5 s	1,5x
B	2	6	261ms	224,0 s	325,0 s	0,7x	147,0 s	1,5x	152,0 s	1,5x	93,5 s	2,4x
C	2	6	386ms	162,0 s	275,0 s	0,6x	98,0 s	1,7x	110,0 s	1,5x	48,0 s	3,4x

Messung 1: Die Schneeflocke wird von 12288 auf 49152 Knoten erweitert.
Messung 2: Die Schneeflocke wird von 49152 auf 196608 Knoten erweitert.

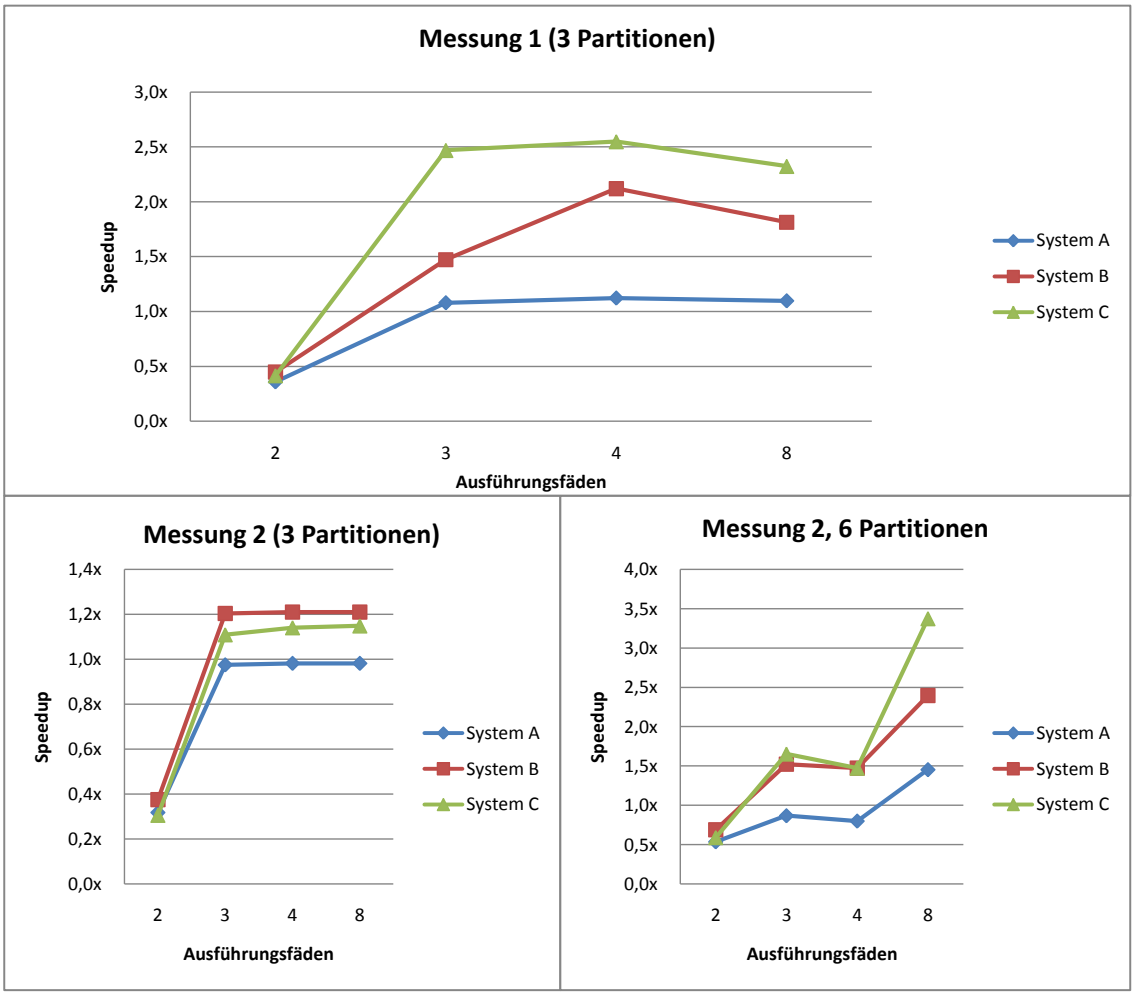

Abbildung 6.7: Messergebnisse der „Kochschen Schneeflocke".

7. Zusammenfassung & Ausblick

7.1 Zusammenfassung

Aufgrund des aktuellen Trends zu Mehrkernarchitekturen entstand bei den Entwicklern von GrGen.NET der Wunsch nach einer parallelisierten Version und die Idee zu dieser Diplomarbeit. Ziel war die Untersuchung, wie die Graphersetzung auf Mehrkernsystemen ausgeführt werden kann. Des Weiteren sollte eine erste Implementierung mit GrGen.NET durchgeführt werden. Dass sich GrGen.NET hierfür eignet, hat ein Vergleich mit anderen Graphersetzungssystemen bestätigt.

Als wesentlicher Punkt einer Parallelisierung wurde die Partitionierung des Arbeitsgraphen identifiziert. Die verfügbaren Partitionierungsalgorithmen haben sich jedoch als schwer anwendbar herausgestellt: Sie stützen sich meist auf Anwendungsdaten oder Heuristiken, die eine bestimmte Struktur des Graphen voraussetzen oder selbst im Graphen erkennen. Der Versuch, diese Algorithmen einzubinden, hätte der Arbeit einen ganz neuen Schwerpunkt gegeben. Allerdings ist es im Rahmen der Arbeit gelungen, einen Algorithmus aus der Gebietsplanung auf Graphen zu übertragen, der eine akzeptable Partitionierung eines Graphen vornimmt.

Die Partitionierung des Arbeitsgraphen stellt eine statische Aufteilung des Suchraumes dar. Allerdings wurde in dieser Arbeit gezeigt, dass auch eine dynamische Aufteilung verwendet werden kann. So können zu durchsuchende Elemente auch ohne Partitionierung auf mehrere Anwendungsfäden verteilt werden. Des Weiteren wurde vorgestellt, auf welche Weise die Suche nach einer einzigen Musterinstanz parallel durchgeführt werden kann. Es war hierbei zu sehen, dass hierbei die Größe und Form des Musters die Möglichkeit zur parallelen Suche stark beeinflusst.

Für die Evaluierung wurden Graphpartitionen und die parallele Mustersuche ohne Partitionierung umgesetzt. In Situationen, in denen der Listentrick (5.3.2.2) keinen Einfluss ausübt, konnte eine Leistungssteigerung verzeichnet werden. Mit einer GrGen.NET-Version ohne Listentrick konnte gezeigt werden, dass die in dieser Arbeit vorgestellten Parallelisierungstechniken mit der Anzahl der verwendeten Rechenkerne skalieren. Des Weiteren wurden die bisherigen theoretischen Kenntnisse der parallelen Graphersetzung um Erfahrungen der praktischen Umsetzung in ein Softwaresystem ergänzt.

Offen war zu Beginn der Arbeit, ob ein paralleles Graphersetzungssystem überhaupt benötigt wird. Obwohl die Graphersetzung bereits seit den 70er Jahren ausführlich erforscht wird, kommt sie gegenwärtig in wenigen Gebieten zur Anwendung. In diesen Fällen sind die Ausführungsgeschwindigkeiten gut genug, um keine weitere Leistungssteigerung zu erfordern. Daher war die erste Aufgabe, ein potentielles Anwendungsgebiet für die Graphersetzung zu finden, das deutlich höhere Anforderungen an die Leistungsfähigkeit der Systeme stellt.

Da Abbildungen von DNA-Molekülen eine offensichtliche Ähnlichkeit zu Darstellungen von Graphen besitzen, entstand die Idee, genetische Prozesse mittels Graphersetzung zu simulieren. Dieses Fachgebiet bietet sehr hohe Anforderungen bezüglich der Datenmengen und Vielfalt der Operationen (das menschliche Genom besteht aus mehr als 3000 Millionen Nucleotiden, keine andere gegenwärtige Anwendung stellt die Graphersetzung vor die Aufgabe, solche Datenmengen zu bewältigen). Wie Gespräche mit Biologen der Universität Karlsruhe und des Forschungszentrums Karlsruhe bestätigt haben, besteht ein vitales Interesse an neuartigen Simulationstechniken im Bereich der Molekularbiologie und Genetik. Bestätigt wurde diese Idee weiterhin durch eine Veröffentlichung (siehe [McNi01]), die bereits vor mehreren Jahren die Idee aufbrachte, genetische Prozesse durch Graphersetzung zu simulieren. In dieser Arbeit wurde daher weiterhin ein Benchmark für Graphersetzungssysteme entwickelt, der die Genexpression, einen fundamentalen Prozess der Genetik, simuliert.

7.2 Ausblick

In diesem Abschnitt werden einige Ansätze für weiterführende Arbeiten aufgezeigt.

Explizite Parallelisierung

Die Ausführungsgeschwindigkeit der in dieser Arbeit erstellten parallelen Version von GrGen.NET leidet unter dem Synchronisierungsaufwand. Dieser ist notwendig, um die identischen Ergebnisse zu erzielen, wie bei der sequentiellen Version von GrGen.NET. Zukünftige Arbeiten könnten dem Anwendungsentwickler ermöglichen, den Synchronisierungsaufwand zu beeinflussen. Ist dem Entwickler beispielsweise bekannt, dass zwei Regeln A und B potentiell konfliktieren könnten, diese Situation in den verwendeten Arbeitsgraphen jedoch nicht auftreten kann, so könnte er solche Regeln entsprechend markieren. Diese Regeln können dann ohne weitere Überprüfungen parallel verarbeitet werden.

Erzeugen von Partitionen

Der vorgestellte Rubini-Algorithmus erzeugt Partitionen, deren Qualität stark von den zufällig gewählten Ankerpunkten abhängt. In weiteren Arbeiten sollte untersucht werden, wie andere Partitionierungsalgorithmen die Ausführungsgeschwindigkeit beeinflussen können.

Einbindung weiterer Fähigkeiten von GrGen.NET

In dieser Arbeit konnten nicht alle Möglichkeiten, die GrGen.NET dem Anwender bietet, parallelisiert werden. Insbesondere, da einige Neuerungen, während diese Arbeit umgesetzt wurde, noch in Entwicklung waren ([Jaku08]). In zukünftigen Arbeiten kann untersucht werden, welche Teile von GrGen.NET weiterhin parallelisiert werden können.

Leistungssteigerung

Es konnte gezeigt werden, dass die Parallelisierung von GrGen.NET, abhängig vom Anwendungsfall, eine Steigerung der Leistung zur Folge hat. Allerdings sind noch viele Punkte offen, an denen die vorliegende parallele Version verbessert werden kann. Offensichtlich ist beispielsweise die Verwaltung von Objekten: Um den Code übersichtlich und lesbar zu halten, werden vorhandene Optimierungsmechanismen wie das Wiederverwerten von nicht mehr benötigten Objekten in der parallelen Version noch nicht genutzt. An dieser Stelle muss auch der Listen-Trick genannt werden, da er, wie in dieser Arbeit beschrieben, den Rechenaufwand um ein Vielfaches reduzieren kann. Dass die Leistung weiterhin gesteigert werden kann, darf als wahrscheinlich bezeichnet werden.

Verschachtelter Parallelismus

Obwohl die vorliegende Version von GrGen.NET an verschiedenen Stellen Möglichkeiten zum verschachtelten Parallelismus bietet, wurden diese noch nicht genutzt. Es bleibt zu untersuchen, unter welchen Bedingungen eine verschachtelte Ausführung gewinnbringend eingesetzt werden kann.

Auto-Tuning

Am IPD wird gegenwärtig ein System entwickelt, welches Parameter paralleler Anwendungen optimieren kann (*Auto-Tuner*). Mögliche Parameter bei GrGen.NET wären Anzahl und Größe von Partitionen, Partitionierungsstrategie sowie die Anzahl der verwendeten Arbeitsfäden. Es ist geplant, zu untersuchen, ob die Leistung von GrGen.NET durch Auto-Tuning weiter verbessert werden kann.

Parallelisierung durch anwendungsbasierte Partitionen

In [Taen96a] und [Hart95] werden Anwendungen der Graphersetzung vorgestellt, die eine parallele Verarbeitung auf Basis der Anwendungsdomäne ermöglichen. Zukünftige Erweiterungen von GrGen.NET und anderen Graphersetzungssystemen können diesen Ansatz aufgreifen: Die Verwendung von anwendungsbasiertem Wissen kann Formen der parallelen Abarbeitung ermöglichen, die von einer anwendungsunabhängigen Parallelisierung nicht erkannt werden. Im Genexpressions-Benchmark lassen sich beispielsweise innerhalb der verschiedenen Arbeitsschritte parallel ablaufende Prozesse erkennen: So können mehrere Ribosomen gleichzeitig verschiedene RNA-Moleküle in Aminosäure-Ketten transkribiren. Solche Zusammenhänge lassen sich nur schwer durch ein Software-System erkennen, wenn das vorhandene Kontextwissen nicht genutzt wird.

Ein Graphersetzungssystem auf einen bestimmten Anwendungsfall hin zu erweitern erscheint vom theoretischen Standpunkt unbefriedigend. Vielmehr kann in zukünftigen Arbeiten versucht werden, die Graphersetzungssysteme mit den nötigen Werkzeugen auszustatten, um anwendungsspezifische Informationen in das System zu integrieren. In GrGen.NET könnten beispielsweise die *.gm, *.grg und *.grs-Dateien um Befehle zur Verwaltung von Kontextwissen erweitert werden. Eine zentrale Rolle kann hierbei die Partitionierung des Arbeitsgraphen einnehmen: Während die Partitionierung in dieser Arbeit vor dem Anwender weitestgehend versteckt wird,

kann durch eine explizite Aufnahme von Partitions-Operationen in die Regelbeschreibungssprachen die Ausdrucksstärke des Systems deutlich erweitert werden. In der Implementierung des Genexpressions-Benchmarks für GrGen.NET sind Elemente der Anwendungsdomäne wie RNA-Moleküle oder Aminosäureketten nicht als solche klassifiziert. Das Graphersetzungssystem „weiß" nichts von deren Existenz.

Eine erweiterte Graphbeschreibungssprache könnte es ermöglichen, Teilgraphen als Einheit zu betrachten und mit einem Typ zu versehen. DNA-Ketten oder RNA-Ketten könnten somit als Partition vom System erkannt und typisiert werden. Regelsprachen könnten Ausdrücke der Form *„Wähle für jedes RNA-Molekül ein Ribosom und führe jeweils die Regel ‚Translation' aus"* ermöglichen. Die Formulierung von Regeln würde näher an die tatsächliche Anwendungsdomäne geführt, während gleichzeitig parallel unabhängige Prozesse leichter zu erkennen und umzusetzen wären. Abbildung 7.1 veranschaulicht diese Vorgehensweise. Wird die Unabhängigkeit der einzelnen RNA-Moleküle durch Partitionen und darauf arbeitenden Regeln formalisiert, kann eine Verarbeitung ohne Sperren oder weitere Tests bezüglich semantischer Abhängigkeiten stattfinden und die parallele Ausführung somit deutlich beschleunigen.

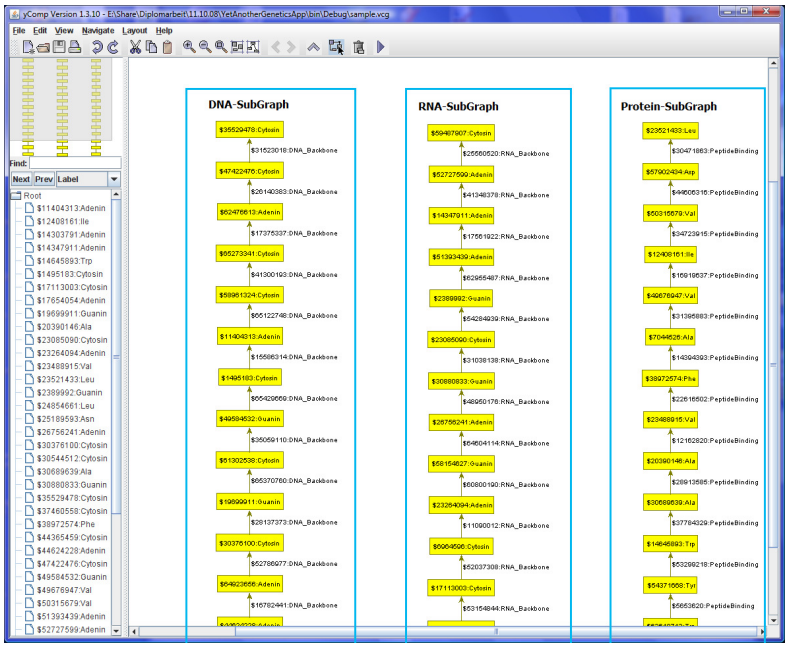

Abbildung 7.1: Die yComp-Anwendung markiert Subgraphen und zeigt deren Typen an.

Anwendungen der Graphersetzung im Bereich der Genetik

Der im Rahmen dieser Arbeit entwickelte Genexpressions-Benchmark veranschaulicht die Möglichkeit, genetische Prozesse mittels Graphersetzung zu simulieren. In weiteren Arbeiten ist zu untersuchen, für welche genetischen Prozesse Simulationswerkzeuge benötigt werden und inwiefern die Graphersetzung zu deren Umsetzung geeignet ist.

Eine mögliche Erweiterung von Graphersetzungssystemen für Genetik-Simulationen wäre die Einbettung eines Zeitgebers, so dass Ersetzungsvorgänge die gleiche Zeit

benötigen wie deren realen Vorbilder. Beispielsweise dauert die Transkription eines Gens in Natur eine gewisse Zeit, während RNA-Moleküle nach einer bestimmten Zeitspanne wieder zerfallen.

Um weitere Teilprobleme der Genetik zu simulieren, erscheint eine enge Kooperation mit Biologen und Bioinformatikern notwendig. Es ist des Weiteren als wahrscheinlich anzusehen, dass die Graphersetzung alleine nicht ausreichen wird, um ehrgeizige Projekte wie die Simulation einer kompletten Zelle umzusetzen. Beispielsweise kennt ein Graphersetzungssystem keine lokalen Zusammenhänge: Eine Polymerase in dem in dieser Arbeit entwickelten Benchmark kann Gene exprimieren, ohne vorher festzustellen, ob sie sich überhaupt nahe genug an einem Promotor aufhält, damit eine chemische Reaktion stattfinden kann. Wie eine Interaktion zwischen Graphersetzung und anderen Simulationstechniken möglich ist, muss in zukünftigen Arbeiten untersucht werden.

Literatur

[730287] *Is parallelism already concurrency? Part 1: Derivations in graph grammars*, London, UK, 1987. Springer-Verlag.

[730387] *Is parallelism already concurrency? Part 2: Non-sequential processes in graph grammars*, London, UK, 1987. Springer-Verlag.

[B. H94] R. Leland B. Hendrickson. The Chaco user's guide: Version 2.0. Technical report sand94-2692, Sandia National Laboratories, Albuquerque, NM, 1994.

[BaNi98] Franz Baader und Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA. 1998.

[Batz05] Gernot Veit Batz. Graphersetzung für eine Zwischendarstellung im Übersetzerbau. Diplomarbeit, Universität Karlsruhe, IPD Goos, 2005.

[BHKK] Jonathan W. Berry, Bruce Hendrickson, Simon Kahan und Petr Konecny. Software and Algorithms for Graph Queries on Multithreaded Architectures.

[BlGe07] Jakob Blomer und Rubino Geiß. The GrGen.NET User Manual. Technischer Bericht 2007-5, Universität Karlsruhe, IPD Goos, July 2007. ISSN 1432-7864.

[CELM+96] Andrea Corradini, Hartmut Ehrig, Michael Löwe, Ugo Montanari und Francesca Rossi. An Event Structure Semantics for Graph Grammars with Parallel Productions. 1996, S. 240–256.

[CHMP+02] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza und Dániel Varró. VIATRA: Visual automated transformations for formal verification and validation of UML models. In *In Proceedings 17th IEEE International Conference on Automated Software Engineering (ASE 2002.* IEEE Computer Society, 2002, S. 267–270.

[DBLP83] Graph-Grammars and Their Application to Computer Science, 2nd International Workshop, Haus Ohrbeck [near Osnabrück], Germany, October 4-8, 1982. In Hartmut Ehrig, Manfred Nagl und Grzegorz Rozenberg (Hrsg.), *Graph-Grammars and Their Application to Computer Science*, Band 153 der *Lecture Notes in Computer Science.* Springer, 1983.

[ecoc] The EcoCyc Database, http://ecocyc.org/.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange und G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA. 2006.

[EhLö93] Hartmut Ehrig und Michael Löwe. Parallel and Distributed Derivations in the Single-Pushout Approach. *Theor. Comput. Sci.*, 109(1&2), 1993, S. 123–143.

[Ehri83] Hartmut Ehrig. Aspects of concurrency in graph grammars. 1983, S. 58–81.

[Ehri88] Hartmut Ehrig. Distributed Parallelism of Graph Transformations. 1988, S. 1–19.

[fast] Das FASTA-Format, http://www.ncbi.nlm.nih.gov/blast/fasta.shtml.

[GaJS74] M. R. Garey, D. S. Johnson und L. Stockmeyer. Some simplified NP-complete problems. In *STOC '74: Proceedings of the sixth annual ACM symposium on Theory of computing*, New York, NY, USA, 1974. ACM, S. 47–63.

[gfk] GfK Geomarketing Glossar, http://www.gfk-geomarketing.de.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 1995.

[Goos00] Gerhard Goos. *Vorlesungen über Informatik Band 1: Grundlagen und funktionales Programmieren*. Springer. Jan 2000.

[GrG] GrGen - Graph Rewrite GENerator. Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe.

[GTBB+08] Rubino Geiß, Gabriele Taentzer, Enrico Biermann, Dénes Bisztray, Bernd Bohnet, Iovka Boneva, Artur Boronat, Leif Geiger, Ákos Horvath, Ole Kniemeyer, Tom Mens und Benjamin Ness et al. Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools. In A. Schürr, M. Nagl und A. Zündorf (Hrsg.), *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07)*, Band NN der *LNCS*. Springer, 2008. http://www.springerlink.com/content/105633/.

[Hart] P. Hartmann. Parallel Graph Replacement Systems for Modelling Biological Cellular Structures.

[Hart95] Peter Hartmann. Parallel and Distributed Processing of Cellular Hypergraphs. In *PaCT '95: Proceedings of the 3rd International Conference on Parallel Computing Technologies*, London, UK, 1995. Springer-Verlag, S. 57–69.

[HeLe95] Bruce Hendrickson und Robert Leland. A multilevel algorithm for partitioning graphs. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, New York, NY, USA, 1995. ACM, S. 28.

[Jaku08] Edgar Jakumeit. Mit GrGen.NET zu den Sternen – Erweiterung der Regelsprache eines Graphersetzungswerkzeugs um rekursive Regeln mittels Sterngraphgrammatiken und Paargraphgrammatiken. Diplomarbeit, jul 2008.

[KeLi70] B. W. Kernighan und S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell system technical journal*, 49(1), 1970, S. 291–307.

[Knip06] Rolf Knippers. *Molekulare Genetik*. Georg Thieme Verlag. 2006.

[Kreo81] Hans-Jörg Kreowski. A Comparison Between Petri-Nets and Graph Grammars. In *WG '80: Proceedings of the International Workshop on Graphtheoretic Concepts in Computer Science*, London, UK, 1981. Springer-Verlag, S. 306–317.

[LöEh91] Michael Löwe und Hartmut Ehrig. Algebraic Approach to Graph Transformation Based on Single Pushout Derivations. In *WG '90: Proceedings of the 16rd International Workshop on Graph-Theoretic Concepts in Computer Science*, London, UK, 1991. Springer-Verlag, S. 338–353.

[Matt04] Timothy G. Mattson. *Patterns for Parallel Programming*. Addison-Wesley Longman, Amsterdam. 2004.

[McNi01] John S. McCaskill und Ulrich Niemann. Graph Replacement Chemistry for DNA Processing. 2001, S. 103–116.

[MüGe07] Jens Müller und Rubino Geiß. Speeding up Graph Transformation through Automatic Concatenation of Rewrite Rules. Technischer Bericht, 2007.

[Nagl77] Manfred Nagl. On the Relation Between Graph Grammars and Graph L-Systems. In *FCT*, 1977.

[Parr] Terence Parr. An Introduction to ANTLR.

[PSJT08] Victor Pankratius, Christoph Schaefer, Ali Jannesari und Walter F. Tichy. Software engineering for multicore systems: an experience report. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, New York, NY, USA, 2008. ACM, S. 53–60.

[Reis81] Wolfgang Reisig. A Graph Grammar Representation of Non-Sequential Processes. 1981, S. 318–325.

[SCEH75] H J SCHNEIDER und 297-316 EHRIG, H Acta lnf 6 (1976). Grammars on partial graphs. *Acta Informatica*, Band Volume 6, Number 3 / September, 1976, 28 February 1975, S. 297–316.

[Sedg88] Robert Sedgewick. *Algorithms, Second Edition.* Addison-Wesley Publishing Company, Inc. 1988.

[Taen96a] Gabriele Taentzer. Hierarchically Distributed Graph Transformation, 1996.

[Taen96b] Gabriele Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems.* Dissertation, Technical University of Berlin, 1996.

[Taen96c] Gabriele Taentzer. Towards synchronous and asynchronous graph transformations. *Fundam. Inf.*, 26(3-4), 1996, S. 387–406.

[Taen99] Gabriele Taentzer. AGG: A Tool Environment for Algebraic Graph Transformation. In *AGTIVE*, 1999, S. 481–488.

[Tane01] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA. 2001.

[TaYZ89] Peiyi Tang, Pen-Chung Yew und Chuan-Qi Zhu. A parallel linked list for shared-memory multiprocessors. Technischer Bericht CSRD 879, September 1989.

[ToMa87] Tommaso Toffoli und Norman Margolus. *Cellular automata machines: a new environment for modeling.* MIT Press, Cambridge, MA, USA. 1987.

[VaSV05] Gergely Varro, Andy Schurr und Daniel Varro. Benchmarking for Graph Transformation. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, Washington, DC, USA, 2005. IEEE Computer Society, S. 79–88.

[Wank87] Frank Wankmüller. Application of Graph Grammars in Music Composing Systems. 1987, S. 580–592.