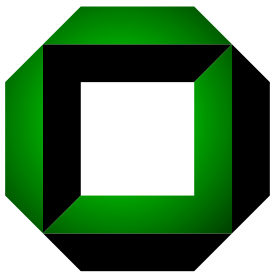


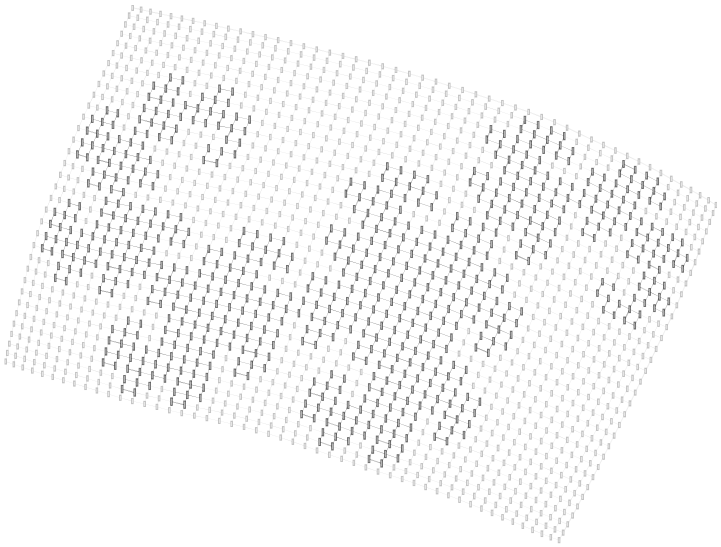
Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825

Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation
Lehrstuhl Prof. Goos

GrGen.NET: Portierung und Erweiterung des Graphersetzungssystems GrGen

Studienarbeit von Moritz A. Kroll

Überarbeitete Fassung vom 29. Oktober 2007



Betreuer:
Dipl.-Inform. Rubino Geiß
Dipl.-Inform. Tom Gelhausen

Verantwortlicher Betreuer:
Prof. em. Dr. Dr. h.c. Gerhard Goos

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt zu haben.

Ort, Datum Unterschrift

Kurzfassung

GrGen.NET ist ein Graphersetzungssystem auf Basis des *single-pushout*-Ansatzes, das, um Muster möglichst schnell zu finden, seine Suchstrategien zur Laufzeit an den Arbeitsgraph und die einzelnen Graphersetzungsregeln anpassen kann. Mit dem schrittweisen Ausführen von Regeln und der visuellen Darstellung der Änderungen mittels des Graphanzeigesystems yComp bietet GrGen.NET eine große Hilfe bei der Entwicklung von komplexen Regeln und bei der Suche nach Fehlern. Wie Messungen zeigen, wurde der Ressourcenverbrauch (Rechenzeit und Speicher) im Vergleich zum Vorgänger GrGen – für größere Probleme – signifikant verringert.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Grundaufgabe eines Graphersetzungssystems ist es, Muster in einem Graphen (im Folgenden Arbeitsgraph genannt) zu finden und durch andere Muster zu ersetzen. Dabei werden die Muster über Knoten und Kanten und evtl. über weitere Eigenschaften wie Typ- oder Attribut-Bedingungen definiert. Da das Suchen von Mustern in Graphen ein NP-vollständiges Problem ist, ist ein naives Herangehen für die Praxis vollkommen unbrauchbar. Mit Heuristiken versucht man daher, besondere Eigenschaften der jeweiligen Muster auszunutzen, um möglichst viele Probleminstanzen in $\mathcal{P}$ zu lösen.

Das zur Zeit schnellste, bekannte Graphersetzungssystem ist GrGen mit Suchplan-Erzeugung [GBG+06], das am IPD Goos (Institut für Programm- und Datenstrukturen) der Universität Karlsruhe(TH) entwickelt wurde und auf das im nächsten Kapitel näher eingegangen wird.

In dieser Studienarbeit wird ein an GrGen angelehntes Graphersetzungssystem mit folgenden Zielen entwickelt:

- Ermöglichung einer größeren Verbreitung von GrGen durch die Portierung auf die objektorientierte .NET-Umgebung, ohne jedoch die Leistung wesentlich zu verschlechtern.
- Weiterentwicklung der Suchplanerzeugung.
- Erweiterung der regulären Graphersetzungssequenzen.
- Unterstützung der Fehlersuche bei der Entwicklung von Graphersetzungsregeln.
- Erstellung eines Programms zum Ausmessen der Ausführungszeiten aller möglichen Suchpläne zu gegebenen Regeln, um die hier verwendete Heuristik aus [Bat06] bewerten zu können.

Kapitel 2

Verwandte Arbeiten

Diese Studienarbeit bezieht sich hauptsächlich auf das Graphersetzungssystem GrGen. Die dazugehörige Arbeit [GBG+06] bietet auch einen kurzen Überblick über andere Systeme.

2.1 GrGen

GrGen ist ein in C und Java geschriebenes Graphersetzungssystem mit dem in Abbildung 2.1 gezeigten Aufbau: Die in GrGen enthaltene Graphbibliothek „libGr“ bietet Anwendungen Funktionen an, mit denen Graphen erzeugt, manipuliert, dargestellt, gelöscht und ersetzt werden können. Die interne Darstellung und Verwaltung der Graphen und Graphersetzungsregeln wird von sogenannten „Backends“ implementiert. Neben dem nicht-persistenten „LGSP-Backend“, das auf Suchplänen basiert und das bisher schnellste Backend ist, gibt es zum Beispiel noch das ältere, persistente und deutlich langsamere „PSQL-Backend“, das auf einer PostgreSQL-Datenbank aufbaut.

Die vom Benutzer erstellten Modell- und Regel-Spezifikationen (GM- bzw. GRG-Dateien) werden von dem GrGen-Generator zu backend-spezifischen C-Quelldateien verarbeitet. Diese werden dann zusammen mit Teilen des Quellcodes des ausgewählten Backends zu einer Modell-Bibliothek (model) und einer Regel-Bibliothek (actions) kompiliert, die die libGr verwenden kann.

Eine mitgelieferte Anwendung ist die GrShell. Sie ermöglicht mittels einer simplen Skriptsprache die einfache Nutzung der Graphersetzung, insbesondere durch die Eingabe der von der libGr unterstützten „Graphersetzungssequenzen“. Mit diesen Graphersetzungssequenzen kann man beschreiben, wie eine Menge von Graphersetzungsregeln angewendet werden soll. Beschreibt $\mathcal{P}$ die Menge der Ersetzungsregeln, $\mathcal{R}$ die Menge der Graphersetzungssequenzen und sind $p \in \mathcal{P}$, $R, R_1, R_2 \in \mathcal{R}$ und $n \in \mathbb{N}$, so wird die Bedeutung einer solchen Sequenz wie folgt definiert:

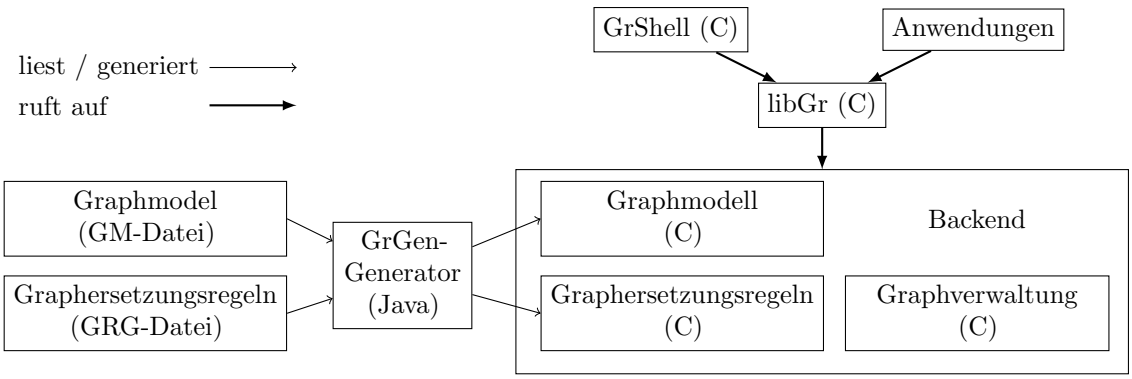


Abbildung 2.1: Aufbau des GrGen-Systems

p	Die Regel p wird angewendet. p schlägt fehl, wenn das Muster nicht gefunden wird.
$[p]$	Die Regel p wird auf alle Vorkommen des entsprechenden Musters quasi „gleichzeitig“ angewendet. Der Benutzer muss dabei sicherstellen, dass eine Ersetzung einer Fundstelle keine andere Fundstelle so beeinflussen kann, dass diese ungültig wird. Ansonsten ist das Verhalten undefiniert. $[p]$ schlägt fehl, wenn das Muster nicht gefunden wird.
$R_1\ R_2$	Zuerst wird R_1 und dann R_2 ausgeführt, selbst wenn R_1 fehlschlägt. $R_1\ R_2$ schlägt fehl, wenn sowohl R_1 als auch R_2 fehlschlagen.
(R)	Klammerung zur Festlegung der Auswertungsreihenfolge.
$R*$	Die Sequenz R wird so oft angewendet, bis sie fehlschlägt. $R*$ schlägt fehl, wenn R kein mal erfolgreich war.
$R\{n\}$	Die Sequenz R wird so oft angewendet, bis sie fehlschlägt, aber maximal n-mal. $R\{n\}$ schlägt fehl, wenn R kein mal erfolgreich war.

Zusammen mit diesen Graphersetzungsequenzen ist die Graphersetzung von GrGen turingmächtig.

2.2 Die libGr von GrGen

Bedingt durch die Programmiersprache C konnte die libGr objektorientiertes Programmieren nur teilweise in einigen „Interfaces“ mittels Funktionszeigern in Verbunden verwirklichen. Dort werden als einzige Schnittstellen „Backend“, „Graph“ und „Actions“ angeboten, über die fast die gesamte Funktionalität angeboten wird:

- Ein „Backend“-Objekt bietet Funktionen zum Erstellen und Laden von Graphen an.
- Ein „Graph“-Objekt stellt alle Funktionen bezüglich des Graphmodells und dem Erstellen, Manipulieren und Löschen von Knoten und Kanten zur Verfügung.
- Mit einem „Actions“-Objekt können die von diesem Objekt angebotenen Regeln abgefragt und auf einen Graphen angewendet werden.

Weiterhin kann die libGr Graphen in verschiedenen Formaten ausgeben.

2.3 Das LGSP-Backend von GrGen

Die Idee, die im LGSP-Backend verwirklicht wurde, ist es, anhand von statistischen Informationen über die Struktur des Arbeitsgraphen und statischen Informationen über das Suchmuster eine heuristisch „gute“ Reihenfolge (Suchplan) zu berechnen, in der die einzelnen Elemente des Musters im Arbeitsgraphen gesucht werden sollen. „Gut“ ist im Hinblick darauf zu verstehen, dass sich die Kandidaten für die Vorkommen des Musters während der Suche nicht unnötig stark aufspalten, da das sonst zu einem exponentiellen Wachstum der Anzahl der Kandidaten führen kann und somit zu exponentieller Laufzeit.

Ein Suchplan beschreibt eine Reihenfolge von rekursiv ausgeführten Operationen, mit denen man sukzessive den Arbeitsgraphen nach dem entsprechenden Muster absucht. Schlägt eine Operation fehl, weil kein Element im Graphen die Bedingungen für die Operation erfüllt, so wird das gefundene Element der vorherigen Operation verworfen und nach einem nächsten passenden Element gesucht („Backtracking“). Wenn die erste Operation kein passendes Element mehr findet, existiert das Muster nicht im Arbeitsgraphen. Die möglichen Suchoperationen sind:

- Lookup: Sucht einen Knoten eines gegebenen Typs.
- Extend: Sucht von einem gegebenen Knoten aus eine Kante und den entsprechenden adjazenten Knoten jeweils gegebenen Typs.
- Check: Sucht zwischen zwei gegebenen Knoten eine Kante eines gegebenen Typs.

- CheckNegative: Verwirft den aktuellen Kandidaten, falls ein „negatives Muster“ auf die bisher gefundenen Elemente passt. (Wird nicht im Planungsgraphen verwendet)
- CheckCondition: Verwirft den aktuellen Kandidaten, falls eine Bedingung, die sich auf Attribute und/oder Typen der bisher gefundenen Elemente bezieht, nicht erfüllt ist. (Wird nicht im Planungsgraphen verwendet)

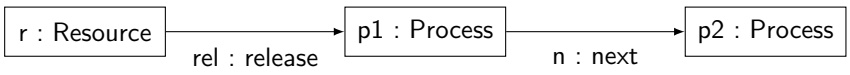
Da zu Anfang noch keine sinnvollen statistischen Informationen über den Arbeitsgraphen existieren, werden von GrGen initiale Suchpläne („statische Suchpläne“) erzeugt, damit auch dann Graphersetzungen durchgeführt werden können. Diese Suchpläne erfüllen jedoch nur die Bedingung, dass es für jede Zusammenhangskomponente im Suchmuster nur eine Lookup-Operation gibt, und berücksichtigen eventuell in den Regeln angegebene Präferenzen für die Reihenfolge des Suchens. Die Bedingung sorgt dafür, dass zum Beispiel nicht erst beliebige (passende) Knoten im Arbeitsgraphen gewählt werden und dann versucht wird die Kanten zwischen ihnen zu finden.

Um für ein Muster wie in Abbildung 2.2(a) einen „guten“ Suchplan zu erstellen, erzeugt man zuerst einen Planungsgraphen (siehe Abbildung 2.2(b)), der alle Pfade darstellt, auf denen man von „nichts“ („Root“-Knoten) ausgehend alle Elemente finden kann, und wiedergibt, welche Operationen dazu in welcher Reihenfolge (entlang des Pfades) notwendig sind. Wie in [Bat06] ausführlich beschrieben, werden dabei jeder Operation Kosten zugeordnet, die wiederspiegeln, wie stark sich der aktuelle Kandidat bei Ausführung dieser Operation voraussichtlich aufspalten würde.

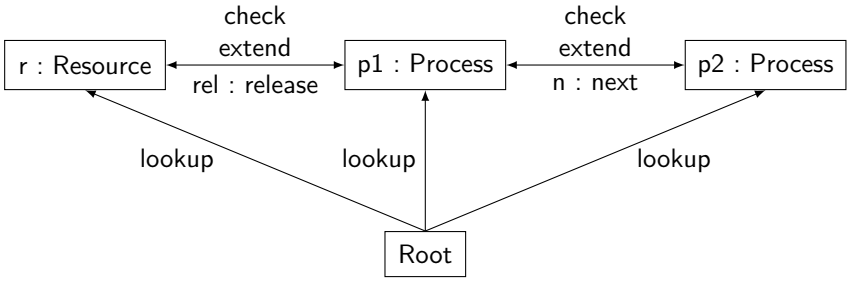
Mittels eines minimalen aufspannenden Arboreszenten werden dann die Operationen ausgewählt, die zum Einen notwendig sind, um das Muster vollständig zu finden, und zum Anderen mit Hilfe der Kosten die postulierte Aufspaltung der Kandidaten minimiert (siehe Abbildung 2.2(c)). Da jedoch auch die Reihenfolge der Operationen die Expansion der Kandidaten beeinflusst, wird in einem letzten Schritt der Suchplan-Erzeugung die Reihenfolge unter Berücksichtigung der Abhängigkeiten zwischen den Operationen so gewählt, dass Operationen mit niedrigen Kosten möglichst früh und solche mit hohen Kosten möglichst spät ausgeführt werden (in Abbildung 2.2(c) gibt es aufgrund der Abhängigkeiten nur eine mögliche Reihenfolge).

Für die Muster der negativen Anwendungsbedingungen werden nur die statischen Suchpläne verwendet, welche im Suchplan auch nur grundsätzlich als letzte Operationen eingeplant werden. Das Einplanen der Bedingungen wurde leider nicht implementiert, so dass es bei der Verwendung von Regeln mit Bedingungen nach der dynamischen Generierung zu falschen Fundstellen kommt.

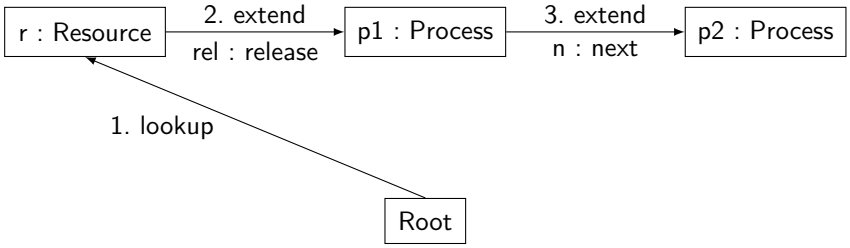
Der Suchplan wird dann in Form eines „Suchprogramms“ (Matcher-Programm) im Speicher abgelegt, das die Reihenfolge und Art der Suchoperationen festlegt, und während des Suchens von einem Interpretierer ausgeführt wird.



(a) Das Suchmuster



(b) Der Planungsgraph



(c) Der nach der Heuristik berechnete Suchplan

Abbildung 2.2: Die „giveRule“ des Mutex-Benchmarks

Kapitel 3

GrGen.NET

GrGen.NET ist das im Rahmen dieser Studienarbeit implementierte, an GrGen angelehnte Graphersetzungssystem. Während dieses Kapitel auf den Aufbau und die Schnittstellen des Systems eingeht, wird sich das nächste Kapitel mit Implementierungsdetails der GrGen.NET-Version des LGSP-Backends beschäftigen, welches das zur Zeit einzige Backend für GrGen.NET ist.

3.1 Grundaufbau

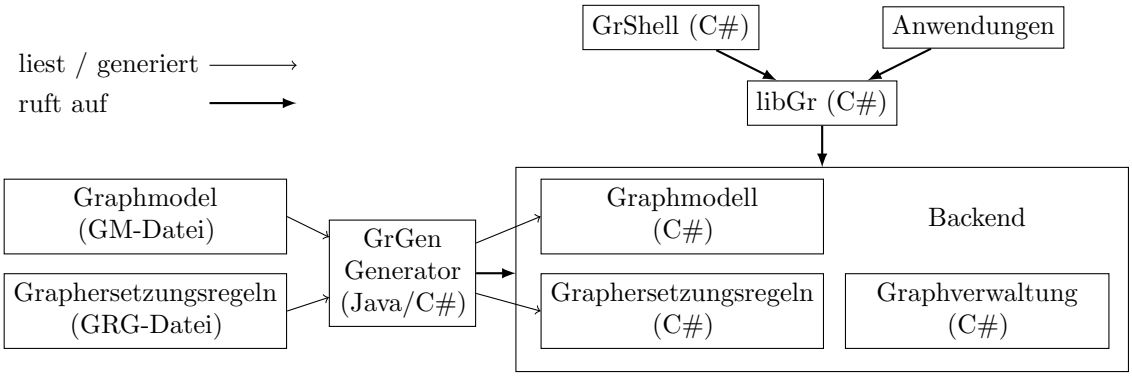


Abbildung 3.1: Aufbau des GrGen.NET-Systems

Wie Abbildung 3.1 zeigt, ist der Aufbau von GrGen.NET sehr ähnlich zu dem von GrGen (vergleiche Abschnitt 2.1). Neben dem Unterschied, dass alle C-Komponenten hier in C# geschrieben sind, hat der „GrGen Generator“ noch einen C#-Anteil erhalten. Dieser übernimmt zum Einen die Generierung der statischen Suchpläne für das LGSP-Backend und erstellt zum Anderen .NET-Graphmodell- und -Graphersetzungsregel-Bibliotheken, die durch die libGr verwendet werden. Die Generierung der statischen Suchpläne wurde in ein C#-Programm verlagert, da die Routinen des LGSP-Backends zur Erzeugung der dynamischen Suchpläne wieder verwendet werden konnten, so dass diese Funktionalität nicht noch einmal in Java implementiert werden musste.

3.2 Die libGr

Im Vergleich zur GrGen-Version der libGr bietet die GrGen.NET-Version, wie Abbildung 3.2 deutlich mehr Objekte bzw. Schnittstellen an, die in den folgenden Abschnitten beschrieben werden.

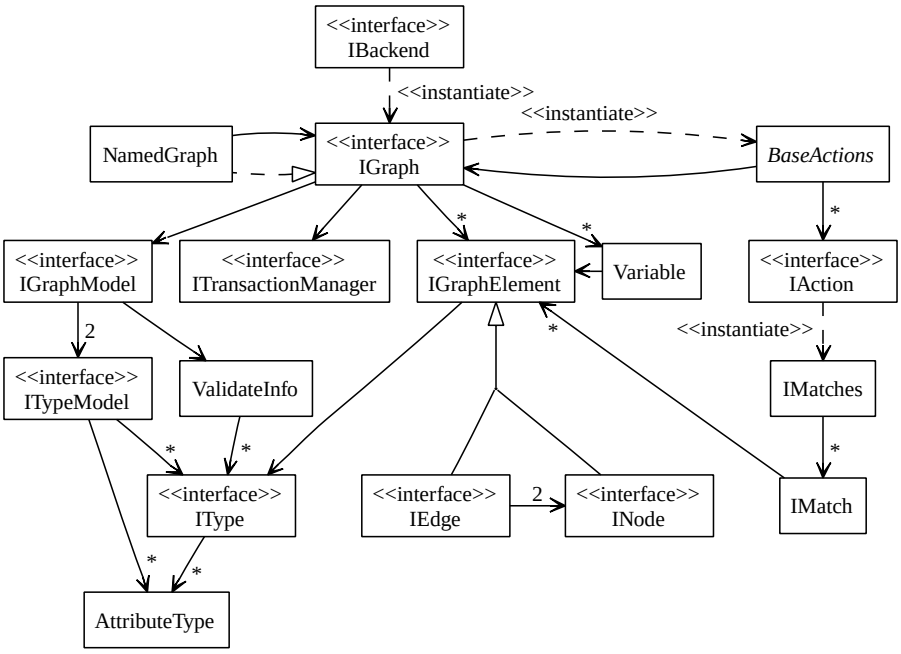


Abbildung 3.2: Die Klassenstruktur der libGr

3.2.1 Die Backend-Schnittstelle IBackend

Mit der `IBackend`-Schnittstelle können neue Graph-Instanzen erzeugt und, sofern es das Backend erlaubt, geöffnet werden, die die `IGraph`-Schnittstelle implementieren. Dabei muss eine Graphmodell-Spezifikation in Form einer `IGraphModel`-Instanz übergeben werden (siehe Abschnitt 3.2.3).

3.2.2 Die Schnittstelle IGraph

Eine `IGraph`-Instanz repräsentiert einen (nicht notwendigerweise zusammenhängenden) Graphen und verwaltet zusätzlich eine (abgesehen von Speichereinschränkungen) unbeschränkte Menge von Variablen, die auf `IGraphElement`-Instanzen (siehe Abschnitt 3.2.6) verweisen. Über die Schnittstelle können alle Graph-Verwaltungsaufgaben abgewickelt werden:

- Hinzufügen und Entfernen von Knoten und Kanten (*AddNode, AddEdge, Remove, RemoveEdges*)
- Umtypisieren von Knoten und Kanten. Hierbei werden typgemeinsame Attribute beibehalten und alle anderen auf ihren Standardwert gesetzt (*SetNodeType, SetEdgeType*)
- Abfragen von eindeutigen Elementnamen zur Identifizierung der Elemente in externen Umgebungen (*GetElementName*)
- Setzen und Abfragen des Wertes einer Variablen (*SetVariableValue, GetVariableValue*)
- Abfragen aller Variablen, die auf ein Element zeigen (*GetElementVariables*)
- Überprüfen des Graphen auf Gültigkeit anhand der Graphmodell-Spezifikation (*Validate*)
- Ausgabe des Graphen oder eines Vorkommens eines Musters (*Dump, DumpMatch*)
- Erstellen einer Kopie des Graphen (*Clone*)
- Zugriff auf die zugehörige `ITransactionManager`-Instanz (*TransactionManager*-Eigenschaft)

- Zugriff auf die `IGraphModel`-Instanz (*Model*-Eigenschaft)
- Laden einer `BaseActions`-Instanz (*LoadActions*)
- Generischer Zugriff auf backend-spezifische Funktionen (*Custom*)

Zudem bietet sie einige Ereignisse an, die während Transaktionen zur Speicherung der Änderungen auch von der libGr selbst verwendet werden:

- *OnNodeAdded*/ *OnEdgeAdded*: Wird ausgelöst, nachdem ein Knoten bzw. eine Kante hinzugefügt wurde.
- *OnRemovingNode*/ *OnRemovingEdge*/ *OnRemovingEdges*: Wird ausgelöst, bevor ein Knoten bzw. eine Kante bzw. alle Kanten eines Knotens gelöscht werden.
- *OnChangingNodeAttribute*/ *OnChangingEdgeAttribute*: Wird ausgelöst, bevor ein Attribut eines Knotens bzw. einer Kante geändert wird.
  Achtung: Da die Graphelemente ihren Graphen aus Speicherplatzgründen nicht kennen, muss das Ereignis manuell mittels der `IGraph`-Funktion *ChangingNodeAttribute* bzw. *ChangingEdgeAttribute* ausgelöst werden, wenn ein Attribut eines Graphelements über *SetAttribute* von `IGraphElement` (siehe Abschnitt 3.2.6) oder, falls möglich, direkt gesetzt wird.
- *OnSettingNodeType*/ *OnSettingEdgeType*: Wird ausgelöst, bevor der Typ eines Knotens bzw. einer Kante geändert wird.

Variablen werden über `Variable`-Instanzen dargestellen, die über einen Namen (*Name*-Eigenschaft) und eine Referenz auf eine `IGraphElement`-Instanz (*Element*-Eigenschaft) verfügen. Variablen die auf kein Element verweisen, geben als „Element“ einen Nullzeiger zurück.

3.2.3 Die Schnittstelle IGraphModel

`IGraphModel` beschreibt die Graphmodell-Spezifikation, indem sie das Typmodell der Knoten und der Kanten (*NodeModel* und *EdgeModel*-Eigenschaft) als `ITypeModel`-Objekte (siehe Abschnitt 3.2.4) und Informationen über erlaubte Verbindungen zwischen Knoten in Form von `ValidateInfo`-Objekten (*ValidateInfo*-Eigenschaft) anbietet.

`ValidateInfo`-Objekte beschreiben Zusicherungen darüber, wie viele Kanten eines Typs von bzw. zu einem Knoten eines Typs existieren dürfen, wobei auch der Typ des adjazenten Knoten angegeben wird. Diese Zusicherungen werden allerdings nicht forciert, sondern nur von der *Validate*-Funktion der `IGraph`-Schnittstelle verwendet, um die Gültigkeit des Graphen zu überprüfen. „Illegale“ Graphen können also durchaus erzeugt werden.

3.2.4 Die Schnittstelle ITypeModel

Eine `ITypeModel`-Instanz enthält eine Menge von Knoten- bzw. Kantentypen (`IType`-Instanzen, siehe Abschnitt 3.2.5), auf die sowohl über ihre Identifikationsnummer (*Types*-Eigenschaft) als auch über ihren Namen (*GetType*) zugegriffen werden kann. Ob es sich bei der Instanz um ein Knoten- oder ein Kanten-Typmodel handelt, kann über die *IsNodeModel*-Eigenschaft abgefragt werden. Die *RootType*-Eigenschaft gibt den Wurzeltypen der Typhierarchie an.

3.2.5 Die Schnittstelle IType

Knoten- und Kantentypen werden in der libGr als `IType`-Instanzen repräsentiert. Mittels ihrer Schnittstelle kann man herausfinden,

- wie der Typ heißt (*Name*-Eigenschaft),
- welche Identifikationsnummer der Typ hat (*TypeID*-Eigenschaft),

- ob der Typ ein Knotentyp ist (*IsNodeType*-Eigenschaft),
- ob der Typ typkompatibel zu einem anderen Typen ist (*IsA*),
- welche Unter- und Obertypen der Typ hat (*SubTypes*-, *SubOrSameTypes*-, *SuperTypes*- und *SuperOrSameTypes*-Eigenschaft),
- wie viele Attribute der Typ hat (*NumAttributes*-Eigenschaft),
- welche Attribute der Typ hat (*AttributeTypes*-Eigenschaft) und
- welches Attribut einen gegebenen Namen hat (*GetAttributeType*).

Die Attribute werden mit `AttributeType`-Instanzen beschrieben, die den Namen (*Name*-Eigenschaft), den einschließenden Typen (*OwnerType*-Eigenschaft) und den Typen des Attributs (*Kind*-Eigenschaft) in Form eines `AttributeKind`-Enums darbietet. `AttributeKind` bietet dabei diese Konstanten an:

- *IntegerAttr*: 32-bit Ganzzahl
- *BooleanAttr*: 2-wertiger Wahrheitswert (true, false)
- *StringAttr*: Beliebige Zeichenkette
- *EnumAttr*: Ein Wert eines durch die *EnumType*-Eigenschaft beschriebenen Aufzählungstypen

3.2.6 Die Schnittstellen IGraphElement, INode und IEdge

Die libGr stellt Knoten und Kanten als `INode`- bzw. `IEdge`-Instanzen dar, die beide von der `IGraphElement`-Schnittstelle erben. Über diese Schnittstelle kann man

- den Knoten- bzw. Kantentypen abfragen (*Type*-Eigenschaft),
- herausfinden, ob ein Element eine Instanz eines Typen ist (*InstanceOf*),
- den Wert eines Attributs abfragen und setzen (*GetAttribute* bzw. *SetAttribute*).

Die `INode`-Schnittstelle bietet daneben noch alle ein- bzw. ausgehenden Kanten eines gegebenen Typs (*GetIncoming* bzw. *GetOutgoing*) und die `IEdge`-Schnittstelle Quelle und Ziel der jeweiligen Kante an (*Source*- bzw. *Target*-Eigenschaft).

Das LGSP-Backend bietet hierüber hinaus auch einen direkten Zugriff auf die Attribute. Dazu muss das *attributes*-Attribut von `LGSPNode` bzw. `LGSPEdge` in einen modellspezifischen Attributtypen umgewandelt werden, der dann spezielle Eigenschaften für die einzelnen Attribute anbietet.

3.2.7 Die Schnittstelle ITransactionManager

Mit der einer `IGraph`-Instanz zugehörigen `ITransactionManager`-Instanz können Transaktionen

- begonnen (*StartTransaction*),
- abgeschlossen (*Commit*) und
- abgebrochen werden (*Rollback*).

Wird eine Transaktion abgebrochen, so werden alle Änderungen am Graphen seit Beginn der Transaktion rückgängig gemacht; eingeschlossene Transaktionen werden verworfen. Wie in Abschnitt 3.2.2 erklärt, müssen dem entsprechenden `IGraph`-Objekt (und damit der `ITransactionManager`-Instanz) Attributänderungen jedoch explizit mittels *ChangingNodeAttribute* bzw. *ChangingEdgeAttribute* mitgeteilt werden, damit auch diese Änderungen bei einem Abbruch automatisch zurückgenommen werden können.

3.2.8 Die Klasse NamedGraph

`NamedGraph` ist eine von `IGraph` abgeleitete Hüllenklasse, die `IGraph` um (eindeutige) Namen für Graphelemente erweitert. Mit `GetGraphElement` kann auf Graphelemente anhand ihres Namens zugegriffen werden.

3.2.9 Die Schnittstelle IAction

`IAction`-Instanzen sind die Repräsentationen der Regeln und verfügen über:

- einen Namen (*Name*-Eigenschaft)
- eine (potentiell leere) Liste von Eingabetypen (*Inputs*-Eigenschaft)
- eine (potentiell leere) Liste von Ausgabetypen (*Outputs*-Eigenschaft)
- eine Funktion zum Suchen des durch die Regel definierten Musters in einem gegebenen Graphen (*Match*)
- eine Funktion zum Anwenden der durch die Regel definierten Ersetzung an einer gegebenen Fundstelle. Im Falle einer „test“-Regel wird der Graph nicht verändert (*Modify*)

Die Menge der Fundstellen wird dabei als `IMatches`-Objekt zurückgegeben, die neben einer Referenz auf die erzeugende `IAction`-Instanz (*Producer*-Eigenschaft) eine Liste von `IMatch`-Objekten (*Matches*-Eigenschaft) enthält. Ein `IMatch`-Objekt besteht aus einer Kanten- und einer Knoten-Liste (*Nodes*- bzw. *Edges*-Eigenschaft), die eine einzelne Fundstelle beschreiben. Das `IMatches`-Objekt und die enthaltenen `IMatch`-Objekte sind nur bis zum Aufruf von *Modify* oder dem nächsten Aufruf von *Match* gültig.

3.2.10 Die abstrakte Klasse BaseActions

Klassen die `BaseActions` implementieren, stellen einen Satz von Regeln dar und dienen dazu Graphersetzungsregeln und Graphersetzungssequenzen anzuwenden. `BaseActions` sieht dabei folgende Funktionalität vor:

- Abfrage des Namens für den Regelsatz (*Name*-Eigenschaft)
- Abfrage des assoziierten Graphen (*Graph*-Eigenschaft)
- Abfrage von Regeln (*Actions*-Iterator und *GetAction*)
- Iteratives Anwenden einer Regel (*ApplyGraphRewrite*), wobei über einen Parameter gesteuert werden kann, wie oft die Regel angewendet wird:
  - *ActionMode.Zero*: Bis die Regel nicht mehr anwendbar ist
  - *ActionMode.Fix*: Bis sich die Anzahl der Fundstellen nicht mehr ändert
  - *ActionMode.Max*: Immer

  Über einen weiteren Parameter wird zusätzlich eine maximale Anzahl von Iterationsschritten angegeben.
- Anwenden einer Graphersetzungssequenz (*ApplyGraphRewriteSequence*)
- Festlegen und Bestimmen der maximalen Anzahl zu findender Fundstellen (*MaxMatches*-Eigenschaft)
- Setzen und Abfragen einer Ausgabeobjekt-Fabrik, die für markierte Regeln während einer Graphersetzungssequenz (*ApplyGraphRewriteSequence*) bzw. optional während des Anwendens einer Regel (*ApplyGraphRewrite*) benutzt wird, um die Fundstellen auszugeben (*DumperFactory*-Eigenschaft).

Außerdem werden einige Ereignisse zur Manipulation der Fundstellen und zum Anschluss von Hilfsmitteln zur Fehlersuche angeboten:

- *OnMatched*: Wird ausgelöst, direkt nachdem ein Muster gesucht wurde. Es muss kein Vorkommen gefunden worden sein, um das Ereignis auszulösen.
- *OnFinishing*: Wird ausgelöst, nachdem ein Muster entdeckt wurde und vor der Ersetzung.
- *OnFinished*: Wird nach der Ersetzung ausgelöst.
- *OnEnteringSequence*: Wird am Anfang einer (Teil-)Sequenz ausgelöst.
- *OnExitingSequence*: Wird am Ende einer (Teil-)Sequenz ausgelöst.

Die Graphersetzungssequenzen werden als Baum aus `Sequence`-Instanzen dargestellt, der entweder manuell über statische Fabrikfunktionen von `Sequence` oder über den `SequenceParser` erzeugt werden kann. Der `SequenceParser` übersetzt einen GRS-String mit Hilfe einer `BaseActions`-Instanz in eine Graphersetzungssequenz, womit die Benutzung in eigenen Programmen deutlich vereinfacht wird. Mehr Informationen über diese GRS-Strings sind im Abschnitt 3.5 zu finden.

3.3 Das LGSP-Backend

Das LGSP-Backend bietet neben den Implementierungen der Schnittstellen und abstrakten Klassen der libGr noch ein paar Zusatzfunktionen an, mit denen die Suchpläne erzeugt werden:

- `LGSPGraph.`*AnalyseGraph*: Analysiert den entsprechenden Graphen und sammelt dabei statistische Informationen, die während der Suchplangenerierung zur Berechnung der Kosten der Operationen verwendet werden. Da die Suchplanberechnung alleine von diesen Informationen abhängt, sollte diese Funktion zeitnah vor der Berechnung von Suchplänen ausgeführt werden, um der aktuellen Struktur des Graphen zu entsprechen. Sie muss jedoch auf jeden Fall vor der ersten Generierung eines Suchplans ausgeführt werden.
- `LGSPActions.`*GenerateSearchPlan*: Erzeugt anhand der statistischen Informationen des entsprechenden Graphen einen dynamischen Suchplan.
- `LGSPActions.`*GenerateSearchPlans*: Erzeugt anhand der statistischen Informationen des entsprechenden Graphen einen oder mehrere dynamische Suchpläne. Da der C#-Übersetzer auf diese Weise für mehrere Regeln nur ein einziges Mal aufgerufen werden muss, sollte diese Funktion der *GenerateSearchPlan*-Funktion vorgezogen werden, wenn mehr als ein Suchplan auf einmal zu generieren ist.
- `LGSPActions.`*ReplaceAction*: Ersetzt eine durch einen Namen gegebene Regel durch eine andere bzw. fügt sie als neue Regel ein, falls dem Namen noch keine Regel zugeordnet worden ist.

Wichtige Details zur Implementierung und wesentliche Unterschiede zur GrGen-Version des LGSP-Backends werden im nächsten Kapitel besprochen.

3.4 Beispiel: „Busy Beaver“ mit der libGr und dem LGSP-Backend

In diesem Abschnitt soll ein „Busy Beaver“ mit der libGr nachgebaut werden, also eine Turingmaschine mit einer bestimmten Anzahl von Zuständen, die so viele Einsen wie möglich auf das Turingband schreibt und terminiert. Als Beispiel diene ein nicht ganz so fleißiger 5-Zustände Biber, der in 2.358.064 Schritten 1.471 Einsen auf das Band schreibt, der in [MB00] unter „No 7“ beschrieben wird.

Um eine Turingmaschine in GrGen.NET zu spezifizieren, wird ein Graphmodell und ein Regelsatz benötigt.

Ein Graphmodell für eine allgemeine, deterministische Turingmaschine

Listing 3.1 stellt ein GrGen.NET Graphmodell dar, in dem das Turingband als eine durch „right“-Kanten verbundene Kette von „BandPosition“-Knoten modelliert wird. Der Wert einer „BandPosition“ wird als reflexive „value“-Kante dargestellt. Der Zustandsautomat besteht aus Zustandsknoten vom Typ „State“, die über „WriteValue“-Knoten miteinander verbunden sind: Von einem „State“-Knoten kommt man über eine „value“-Kante zu einem „WriteValue“-Knoten und von dort über eine „moveLeft“-, „dontMove“- oder „moveRight“-Kante zu dem nächsten Zustandsknoten. Die aktuelle Konfiguration der Turingmaschine wird über eine „rwhead“-Kante modelliert, die von dem aktuellen Zustand zur aktuellen Bandposition zeigt.

Listing 3.1: Graphmodell-Datei TuringModel.gm

```
model TuringModel;

node class BandPosition;
node class State;

node class WriteValue;
node class WriteZero extends WriteValue;
node class WriteOne extends WriteValue;
node class WriteEmpty extends WriteValue;

edge class right
    connect BandPosition [0:1] -> BandPosition [0:1];

edge class value
    connect BandPosition [1] -> BandPosition [1];
edge class zero extends value;
edge class one extends value;
edge class empty extends value;

edge class rwhead;

edge class moveLeft;
edge class dontMove;
edge class moveRight;
```

Ein Regelsatz für eine allgemeine Turingmaschine

Ein Schritt der Turingmaschine besteht nun darin,

- je nach Wert an der aktuellen Bandposition die entsprechende ausgehende Kante vom aktuellen Zustand zu verfolgen,
- den dem „WriteValue“-Knoten entsprechenden Wert (dargestellt als Untertypen) an die aktuelle Bandposition zu schreiben,
- den Schreiblesekopf entsprechend der Kante zwischen dem „WriteValue“-Knoten und dem nächsten Zustand zu bewegen und
- den nächsten Zustand als neuen aktuellen Zustand anzunehmen.

Listing 3.2 teilt diese Schritte aufgrund ihrer Struktur in 9 Regeln auf. Zusätzlich gibt es noch zwei Regeln, die dafür zuständig sind das Band bei Bedarf zu verlängern.

Listing 3.2: Graphersetzungsregel-Datei Turing.grg

```
actions Turing using TuringModel;

rule readZeroRule {
    pattern {
        s:State -:rwhead-> bp:BandPosition -zv:zero-> bp;
        s -zr:zero-> wv:WriteValue;
    }
    replace {
        s -zr-> wv;
        bp -zv-> bp;
        wv -:rwhead-> bp;
    }
}

rule readOneRule {
    pattern {
        s:State -:rwhead-> bp:BandPosition -ov:one-> bp;
        s -or:one-> wv:WriteValue;
    }
    replace {
        s -or-> wv;
        bp -ov-> bp;
        wv -:rwhead-> bp;
    }
}

rule readEmptyRule {
    pattern {
        s:State -:rwhead-> bp:BandPosition -ev:empty-> bp;
        s -er:empty-> wv:WriteValue;
    }
    replace {
        s -er-> wv;
        bp -ev-> bp;
        wv -:rwhead-> bp;
    }
}

rule writeZeroRule {
    pattern {
        wv:WriteZero -rw:rwhead-> bp:BandPosition -:value-> bp;
    }
    replace {
        wv -rw-> bp -:zero-> bp;
    }
}

rule writeOneRule {
    pattern {
        wv:WriteOne -rw:rwhead-> bp:BandPosition -:value-> bp;
    }
    replace {
        wv -rw-> bp -:one-> bp;
    }
}

rule writeEmptyRule {
```

```
    pattern {
        wv:WriteEmpty -rw:rwhead-> bp:BandPosition -:value-> bp;
    }
    replace {
        wv -rw-> bp -:empty-> bp;
    }
}

rule ensureMoveLeftValidRule {
    pattern {
        wv:WriteValue -m:moveLeft-> s:State;
        wv -rw:rwhead-> bp:BandPosition;
        negative {
            bp <-:right- lbp:BandPosition;
        }
    }
    replace {
        wv -m-> s;
        wv -rw-> bp <-:right- lbp:BandPosition -:empty-> lbp;
    }
}

rule ensureMoveRightValidRule {
    pattern {
        wv:WriteValue -m:moveRight-> s:State;
        wv -rw:rwhead-> bp:BandPosition;
        negative {
            bp -:right-> rbp:BandPosition;
        }
    }
    replace {
        wv -m-> s;
        wv -rw-> bp -:right-> rbp:BandPosition -:empty-> rbp;
    }
}

rule moveLeftRule {
    pattern {
        wv:WriteValue -m:moveLeft-> s:State;
        wv -:rwhead-> bp:BandPosition <-r:right- lbp:BandPosition;
    }
    replace {
        wv -m-> s;
        s -:rwhead-> lbp -r-> bp;
    }
}

rule moveRightRule {
    pattern {
        wv:WriteValue -m:moveRight-> s:State;
        wv -:rwhead-> bp:BandPosition -r:right-> rbp:BandPosition;
    }
    replace {
        wv -m-> s;
        s -:rwhead-> rbp <-r- bp;
    }
}
```

```
rule dontMoveRule {
    pattern {
        wv:WriteValue -m:dontMove-> s:State;
        wv -:rwhead-> bp:BandPosition;
    }
    replace {
        bp;
        wv -m-> s;
        s -:rwhead-> bp;
    }
}
```

Ein „Busy Beaver“-C#-Programm

Listing 3.3 verwendet das LGSP-Backend mit Suchplanerzeugung, um die Wartezeit von etwa 42 Minuten mit statischen Suchplänen auf etwa 1 Minute mit dynamischen Suchplänen zu drücken. Dazu werden nach 100 Turingmaschinen-Schritten statistische Informationen über den Graphen gesammelt, mit deren Hilfe die Heuristik deutlich bessere Suchpläne erzeugen kann als der GrGen-Generator, dem diese Informationen nicht zur Verfügung stehen. Es ist allerdings möglich, die statischen Suchpläne durch Angabe von Prioritäten für die Reihenfolge der Suche manuell zu optimieren, indem Graphelemente in den Regeln mit „prio“ attributiert werden.

Listing 3.3: C#-Quelltext BusyBeaver.cs

```
using System;
using de.unika.ipd.grGen.lgsp;
using de.unika.ipd.grGen.models.Turing;
using de.unika.ipd.grGen.actions.Turing;
using de.unika.ipd.grGen.libGr;
using de.unika.ipd.grGen.libGr.sequenceParser;

namespace busyBeaver
{
    class BusyBeaver
    {
        const int L = -1;
        const int D = 0;
        const int R = 1;

        LGSPGraph graph;
        TuringActions actions;

        void GenStateTransition(String srcState, int input,
            String destState, int output, int move)
        {
            IType moveType;
            switch(move)
            {
                case L: moveType = EdgeType_moveLeft.typeVar; break;
                case D: moveType = EdgeType_dontMove.typeVar; break;
                case R: moveType = EdgeType_moveRight.typeVar; break;
                default:
                    throw new ArgumentException("Illegal move: " + move);
            }

            INode writeNode = graph.AddNode(
                output == 0 ? (IType) NodeType_WriteEmpty.typeVar
                            : (IType) NodeType_WriteOne.typeVar);
```

```
        graph.AddEdge(input == 0 ? (IType) EdgeType_empty.typeVar
                                 : (IType) EdgeType_one.typeVar,
            (INode) graph.GetVariableValue(srcState), writeNode);
        graph.AddEdge(moveType, writeNode,
            (INode) graph.GetVariableValue(destState));
    }

    void DoBusyBeaver()
    {
        graph = new LGSPGraph(new TuringGraphModel());
        actions = new TuringActions(graph);

        // Initialize tape
        INode bp = graph.AddNode(NodeType_BandPosition.typeVar);
        graph.AddEdge(EdgeType_empty.typeVar, bp, bp);

        // Initialize states
        INode sA = graph.AddNode(NodeType_State.typeVar, "sA");
        graph.AddNode(NodeType_State.typeVar, "sB");
        graph.AddNode(NodeType_State.typeVar, "sC");
        graph.AddNode(NodeType_State.typeVar, "sD");
        graph.AddNode(NodeType_State.typeVar, "sE");
        graph.AddNode(NodeType_State.typeVar, "sH");

        // Set initial state
        graph.AddEdge(EdgeType_rwhead.typeVar, sA, bp);

        // Create state transtitions
        GenStateTransition("sA", 0, "sB", 1, L);
        GenStateTransition("sA", 1, "sD", 1, L);
        GenStateTransition("sB", 0, "sC", 1, R);
        GenStateTransition("sB", 1, "sE", 0, R);
        GenStateTransition("sC", 0, "sA", 0, L);
        GenStateTransition("sC", 1, "sB", 0, R);
        GenStateTransition("sD", 0, "sE", 1, L);
        GenStateTransition("sD", 1, "sH", 1, L);
        GenStateTransition("sE", 0, "sC", 1, R);
        GenStateTransition("sE", 1, "sC", 1, L);

        // A little warm up for the beaver
        PerformanceInfo perfInfo = new PerformanceInfo();

        actions.ApplyGraphRewriteSequence(SequenceParser.ParseSequence(
            "((readOneRule|readEmptyRule);(writeOneRule|writeEmptyRule);"
            + "(ensureMoveLeftValidRule|ensureMoveRightValidRule);"
            + "(moveLeftRule|moveRightRule)){100}", actions), perfInfo);
        Console.WriteLine(perfInfo.MatchesFound + " matches found in "
            + perfInfo.TotalTimeMS + " ms.");

        // Calculate search plans
        graph.AnalyzeGraph();
        actions.GenerateSearchPlans("readOneRule", "readEmptyRule",
            "writeOneRule", "writeEmptyRule", "ensureMoveLeftValidRule",
            "ensureMoveRightValidRule", "moveLeftRule", "moveRightRule");

        // Go, beaver, go!
        perfInfo.Reset();
        actions.ApplyGraphRewriteSequence(SequenceParser.ParseSequence(
```

```
                "((readOneRule|readEmptyRule);(writeOneRule|writeEmptyRule);"
                + "(ensureMoveLeftValidRule|ensureMoveRightValidRule);"
                + "(moveLeftRule|moveRightRule))*", actions), perfInfo);
            Console.WriteLine(perfInfo.MatchesFound + " matches found in "
                + perfInfo.TotalTimeMS + " ms.");
        }

        static void Main(string[] args)
        {
            BusyBeaver beaver = new BusyBeaver();

            int startTime = Environment.TickCount;
            beaver.DoBusyBeaver();
            int stopTime = Environment.TickCount;

            Console.WriteLine("Time: " + (stopTime - startTime) + " ms");
        }
    }
}
```

3.5 Die GrShell

Im Vergleich zur GrGen-Version der GrShell hat sich viel am Syntax geändert. Es wurde versucht, den Syntax etwas einheitlicher zu gestalten und teilweise auch näher an die Regelspezifikationssprache zu bringen. Zur Fehlersuche bzw. Veranschaulichung der Veränderungen am Arbeitsgraphen wurde ein Debug-Modus eingebaut, in dem man detailiert sehen kann, wie eine Ersetzungsregel angewendet wird, aber auch komplette Schleifen überspringen kann (siehe A.2.3). Zudem wurde Unterstützung für Variablen, Regelparameter und Regelrückgabewerte hinzugefügt und die Graphersetzungssequenzen deutlich erweitert:

Beschreibt $\mathcal{P}$ wieder die Menge der Ersetzungsregeln, $\mathcal{R}$ die Menge der Graphersetzungssequenzen, $\mathcal{V}$ die Menge der Variablen, N die Menge aller Elementnamen, $p \in \mathcal{P}$, $R, R_1, R_2 \in \mathcal{R}$, $x, y, v_1, \ldots, v_l, b_1, \ldots, b_k \in \mathcal{V}$, $q \in N$, $a_1, \ldots, a_m \in \mathcal{V} \cup \mathcal{N}$ und $k, l, n \in \mathbb{N}_0$, so wird die Bedeutung einer solchen Sequenz nun wie folgt definiert:

$(b_1, \ldots, b_l) = p(a_1, \ldots, a_m)$	Die Regel p wird mit den Parametern $(a_1, \ldots, a_m)$ angewendet und die Ergebnisse in die Variablen $(b_1, \ldots, b_l)$ geschrieben. p schlägt fehl, wenn das Muster nicht gefunden wird.
$[(b_1, \ldots, b_l) = p(a_1, \ldots, a_m)]$	Die Regel p wird mit den Parametern $(a_1, \ldots, a_m)$ auf alle Vorkommen des entsprechenden Musters quasi „gleichzeitig“ angewendet und die Ergebnisse der letzten Fundstelle in die Variablen $(b_1, \ldots, b_l)$ geschrieben. Der Benutzer muss dabei sicherstellen, dass eine Ersetzung einer Fundstelle keine andere Fundstelle so beeinflussen kann, dass diese ungültig wird. Ansonsten ist das Verhalten undefiniert. $[p]$ schlägt fehl, wenn das Muster nicht gefunden wird.

$p(a_1, \ldots, a_m)$	Wie $(b_1, \ldots, b_l) = p(a_1, \ldots, a_m)$, nur dass eventuelle Ergebnisse verworfen werden.
$[p(a_1, \ldots, a_m)]$	Wie $[(b_1, \ldots, b_l) = p(a_1, \ldots, a_m)]$, nur dass eventuelle Ergebnisse verworfen werden.
R_1 & R_2	Zuerst wird R_1 und dann, wenn R_1 erfolgreich war, R_2 ausgeführt. Schlägt R_2 fehl, so werden die Änderungen von R_1 rückgängig gemacht. R_1 & R_2 schlägt fehl, wenn sowohl R_1 als auch R_2 fehlschlagen.
$R_1 \mid R_2$	Zuerst wird R_1 ausgeführt. Schlägt R_1 fehl, so wird R_2 ausgeführt. $R_1 \mid R_2$ schlägt fehl, wenn sowohl R_1 als auch R_2 fehlschlagen.
R_1 ; R_2	Zuerst wird R_1 und dann R_2 ausgeführt, selbst wenn R_1 fehlschlägt. R_1 ; R_2 schlägt fehl, wenn sowohl R_1 als auch R_2 fehlschlagen.
$-Operatormodifikator	Ein einem Operator vorangestelltes '$' hebt die Auswertungsreihenfolge des Operators auf. Gleichverteilt wird die linke oder die rechte Seite des Operators als erster Teil der Sequenz aufgefasst.
(R)	Klammerung zur Festlegung der Auswertungsreihenfolge.
$R*$	Die Sequenz R wird so oft angewendet, bis sie fehlschlägt. $R*$ schlägt fehl, wenn R kein mal erfolgreich war.
$R\{n\}$	Die Sequenz R wird so oft angewendet, bis sie fehlschlägt, aber maximal n-mal. $R\{n\}$ schlägt fehl, wenn R kein mal erfolgreich war.
$def(v_1, \ldots, v_k)$	Ist genau dann erfolgreich, wenn alle Variablen $v_1, \ldots, v_k$ belegt sind.
$true$	Ist immer erfolgreich.
$false$	Ist nie erfolgreich.
$x = y$	Der Variablen x wird der Wert der Variablen y zugewiesen. Ist y unbelegt (z.B. die Variable „null“), so ist x danach auch unbelegt.
$x = @(q)$	Die Variable x wird auf das Element mit dem Namen q gesetzt. Existiert ein solches Element nicht (mehr), so ist x danach unbelegt.

Ist ein Parameter einer Regelanwendung eine unbelegte Variable, wird das entsprechende Musterelement wie ein ebenfalls zu suchendes Element betrachtet („Lookup“) und die Variable bleibt unbelegt (sofern sie nicht Empfänger eines Rückgabewertes ist). Da die Suchplanerzeugung davon ausgeht, dass Parameter immer „kostenlos“ sind, kann das Hineinreichen von unbelegten Variablen die Laufzeit erheblich verschlechtern.

Der vollständige Syntax der GrShell-Skriptsprache kann im Anhang A nachgelesen werden.

3.6 Beispiel: „Busy Beaver“ mit der GrShell

Das GrShell-Beispiel ist eine Implementierung des Listings 3.3 in der GrShell-Skriptsprache:

Listing 3.4: GrShell-Skript BusyBeaver_5_7.grs

```
# Initialize environment
select backend "lgspbackend.dll"
new graph "lgsp-TuringModel.dll" "Busy beaver 5 No. 7
    ↪(http://drb9.drb.insel.de/~heiner/BB/bb-list)"
select actions "lgsp-TuringActions.dll"

# Initialize tape
new bp:BandPosition($="Startposition")
new bp-:empty->bp

# Initialize states
new sA:State($="A")
new sB:State($="B")
new sC:State($="C")
new sD:State($="D")
new sE:State($="E")
```

```
new sH:State($="Halt")

# Set initial state
new sA -:rwhead-> bp

#
# Create state transitions
#
new sA_0:WriteOne
new sA -:empty-> sA_0
new sA_0 -:moveLeft-> sB

new sA_1:WriteOne
new sA -:one-> sA_1
new sA_1 -:moveLeft-> sD


new sB_0:WriteOne
new sB -:empty-> sB_0
new sB_0 -:moveRight-> sC

new sB_1:WriteEmpty
new sB -:one-> sB_1
new sB_1 -:moveRight-> sE


new sC_0:WriteEmpty
new sC -:empty-> sC_0
new sC_0 -:moveLeft-> sA

new sC_1:WriteEmpty
new sC -:one-> sC_1
new sC_1 -:moveRight-> sB


new sD_0:WriteOne
new sD -:empty-> sD_0
new sD_0 -:moveLeft-> sE

new sD_1:WriteOne
new sD -:one-> sD_1
new sD_1 -:moveLeft-> sH


new sE_0:WriteOne
new sE -:empty-> sE_0
new sE_0 -:moveRight-> sC

new sE_1:WriteOne
new sE -:one-> sE_1
new sE_1 -:moveLeft-> sC

# Warm up
grs ((readOneRule|readEmptyRule) ; (writeOneRule|writeEmptyRule) ;
    ↪(ensureMoveLeftValidRule|ensureMoveRightValidRule) ;
    ↪(moveLeftRule|moveRightRule)){100}

# Calculate search plans
```

```
custom graph analyze_graph
custom actions gen_searchplan readOneRule readEmptyRule writeOneRule
    ↪writeEmptyRule ensureMoveLeftValidRule ensureMoveRightValidRule
    ↪moveLeftRule moveRightRule

# Let the games begin
grs ((readOneRule|readEmptyRule) ; (writeOneRule|writeEmptyRule) ;
    ↪(ensureMoveLeftValidRule|ensureMoveRightValidRule) ;
    ↪(moveLeftRule|moveRightRule))*
```

Kapitel 4

Das LGSP-Backend im Detail

In diesem Kapitel werden wichtige Unterschiede zum LGSP-C-Backend von GrGen (kurz C-Backend) und einige Entwurfsentscheidungen während der Implementierung des LGSP-C#-Backends von GrGen.NET (kurz C#-Backend) dargestellt.

4.1 Interpretieren oder Generieren?

Wie in Kapitel 2 dargestellt wurde, werden die Suchprogramme im C-Backend interpretiert. Die Gründe dafür sind:

- Die dynamische Erzeugung von komplexem ausführbarem Code ist in C, falls überhaupt, nur mittels externer Bibliotheken möglich.
- Ein kompakter Interpretierer passt vielleicht in den Code-Cache. Zusammen mit einer kompakten Darstellung der Matcherprogramme könnte dies viele Cache-Fehlzugriffe vermeiden und somit sehr performant arbeiten.
- Die Erzeugung der internen Darstellung der Matcherprogramme benötigt sehr wenig Laufzeit.

Das C#-Backend setzt hingegen auf einen generativen Ansatz, d.h. für die Suchprogramme wird zur Laufzeit ausführbarer Code erzeugt. Die Gründe hierfür sind:

- Mit den Bibliotheken des .NET Frameworks ist es sehr einfach möglich, komplexen, zur Laufzeit generierten C#-Code zu ausführbarem Code zu kompilieren. Leider ist das Übersetzen jedoch mit 200-400 ms pro Übersetzung recht laufzeitintensiv.
- Die Zusatzkosten eines Interpretierers entfallen.
- Der Code kann genau an die Erfordernisse der jeweiligen Suchprogramme angepasst werden und durch Übersetzer und Laufzeitübersetzer noch zusätzlich optimiert werden.
- Da voraussichtlich viel zusammenhängende Zeit in einzelnen Matcherprogrammen verbracht wird und der erzeugte Code für diese sehr kompakt ist, sollte auch hier der Code-Cache effizient genutzt werden können.

4.2 Suchplanerzeugung

Das C#-Backend verwendet zur Suchplanerzeugung den in Abbildung 4.1 gezeigten Aufbau und richtet sich nach der Heuristik aus [Bat06]. Im Gegensatz zum C-Backend betrachtet das dort vorgestellte Verfahren die Graphelemente und Operationen einheitlich, was das Verfahren deutlich vereinfacht und auch Lookups auf Kanten erlaubt, was im C-Backend aufgrund der internen Darstellung der Suchmuster nicht möglich war.

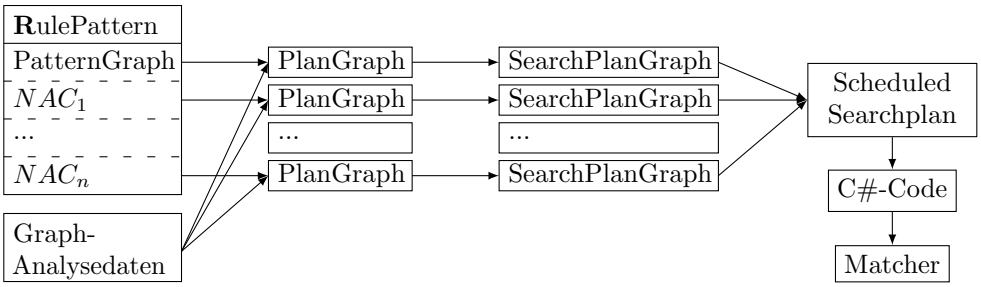


Abbildung 4.1: Aufbau der Suchplanerzeugung im LGSPBackend von GrGen.NET

Die Regel, für die ein Suchplan erzeugt werden soll (siehe Abbildung 4.2(a)), wird in Form eines `RulePattern`-Objektes gegeben, das für das Suchmuster und jede negative Anwendungsbedingung (NAC, Negative Application Condition) ein `PatternGraph`-Objekt enthält. Ein `PatternGraph`-Objekt besteht zum Einen aus Beschreibungen der Knoten und Kanten des jeweiligen Musters und zum Anderen aus Bedingungen und Informationen darüber, welche Graphelemente homomorph statt isomorph auf den Arbeitsgraphen abgebildet werden dürfen.

Aus den `PatternGraph`-Objekten werden *Planungsgraphen* erzeugt (siehe Abbildung 4.2(b)), deren Knoten die zu suchenden Elemente des Musters und deren Kanten die möglichen Suchoperationen darstellen. Die möglichen Suchoperationen sind:

- Preset: Übernimmt ein vorbelegtes Graphelement, das einen gegebenen Typ haben muss.
- Lookup: Sucht ein Graphelement eines gegebenen Typs.
- ExtendIncoming: Sucht eine eingehende Kante eines gegebenen Typs zu einem gegebenen Knoten.
- ExtendOutgoing: Sucht eine ausgehende Kante eines gegebenen Typs zu einem gegebenen Knoten.
- ImplicitSource: Nimmt den Quellknoten einer gegebenen Kante, der einen gegebenen Typ haben muss.
- ImplicitTarget: Nimmt den Zielknoten einer gegebenen Kante, der einen gegebenen Typ haben muss.
- CheckNegative: Verwirft den aktuellen Kandidaten, falls ein „negatives Muster“ auf die bisher gefundenen Elemente passt. (Wird nicht im Planungsgraphen verwendet)
- CheckCondition: Verwirft den aktuellen Kandidaten, falls eine Bedingung, die sich auf Attribute und/oder Typen der bisher gefundenen Elemente bezieht, nicht erfüllt ist. (Wird nicht im Planungsgraphen verwendet)

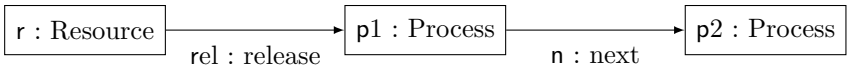
Die Kanten werden mit Kosten annotiert, die sich aus den Graph-Analysedaten ergeben und einen Schätzwert dafür angeben, wie stark sich der Suchraum wahrscheinlich durch die jeweilige Operation aufspalten würde. Zusätzlich enthalten die Planungsgraphen je einen *Root*-Knoten, der als Ausgangspunkt der Suche verwendet wird, an dem also noch kein Element bekannt ist.

Für jeden Planungsgraphen wird nun ein möglichst „guter“ Suchplan $P = \langle o_1, \ldots, o_k \rangle$ mit $o_1, \ldots, o_k$ Folge von Such-Operationen mit den jeweiligen Kosten $c_1, \ldots, c_k \geq 1$ berechnet, indem die Kosten

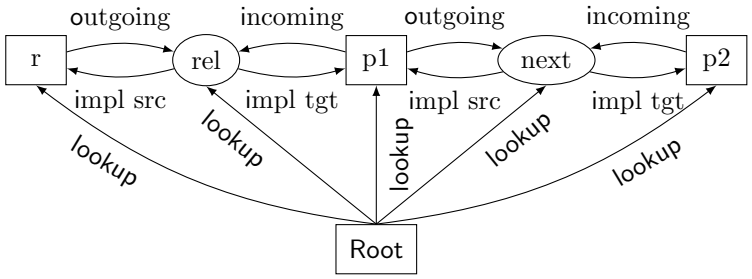
$$c(P) := c_1 + c_1c_2 + c_1c_2c_3 + \cdots + c_1c_2c_3 \ldots c_k$$

minimiert werden.

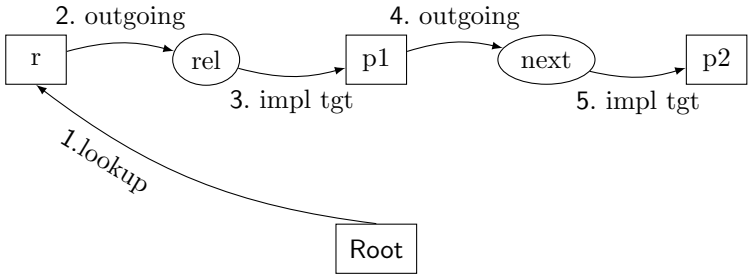
Da der letzte Faktor von $c(P)$, $c(S) := c_1c_2c_3 \ldots c_k$, dominierend ist, versucht die Heuristik zuerst diesen zu minimieren. Dazu werden aus den Planungsgraphen mit einem Algorithmus von Chu



(a) Das Suchmuster



(b) Der Planungsgraph



(c) Der nach der Heuristik berechnete Suchplan

Abbildung 4.2: Die „giveRule“ des Mutex-Benchmarks

und Liu [CL65] minimal aufspannende Arboreszenten[1] gebildet, um eine möglichst billige Auswahl von Operationen zu finden (siehe Abbildung 4.2(c) ohne die Nummerierung zu beachten). Haben zwei Operationen die gleichen Kosten (innerhalb einer kleinen ϵ-Umgebung), werden Operationen in dieser Reihenfolge bevorzugt:

- ImplicitSource/ImplicitTarget: Laufzeitkosten garantiert $\theta(1)$.
- ExtendIncoming/ExtendOutgoing: Hängen zumindest mit einem bereits gefundenen Element zusammen.
- Edge-Lookup: Findet implizit drei Graphelemente auf einmal, ist aber eine unzusammenhängende Operation.
- Node-Lookup: Unzusammenhängende Operation.

Die entstandenen *Suchplangraphen* des Suchmusters und der NACs müssen nun in eine Sequenz von Operationen transformiert werden, damit sie ausgeführt werden können (siehe Abbildung 4.2(c)). Die Heuristik aus [Bat06] sieht hier zunächst vor, für jeden Suchplangraphen die Reihenfolgen der Suchoperationen nach einer *best-first*-Strategie auszuwählen, da nach der Formel für $c(P)$ Operationen um so mehr Einfluss auf die Gesamtkosten haben, je eher sie ausgeführt werden, denn es gilt:

$$c(P) = c_1(1 + c_2(1 + \ldots(1 + c_{k-1}(1 + c_k))\ldots)$$

Die Bedingungen werden danach so „früh“ wie möglich im Suchplan eingeplant, also sobald alle benötigten Elemente im Suchplan bereit stehen. Da die Bedingungen nur von sehr einfachem

[1] Baum mit gerichteten Kanten

Aufbau sind, wird davon ausgegangen, dass sie keinen größeren Einfluss auf die Laufzeit haben. Da String-Vergleiche jedoch im Allgemeinen keine O(1)-Operationen sind, kann diese Vorgehensweise bei sehr langen Strings und sehr früh eingeplanten Bedingungen zu Laufzeitproblemen führen. Wie bereits eingangs erwähnt, wurde das Einplanen der Bedingungen im C-Backend leider nicht implementiert.

Als letztes müssen jetzt noch die NACs eingeplant werden. Dafür wird eine Heuristik verwendet, die jeden NAC der Reihe nach an einer Stelle einplant, so dass gilt:

- Alle von diesem NAC benötigten Elemente wurden an dieser Stelle bereits gefunden. (Dies ist für die Korrektheit notwendig)
- Die Kosten des NACs sind nicht höher als die der von dieser Stelle aus restlichen Operationen. (Dies soll die Gesamtlaufzeit reduzieren).

Existiert keine solche Stelle, wird der jeweilige NAC an den Schluss des Suchplans gesetzt. Mit der zweiten Bedingung für die Platzierung der NACs soll dafür gesorgt werden, dass das positive Muster bevorzugt wird, wenn es einen negativen Kandidaten potentiell schneller aussortieren kann als ein NAC. Das C-Backend plante alle NACs grundsätzlich am Schluss des Suchplans ein.

Der nun fertige und in Form eines `ScheduledSearchplan`-Objektes gespeicherte Suchplan wird dann in einen möglichst gut angepassten C#-Quelltext übersetzt, der dann mit Hilfe von .NET-Bibliotheksfunktionen zu einem `LGSPAction`-Objekt weiterverarbeitet wird. Dieses `LGSPAction`-Objekt kann dann als „Matcher“, also einem Programm zum Suchen des entsprechenden Musters, eingesetzt werden. An dem Graph*ersetzungs*teil, der von dem GrGen-Generator erzeugt worden ist, ändert sich nichts.

4.3 Implementierung mit Hürden

Der zuerst beschrittene Weg der Erzeugung des dynamischen Codes war nicht, C#-Code zu generieren und diesen dann zu kompilieren, sondern direkt über `System.Reflection.DynamicMethod` Bytecode zu generieren, der dann nur noch „gejittet“ werden müsste. Somit hätte man die etwa 200-400 ms des Übersetzens gespart, was insbesondere häufigere Anpassungen der Suchpläne praktikabel machen würde. Durch einen Fehler in .NET blieb dieser Weg jedoch leider versperrt:

Wenn man versucht, auf ein beliebiges öffentliches Feld einer Instanz einer generischen Klasse mit einem Klassen-Typparameter zuzugreifen, erhält man eine `FieldAccessException`. Nimmt man statt einem Klassen-Typparameter einen Wert-Typ-Typparameter, so funktioniert es ohne Probleme. Ein Beispiel-Programm, das diesen Fehler produziert sieht wie folgt aus:

Listing 4.1: FieldAccesException-Bug Listing

```
using System;
using System.Collections.Generic;
using System.Reflection.Emit;
using System.Reflection;

namespace TestIlGenericSmall
{
    public class Test<T>
    {
        public String field;
        public T unrelatedField;
        public Test(String val) { field = val; }
    }

    public delegate String TestInvoker(Test<String> testarg);

    class Program
    {
```

```
        static void Main(string[] args)
        {
            Type[] methodArgs = { typeof(Test<String>) };
            DynamicMethod method = new DynamicMethod("Dynamic test method",
                typeof(String), methodArgs, typeof(Program).Module);
            ILGenerator ilMatcher = method.GetILGenerator();
            ilMatcher.Emit(OpCodes.Ldarg_0);
            FieldInfo finfo = typeof(Test<String>).GetField("field");
            ilMatcher.Emit(OpCodes.Ldfld, finfo);
            ilMatcher.Emit(OpCodes.Ret);
            TestInvoker test = (TestInvoker) method.CreateDelegate(
                typeof(TestInvoker));
            Test<String> testarg = new Test<String>("7");
            test(testarg);
        }
    }
}
```

Der Fehler wurde zwei Mal an Microsoft gemeldet:

- Am 26.7.2006 mit dem Titel „Accessing fields of generic classes with class type variables in dynamic code causes a FieldAccessException right before execution“ (http://connect.microsoft.com/VisualStudio/feedback/ViewFeedback.aspx?FeedbackID=171578): Geschlossen als „by design“ am 8.8.2006.
- Am 8.10.2006 mit dem Titel „Access to field of instance of generic class with a class typettype variable from emitted code leads to FieldAccessException“ (http://connect.microsoft.com/VisualStudio/feedback/ViewFeedback.aspx?FeedbackID=221225): Geschlossen als „not reproducible“ am 9.11.2006. Hier hat jedoch ein „Program Manager“ von Microsoft zu gegeben, dass es ein Fehler sei und er korrigiert werden würde, der Status „not reproducible“ aber gesetzt worden wäre, weil es intern so gehandhabt werden würde.

Da im LGSP-Backend eine gemeinsame Oberklasse der Graphelemente eine generische, invasive Liste verwendet, ist ein Zugriff auf diese Liste aus auf diesem Weg generiertem Bytecode nicht möglich.

Kapitel 5

Resultate und Bewertung

Das erste Ziel der Studienarbeit war es, die Laufzeit des Graphersetzungssystems durch die Portierung von GrGen von C nach C#/.NET nicht „wesentlich zu verschlechtern“. Inwieweit das gelungen ist, soll dieses Kapitel zeigen.

5.1 Das Benchmark-System

Um GrGen.NET mit GrGen und auch anderen Graphersetzungssystemen vergleichen zu können wurden zwei Probleme verwendet:

- Der Mutex-Benchmark von Varró [VSV05], der auch für GrGen verwendet wurde [GBG+06]: Ein Ring von N Prozessen wird aufgebaut, von dem alle Prozesse eine bestimmte Ressource allozieren wollen. Welcher Prozess die Ressource erhalten darf, wird durch ein Token bestimmt, und wird von dem entsprechenden Prozess nach der Allokation und Freigabe an den nächsten Prozess im Ring weitergereicht. Wie in [VSV05] näher beschrieben, gibt es 6 Regeln, die mit folgenden Häufigkeiten aufgerufen werden:

Regel	Anzahl der Anwendungen
newRule	N - 2
mountRule	1
requestRule	N
takeRule	N
releaseRule	N
giveRule	N
Gesamt	5 * N - 1

  Der erzeugte Graph hat maximal $N + 1$ Knoten und 2 * $N + 1$ Kanten.

  Bei diesem Benchmark wird zwischen zwei Varianten unterschieden:

  STSmany: Nutzt keine ins Modell eingebaute Informationen über Kardinalitäten aus, wie zum Beispiel: Ein Prozess kann maximal eine „token“-Kante besitzen („multiplicity optimizations = off“).

  STSone: Nutzt ins Modell eingebaute Informationen über Kardinalitäten aus, um die Graphdarstellung und/oder die Regeln zu optimieren („multiplicity optimizations = on“).

  GrGen und GrGen.NET unterstützen zwar Kardinalitäten, erzwingen diese jedoch nicht. Sie werden bisher nur zur Validierung des Graphen verwendet, da es durchaus erlaubt ist zwischen zwei Regelanwendungen einen „illegalen“ Graphen zu erzeugen, um einfachere Regeln schreiben zu können. Daher können die Datenstrukturen aber auch nicht an feste Kardinalitäten angepasst werden, wie es bei Fujaba der Fall ist.

Tabelle 5.1: Laufzeiten der Mutex-Benchmarks (in Millisekunden)

Benchmark → / Tool ↓	Mutex (STSmany) 10	100	1.000	10.000	100.000	1.000.000
AGG	330	8.300	6.881.000	–	–	–
VarróDB	4.697	19.825	593.500	–	–	–
PROGRES	12	946	459.000	–	–	–
GrGen(PSQL)	30	760	27.715	–	–	–
Fujaba(Varró)	18	49	781	127.234	–	–
Fujaba(Kroll)	34	62	233	30.149	3.077.422	–
GrGen(SP, GrShell)	< 1	1	12	106	1.068	10.768
GrGen.NET(Mono, GrShell)	9	10	14	71	751	7.907
GrGen.NET(.NET, GrShell)	20	23	25	57	468	7.616
GrGen.NET(.NET, direkt)	58	57	60	81	330	5.803
GrGen.NET(Mono, direkt)	65	65	69	107	679	5.542
Fujaba(Kroll, STSone)	30	46	156	11.378	1.423.918	–

Tabelle 5.2: Laufzeiten des „Busy Beaver“-Benchmarks (in Millisekunden)

Benchmark → / Tool ↓	Busy Beaver 5 No 7	5 No 1
GrGen(SP)	34.949	> 2,2 GiB
GrGen.NET(Mono, GrShell)	10.240	179.244
GrGen.NET(.NET, GrShell)	8.743	169.588
GrGen.NET(Mono, direkt)	4.484	72.309
GrGen.NET(.NET, direkt)	3.042	67.167
Fujaba(Kroll)	1.657	36.065

- Zwei „Busy Beaver“ mit 5 Zuständen (plus Haltezustand): Eine optimierte Version des in 3.4 beschriebenen „Busy Beavers“ („5 No 7“) und der aktuelle Weltrekordhalter („5 No 1“).

Name	#Einsen	#Schritte	#Regel-anwendungen	GrGen/GrGen.NET #Knoten	#Kanten	Fujaba #Knoten
5 No 7	1.471	2.358.064	7.076.157	1.982	3.952	1.983
5 No 1	4.098	47.176.870	141.542.898	12.305	24.598	12.306

  Da Fujaba Kanten nur implizit als Attribute von Knoten darstellt, wurde das Modell für Fujaba leicht geändert: Aus der Lese-Schreibkopf-Kante wurde eine Pseudo-Hyperkante, die es erlaubt, einen „EdgeLookup“ zu simulieren. Die Pseudo-Hyperkante besteht aus einem „Head“ Knoten und zwei Kanten.

Die Messungen wurden wie in [GBG+06] auf einem AMD Athlon XP 3000+ mit 1 GiB Hauptspeicher ausgeführt, um einige der Werte übernehmen zu können. Neue Messungen wurden jeweils 40-mal wiederholt, um Schwankungen zu minimieren. Fujaba und GrGen.NET(.NET) wurden unter WinXP gemessen, GrGen(SP) und GrGen.NET(Mono) unter Suse Linux 9.3. Verwendet wurden Microsoft .NET 2.0.50727.42, Mono 1.2.3.1 und Sun Java 1.6.0_01.

5.2 Laufzeitmessungen

Abbildung 5.1 zeigt einen Vergleich zwischen AGG [ERT99], VarróDB [VFV04], PROGRES [Sch99], GrGen(PSQL, GrShell) [Hac03], Fujaba [Fuj06], GrGen(SP, GrShell) [GBG+06] und nicht zuletzt GrGen.NET. Die Messungen für AGG, GrGen(PSQL, GrShell) und GrGen(SP, GrShell) wurden bis $N = 1000$ von [GBG+06] übernommen, die Messungen für VarróDB und PROGRES stammen ursprünglich von [Var05]. Da einige der Zeiten schon im Bereich der Messungenauigkeit

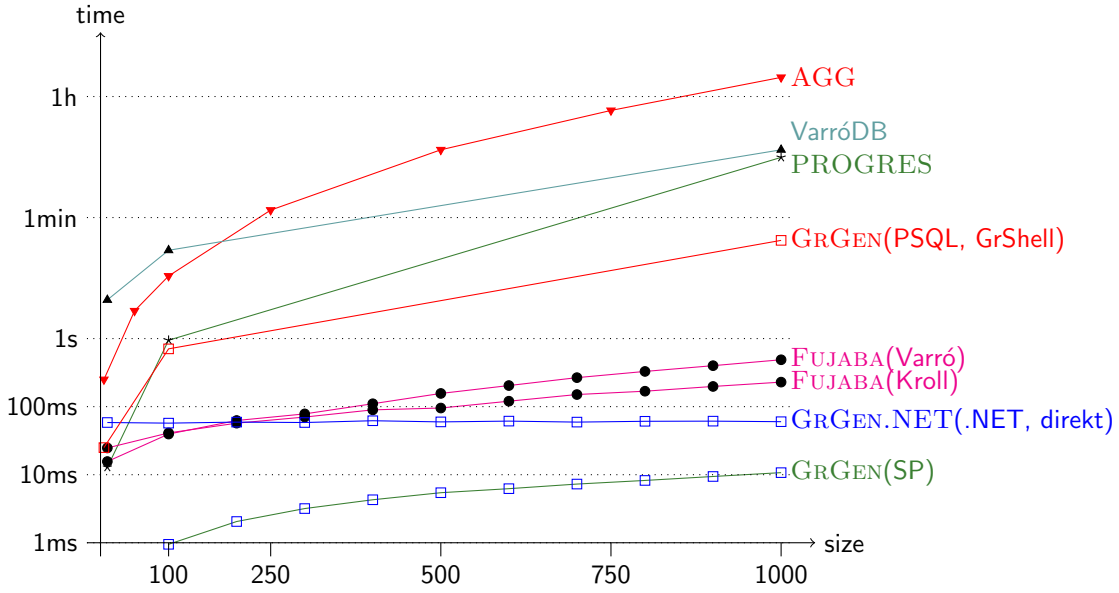


Abbildung 5.1: Laufzeiten des STS Mutex Benchmarks (multiplicity optimizations=off, parameter passing=off, simultaneous execution=off; für Details über die Parameter siehe [VSV05])

lagen, wurde die Problemgröße für die schnelleren Tools auf bis zu 1.000.000 erhöht (siehe Tabelle 5.1 und Abbildung 5.2).

Leider mussten wir jedoch feststellen, dass zumindest Varrós Messungen für Fujaba (Fujaba(Varró)) gravierende Unterschiede zu unseren Messungen aufwiesen: Abbildung 5.3 lässt uns vermuten, dass die Laufzeiten der Fujaba Messung von Varró von der Geschwindigkeit der Konsolenausgabe und seiner Grafikkarte dominiert wird. Außerdem haben seine takeRule, releaseRule und giveRule einen Aufwand von $O(N)$ („Ohne Protokoll“). Mit kleinen Änderungen ließ sich der Aufwand dieser Suchpläne jedoch auf $O(1)$ reduzieren und somit die Gesamtlaufzeit auf das 0,22-fache reduzieren („Optimierte Regeln ohne Protokoll“, Fujaba(Kroll)). Da die requestRule aber immer noch einen Aufwand von $O(N)$ besitzt, bleibt die Gesamtlaufzeit in $O(N^2)$.

Da Fujaba für 1-zu-1 und 1-zu-n Verhältnisse sehr effiziente Darstellungen hat, bieten die Fujaba(Kroll, STSone) Messungen neben den „optimierten Regeln“ eine deutlich besser an Fujaba angepasste und damit fairere Implementierung des Mutex-Benchmarks für den Vergleich mit GrGen und GrGen.NET. Abbildung 5.2 zeigt jedoch, dass Fujaba selbst mit diesen beiden Hilfestellungen (Angabe von Multiplizitäten und handoptimierte Regeln) von GrGen.NET geschlagen wird. Wie der Vergleich mit den Messungen von Varró zeigte, ist es auch für erfahrene Graphersetzer nicht immer leicht, Regeln von Hand zu optimieren. Insbesondere bei komplexen Mustern und Graphen kann dies eine zeitraubende und fehleranfällige Aufgabe sein. GrGen und GrGen.NET können diese Aufgabe übernehmen und sich auf Anforderung sogar zur Laufzeit einer Graphstruktur anpassen, die sich dynamisch ändert. Dies ist bei Fujaba nicht automatisch möglich.

Der Vergleich zwischen GrGen(SP, GrShell) und GrGen.NET (siehe auch Abbildung 5.4) zeigt, dass GrGen.NET das in der Einleitung genannte Ziel, „die Leistung nicht wesentlich zu verschlechtern“, nicht nur erfüllt, sondern bei Weitem übertroffen hat! Beim Mutex-Benchmark mit $N = 1.000.000$ benötigt GrGen.NET(.NET, GrShell) 29,3% weniger Laufzeit als GrGen(SP, GrShell), beim „Busy Beaver 5 No 7“-Benchmark sogar 75,0% weniger.

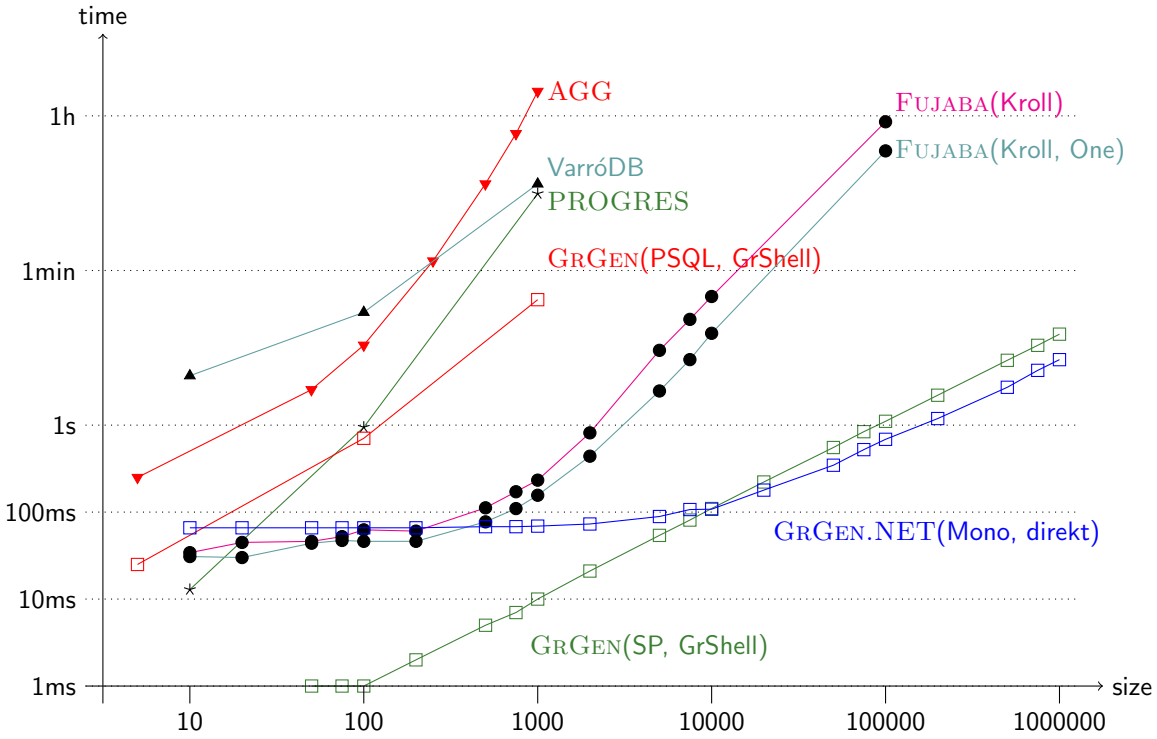


Abbildung 5.2: Laufzeiten des STSmany Benchmarks für verschiedene Systeme und STSone für Fujaba

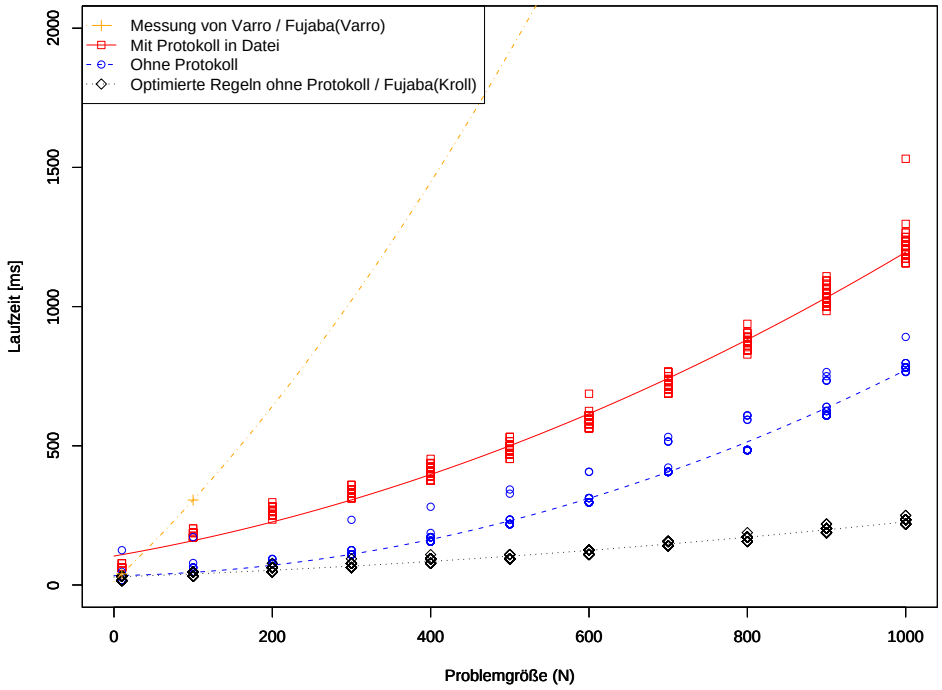


Abbildung 5.3: Verschiedene Laufzeitmessmethoden für Fujaba mit STSmany

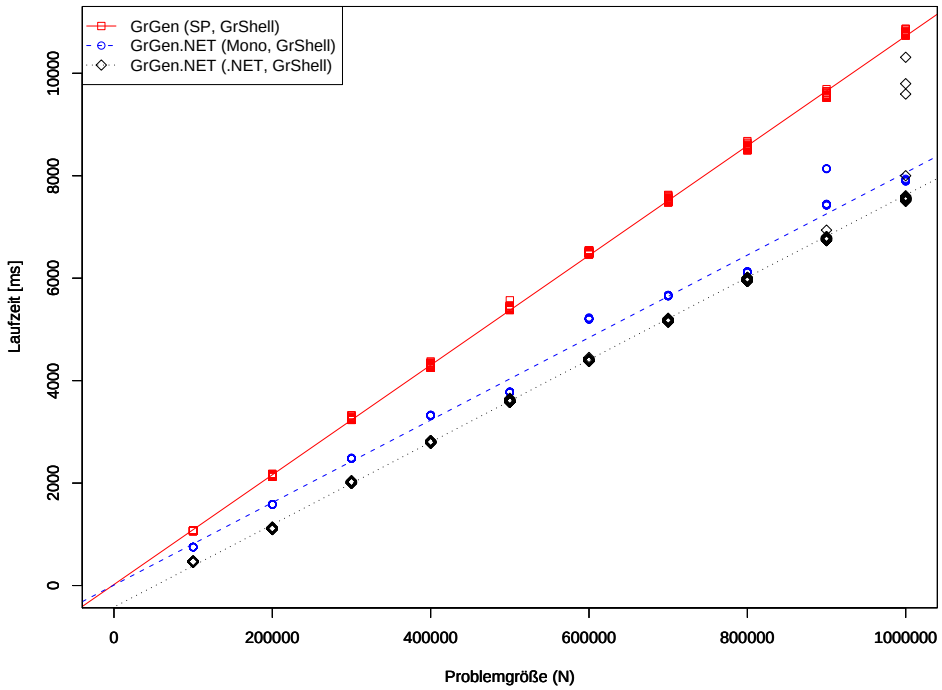


Abbildung 5.4: Laufzeiten von GrGen(C) und GrGen.NET mit dem Mutex-Benchmark

5.3 Speichermessungen

Die Tabellen 5.3 und 5.4 zeigen den Speicherverbrauch der schnelleren Tools, Fujaba, GrGen(SP) und GrGen.NET, für die verwendeten Benchmarks. Gemessen wurde der belegte virtuelle Speicher, da der belegte physische Speicher die ausgelagerten Seiten nicht berücksichigt. Hierfür wurde unter Windows die „PeakPagedMemorySize64“-Eigenschaft der jeweiligen Prozesse verwendet und unter Linux die „VSZ“-Spalte („virtual memory size of the process in KiB“) des „ps“-Befehls, da in Mono die „PeakPagedMemorySize64“-Eigenschaft nicht implementiert ist. Der Speicherverbrauch enthält somit neben dem Speicher für den Programmcode und die Daten auch den belegten Speicher der Java VM bzw. der .NET/Mono CLR.

Beim Vergleich von Fujaba(Kroll) und Fujaba(Kroll, STSone) wird die oben erwähnte „effizientere Darstellung“ bei gegebenen Multiplizitäten besonders deutlich.

Der im Vergleich zu GrGen.NET deutlich höhere Speicherverbrauch von GrGen(SP) (siehe 5.5) ist zum Einen dadurch zu erklären, dass GrGen(SP) für die Speicherallokation einen Obstack verwendet und Speicher nie mehr freigibt, und zum Anderen durch kleinere Graphelemente. Denn während in GrGen(SP) Knoten und Kanten jeweils 80 Byte kosten, kostet ein Knoten in GrGen.NET(.NET) nur 44 Byte und eine Kante 56 Byte.

Tabelle 5.3: Speicherverbrauch des Mutex-Benchmarks (in KiB)

Benchmark →	Mutex (STSmany)					
Tool ↓	10	100	1000	10000	100000	1000000
Fujaba(Kroll)	27.696	29.380	29.380	38.028	227.312	–
GrGen(SP, GrShell)	7.836	7.972	8.396	12.164	49.828	444.648
GrGen.NET(Mono, GrShell)	21.588	21.584	21.816	24.056	40.864	214.668
GrGen.NET(Mono, direkt)	18.204	18.204	18.412	20.100	37.152	210.268
GrGen.NET(.NET, GrShell)	9.354	9.355	9.366	10.690	25.313	178.945
GrGen.NET(.NET, direkt)	7.832	8.044	8.131	10.176	24.888	178.644
Fujaba(Kroll, STSone)	27.702	27.637	29.384	29.384	36.407	–

Tabelle 5.4: Speicherverbrauch des „Busy Beaver“-Benchmarks (in KiB)

Benchmark →	Busy Beaver	
Tool ↓	5 No 7	5 No 1
GrGen(SP, GrShell)	311.984	> 2,2 GiB
Fujaba(Kroll)	27.908	27.912
GrGen.NET(Mono, GrShell)	21.888	24.200
GrGen.NET(Mono, direkt)	20.336	22.932
GrGen.NET(.NET, GrShell)	10.180	11.396
GrGen.NET(.NET, direkt)	9.552	11.268

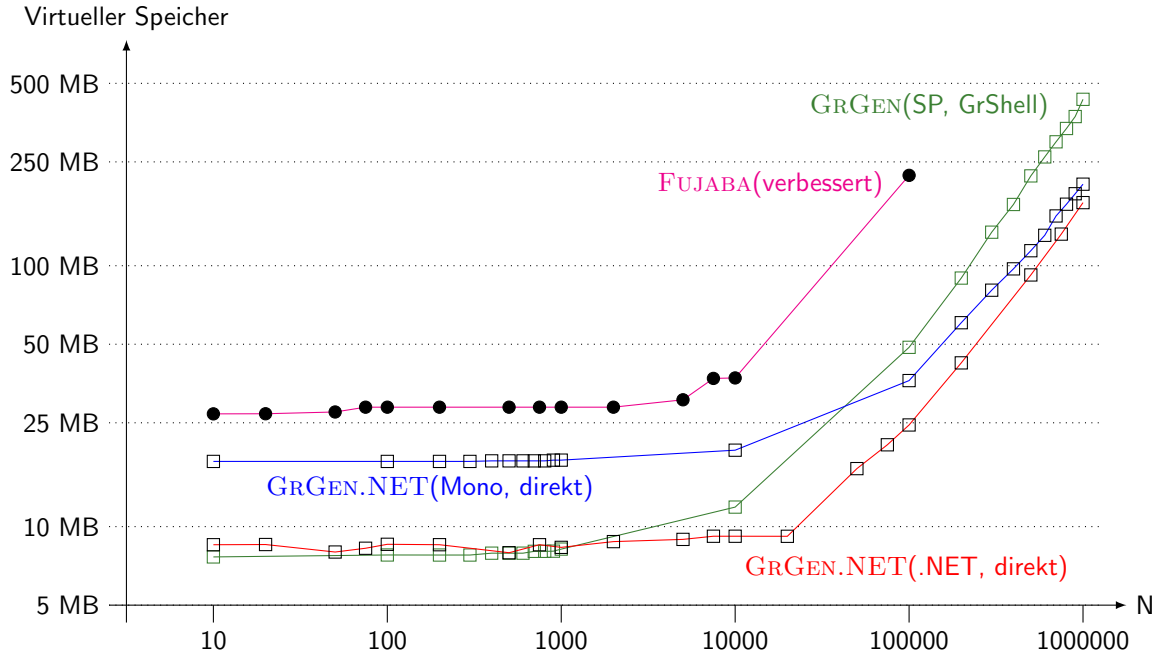


Abbildung 5.5: Speicherverbrauch des Mutex-Benchmarks mit verschiedenen Tools

5.4 Bewertung der Heuristik

Abbildung 5.6 zeigt am Beispiel der „releaseRule“ des Mutex-Benchmarks, wie gut die Heuristik einen Suchplan unter allen möglichen auswählt. c(P) und c(S) sind die in Kapitel 4.2 beschriebenen Kosten nach dem Kostenmaß der Heuristik (rechte Achse). Die Laufzeiten wurden mit GrGen.NET(.NET) gemessen. Insbesondere zeigt das Diagramm, dass Suchpläne mit den geringsten Kosten auch niedrige Laufzeiten haben, und damit auch, dass die Heuristik ein vernünftiges Mittel zur Suchplanerzeugung darstellt. Die sprunglos übergehenden Laufzeiten der Suchpläne mit mittleren c(S)-Kosten sind durch einen aus GrGen(SP) übernommenen Implementierungstrick zu erklären: nachdem ein Muster gefunden wurde, wird an etwa der „gleichen Stelle“ weitersucht, ohne schon mal betrachtetes vorher nochmal zu durchsuchen.

Bei den Messungen wurde ein Timeout von 10 Sekunden verwendet. Höhere Laufzeiten für besonders schlechte Suchpläne sind also eventuell abgeschnitten.

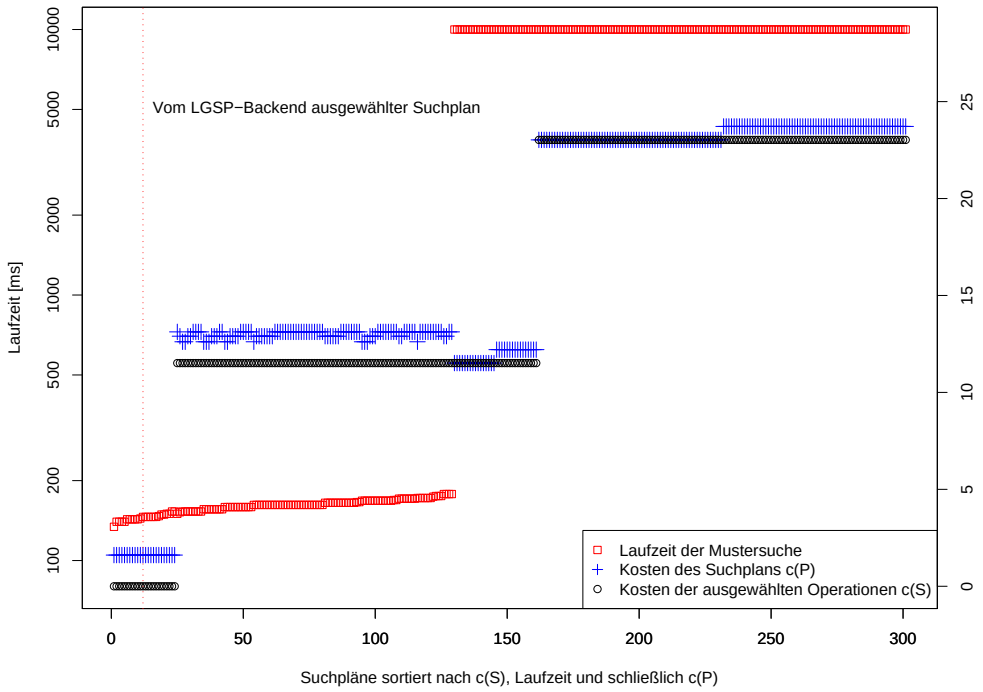


Abbildung 5.6: Laufzeiten und Kosten aller Suchpläne der Mutex-Regel releaseRule mit N=100000

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Mit GrGen.NET wurde ein Graphersetzungssystem entwickelt, das

- über die mächtige und ausdrucksstarke Spezifikationssprache von GrGen verfügt, die für GrGen.NET um Parameter und Rückgabewerte und um einen besseren Umgang mit Kanten erweitert wurde,
- einen deutlich geringeren Speicherverbrauch als GrGen hat,
- eine enorm gesteigerte Geschwindigkeit gegenüber des ohnehin schon schnellen GrGen aufweist und
- eine große Hilfe beim Suchen nach fehlerhaften Regeln anbietet.

Die Heuristik von [Bat06] wurde um ein paar Details erweitert, um vorbelegte Elemente, NACs und Bedingungen einzuplanen und Operationen bei gleichen Kosten auszuwählen.

6.2 Ausblick

Folgende Punkte sind unberücksichtigt geblieben:

- Durchschnittlicher Ausgangsgrad: Die Kanten werden mit Kosten annotiert, die sich aus den Graph-Analysedaten ergeben und einen Schätzwert dafür an, wie stark sich der Suchraum wahrscheinlich durch die jeweilige Operation aufspalten würde. Insbesondere wird also zum Beispiel aufgrund der flachen Kantendarstellung in Knoten[1] ein sehr hoher durchschnittlicher Ausgangsgrad eines Knotens von einem bestimmten Typ nicht berücksichtigt, wenn das Suchmuster an dieser Stelle eine Kante eines Typs verwendet, die an diesem Knoten nur sehr selten vorkommt. In einem solchen Fall wäre vielleicht ein Lookup einem ExtendOutgoing vorzuziehen, um nicht unnötig über viele ausgehende Kanten iterieren zu müssen.
- Kosten für Bedingungen: Könnte man z.B. die Aussage machen, dass nur für 1% der Knoten eines bestimmten Typs ein Attribut „x“ den Wert 42 hat, wäre es vielleicht sinnvoll Knoten diesen Typs früher einzuplanen, um mit dieser Bedingung den Suchraum schneller einschränken zu können. Auch wären vielleicht Kosten für teure String-Vergleiche einzubeziehen.
- Einfluss von NACs auf Suchplan des positiven Musters: Wie bei den Bedingungen könnten auch wahrscheinlich zuschlagende NACs es rechtfertigen, ein Element früher einzuplanen, um den Suchraum schneller einzuschränken.

[1] ein- und ausgehende Kanten werden in den Knoten nicht nach ihrem Typ sortiert

- Optimum-Scheduling statt Best-First-Scheduling: Vielleicht könnte es für größere Muster von Vorteil sein, mehr Aufwand in die Minimierung von c(P) zu stecken, also eine bessere Anordnung der Suchoperationen zu finden.
- Negative Suchpläne in negativen Suchplänen sind bisher noch nicht möglich.
- Debugging von nicht gefundenen Mustern: Bei großen Mustern (z.B. 80 Graphelemente) und entsprechend großen Arbeitsgraphen ist es für den Benutzer nicht mehr so leicht, überhaupt eine Stelle zu finden, wo das Muster seiner Meinung nach gefunden werden sollte. In einem solchen Fall will man wahrscheinlich mehr als nur die Aussage, dass das Muster nicht gefunden wurde. Man will vielleicht wissen, was noch fehlte, um das Muster zu komplettieren, oder welche Stellen am nächsten am Muster dran waren.
- Benchmark mit dynamisch ändernden Graphstrukturen: Interessant wären noch komplexere Benchmarks, bei denen sich die Struktur des Arbeitsgraph zur Laufzeit immer wieder ändert, so dass die Suchpläne entsprechend angepasst werden müssen. Dann könnte GrGen.NET seinen Vorteil der dynamischen Suchplanerzeugung erst richtig ausspielen.

Danksagung

Ich möchte an dieser Stelle den Mitarbeitern und Studenten des IPD Goos für ihre großartige Unterstützung und das sehr angenehme Arbeitsklima bedanken. Dank ihnen habe ich die Entwicklung von GrGen.NET sehr genossen. Insbesondere danke ich Rubino Geiß und Christoph Mallon für die unzähligen und produktiven Diskussionen. Dank gilt auch Tom Gelhausen, Jens Müller und Oliver Denninger, die so manches Wanzennest aufgedeckt haben.

Anhang A

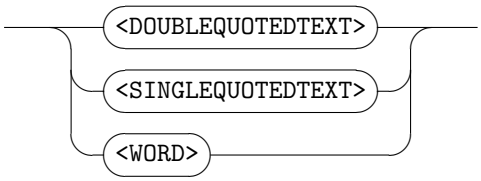
GrShell-Syntaxbeschreibung

Die GrShell-Skriptsprache ist eine zeilenorientierte Sprache, mit der man Graphen erzeugen, manipulieren und ausgeben kann und Graphersetzungen anwenden und debuggen kann.

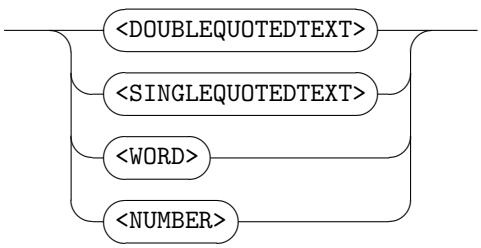
A.1 Allgemeine Definitionen

Zuerst das allerwichtigste: Kommentare werden mit einem '#' eingeleitet und durch ein Zeilenende oder das Dateiende terminiert.

Text

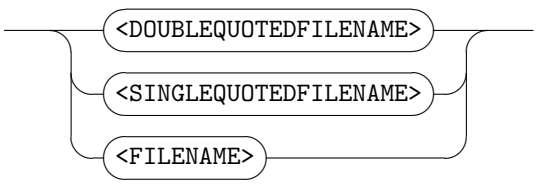


TextOrNumber



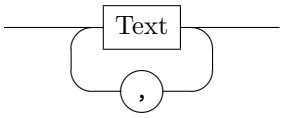
Number

Filename

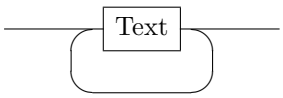


CommandLine

Parameters



SpacedParameters



Diese Bausteine werden in den folgenden Regeln verwendet, um Text, Zahlen, Dateinamen und Parameter darzustellen. Dabei gelten diese Definitionen:

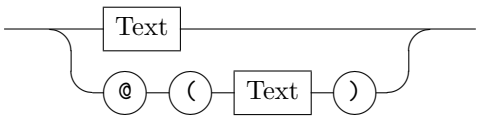
<WORD>:	Eine Folge von Buchstaben, Ziffern und Unterstrichen, die nicht mit einer Ziffer beginnen darf.
<DOUBLEQUOTEDTEXT>:	Beliebiger Text in Anführungsstrichen (" ").
<SINGLEQUOTEDTEXT>:	Beliebiger Text in einfachen Anführungsstrichen (' ').
<NUMBER>:	Eine zusammenhängende Folge von Ziffern.
<FILENAME>:	Ein Dateiname (evtl. mit Pfad) ohne Leerzeichen.
<DOUBLEQUOTEDFILENAME>:	Beliebiger Dateiname in Anführungsstrichen (" ").
<SINGLEQUOTEDFILENAME>:	Beliebiger Dateiname in einfachen Anführungsstrichen (' ').
<COMMANDLINE>:	Beliebiger String bis zum Zeilenende.

Um die möglichen Eingaben bei einigen Befehlen genauer zu beschreiben, werden folgende Spezialisierungen von „Text“ definiert:

NodeType:	Der Name eines Knotentypen des Modells des aktuellen Graphen.
EdgeType:	Der Name eines Kantentypes des Modells des aktuellen Graphen.
AttributeName:	Der Name eines Attributes.
Graph:	Der Name eines Graphen.
Action:	Der Name einer Regel.
Color:	Einer der folgenden Farbnamen: Black, Blue, Green, Cyan, Red, Purple, Brown, Grey, LightGrey, LightBlue, LightGreen, LightCyan, LightRed, LightPurple, Yellow, White, DarkBlue, DarkRed, DarkGreen, DarkYellow, DarkMagenta, DarkCyan, Gold, Lilac, Turqouise, Aquamarine, Khaki, Pink, Orange, Orchid.

Graph-Elemente können sowohl über Variablennamen, als auch über ihre permanenten Namen angesprochen werden:

GraphElement

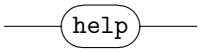


Die Spezialisierungen „Node“ und „Edge“ von „GraphElement“ fordern, dass das Graph-Element ein Knoten bzw. eine Kante sein muss.

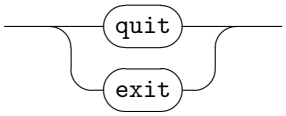
A.2 GrShell-Befehle

Nachdem nun einige grundlegende Element definiert wurden, können die einzelnen Befehle der GrShell erklärt werden. Ein Befehl muss immer mit einem Zeilenumbruch, einem ';;' oder dem Ende der Datei enden.

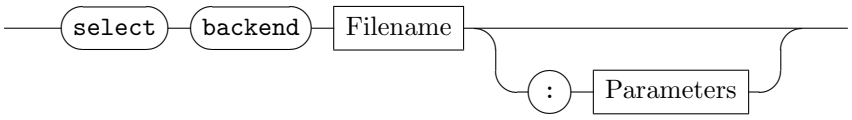
A.2.1 Allgemeine Befehle



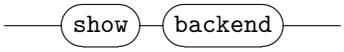
Gibt einen Hilfetext aus, der die möglichen Befehle beschreibt.



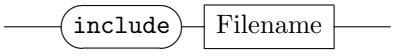
Beendet die GrShell. Wenn die GrShell im Debug-Modus ist, wird auch yComp [LGH+07] geschlossen.



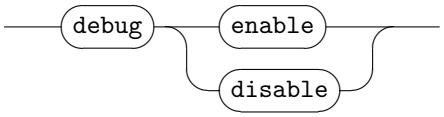
Wählt ein Backend für die Repräsentation des Graphen und der Regeln aus. „Filename“ muss dabei auf ein .NET Modul verweisen, dass das IBackend-Interface genau einmal implementiert (z.B. „lgspBackend.dll“).
Optional können komma-separierte Text-Parameter angegeben werden, sofern das Backend diese unterstützt.



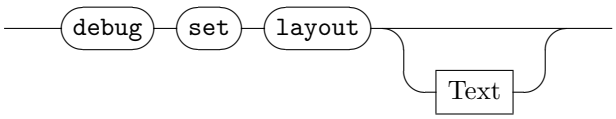
Zeigt alle möglichen Parameter des aktuellen Backends an.



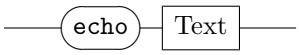
Führt das angegebene GrShell-Skript aus.



Schaltet den Debugmodus ein bzw. aus. Im Debugmodus wird der aktuelle Arbeitsgraph in einem yComp-Fenster [LGH+07] angezeigt und alle Änderungen am Graphen sofort an yComp übertragen.



Setzt den Typ des Graphlayouters. Wird kein Typ angegeben, so wird eine Liste der verfügbaren Layouter angezeigt.

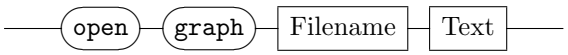


Gibt den übergebenen Text auf der Konsole aus.

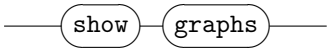
Lässt die gegebene Kommandozeile durch das Betriebssystem ausführen. Um einen Konsolenbefehl auszuführen kann z.B. unter Linux „!sh -C "ls | grep stuff"“ verwendet werden.

A.2.2 Graphen-Befehle

Erzeugt einen neuen Graphen mit dem in „Filename“ spezifizierten Modell und dem in „Text“ angegebenen Namen. Das Modell kann sowohl in Form einer C#-Sourcecode-Datei (.cs) als auch einer .NET-Bibliothek (.dll) übergeben werden, solange die Datei das IGraphModel-Interface genau einmal implementiert.

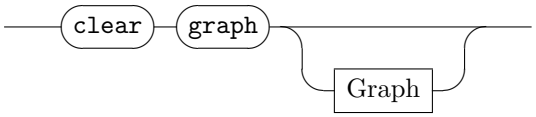


Öffnet einen im Backend vorhandenen Graph mit dem in „Text“ spezifizierten Namen und dem in „Filename“ angegebenen Modell.
Da das LGSP-Backend keine persistenten Graphen unterstützt und bisher kein anderes Backend implementiert ist, ist eine Verwendung dieses Befehls noch nicht sinnvoll.

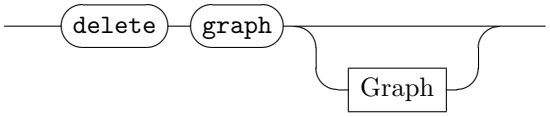


Zeigt alle existierenden Graphen an.

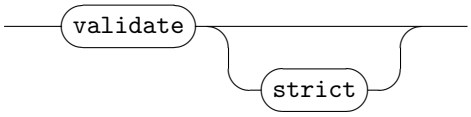
Wählt den aktuellen Arbeitsgraphen aus.



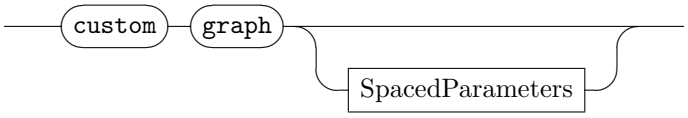
Löscht alle Graphelemente im aktuellen Arbeitsgraphen bzw. im angegebenen Graphen.



Löscht den aktuellen Arbeitsgraphen bzw. den angegebenen Graphen.



Überprüft, ob der aktuelle Arbeitsgraph alle im Graphmodell spezifizierten Verbindungsbedingungen erfüllt. Im „strict“-Modus müssen alle vorhandenen Verbindungen zwischen Knoten spezifiziert sein, damit der Graph „gültig“ ist.

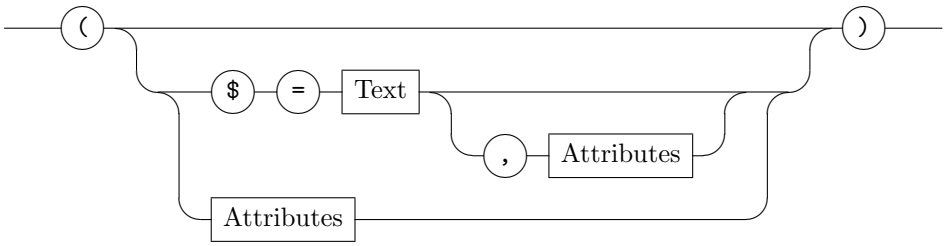


Führt einen backend-spezifischen Graphenbefehl aus. Werden keine Parameter angegeben, wird eine Liste der möglichen Befehle angezeigt.

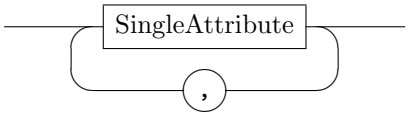
Graph-Manipulations-Befehle

Für die Definition von Graph-Elementen kann optional ein Konstruktor verwendet werden:

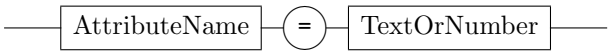
Constructor



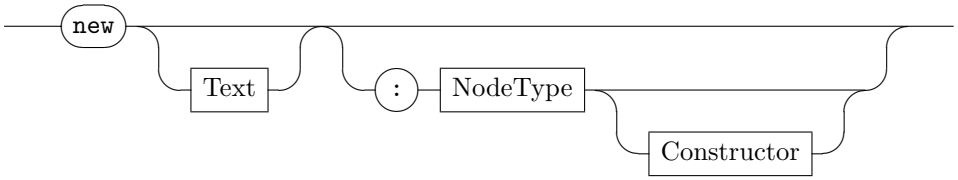
Attributes



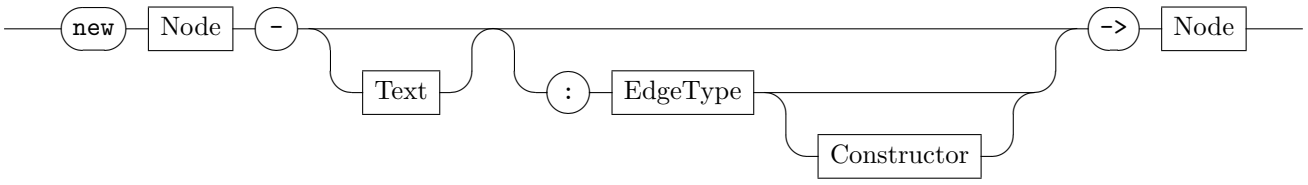
SingleAttribute



Der Konstruktor erhält also eine komma-separierte Liste von Attributdefinitionen. Nicht aufgelistete Attribute erhalten einen typspezifischen Standardwert (int und enum → 0, boolean → false, String → "").
Das „$"-Attribut hat die besondere Bedeutung eines eindeutigen Namen für das neue Element, der über die Lebenszeit des Elements mit diesem verknüpft ist. Dies wurde bei der Definition von „GraphElement" bereits als „permanenter Name" bezeichnet. Dieser Name kann nur mit Hilfe des Konstruktors gesetzt werden.



Erzeugt im aktuellen Graphen einen neuen Knoten und weist ihn optional der durch „Text" spezifizierten Variable zu.
Wird ein Knoten-Typ angegeben, so wird ein Knoten diesen Typs erzeugt und die Attribute können mittels „Constructor" initialisiert werden.
Wird kein Knoten-Typ angegeben, so wird ein Knoten vom Typ der Basis-Knoten-Klasse des aktuellen Graphmodels erzeugt.

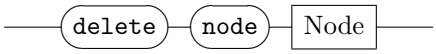


Erzeugt im aktuellen Graphen eine neue Kante von der ersten „Node" zur zweiten und weist sie optional der durch „Text" spezifizierten Variable zu.

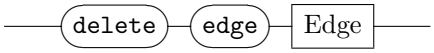
Wird ein Kanten-Typ angegeben, so wird eine Kante diesen Typs erzeugt und die Attribute können mittels „Constructor“ initialisiert werden.
Wird kein Kanten-Typ angegeben, so wird eine Kante vom Typ der Basis-Kanten-Klasse des aktuellen Graphmodels erzeugt.

Setzt den Wert des Attributes des gegebenen Graphelements auf den durch „TextOrNumber“ angegebenen Wert.

Belegt eine mit „Text“ benannte Variable mit dem gegebenen Graphelement. Wurde als Graphelement eine unbelegte Variable verwendet (z.B. „null“), so ist die mit „Text“ benannte Variable danach auch unbelegt.

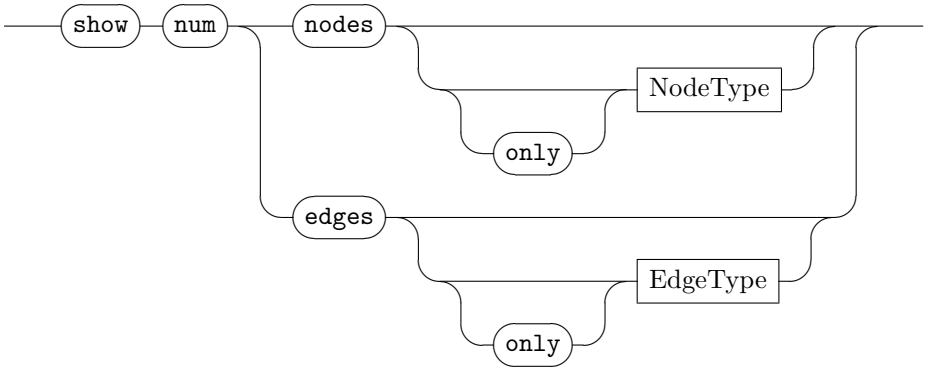


Löscht den angegebenen Knoten aus dem aktuellen Graphen. Eventuell vorhandene inzidente Kanten werden entfernt.

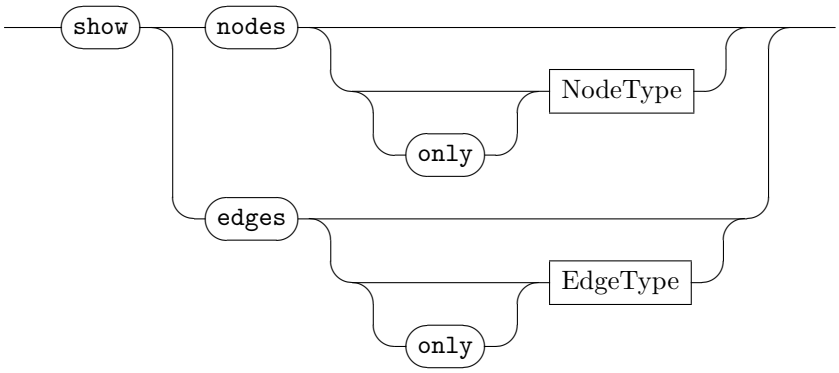


Löscht die angegebene Kante aus dem aktuellen Graphen.

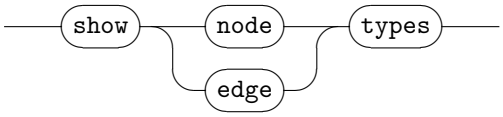
Graph-Abfrage-Befehle



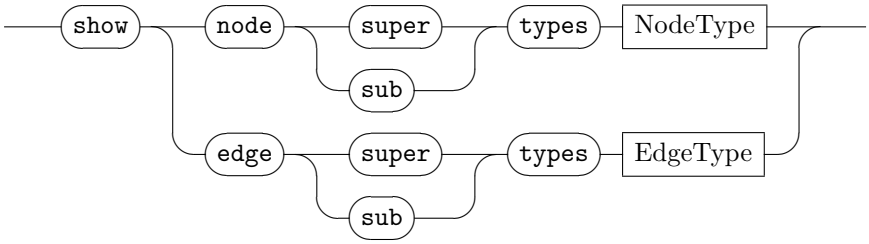
Gibt die Anzahl aller Knoten bzw. Kanten im aktuellen Graphen aus. Wird ein Typ angegeben, werden nur zu diesem Typ kompatible Elemente oder, wenn „only“ angegeben wurde, sogar nur Elemente von genau diesem Typ betrachtet.



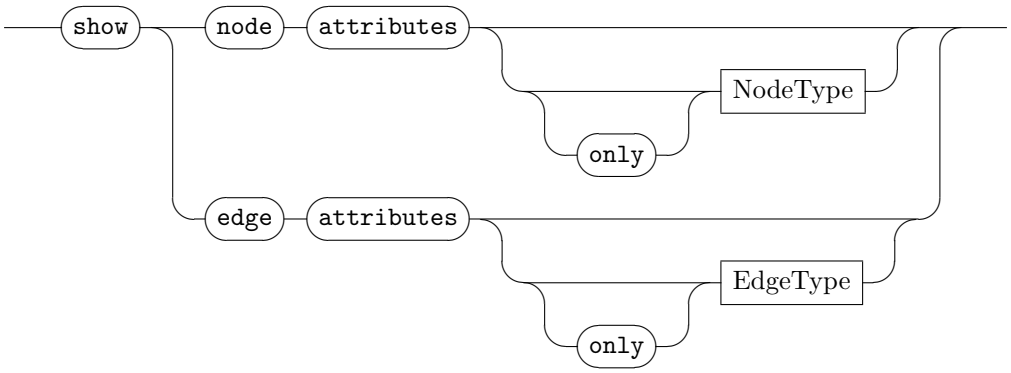
Gibt die permanenten Namen und Typen aller Knoten bzw. Kanten im aktuellen Graphen aus. Wird ein Typ angegeben, werden nur zu diesem Typ kompatible Elemente oder, wenn „only“ angegeben wurde, sogar nur Elemente von genau diesem Typ betrachtet.



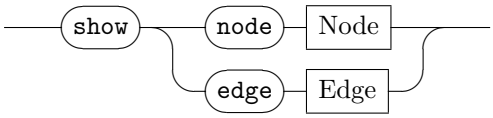
Zeigt alle Knoten- bzw. Kantentypen des aktuellen Modells an.



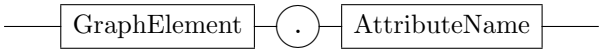
Zeigt alle Ober- bzw. Untertypen des gegebenen Knoten- bzw. Kantentyps an.



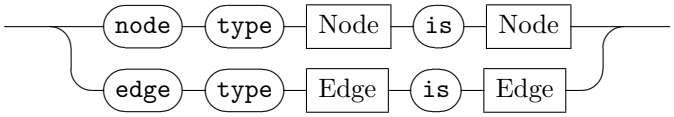
Zeigt alle Knoten- bzw. Kantenattributtypen an. Wird ein Typ angegeben, so werden nur die Attributtypen angezeigt, die ein Objekt diesen Typs besitzt. Wird zusätzlich „only“ angegeben, werden nur die Attributtypen angezeigt, die aus dem gegebenen Typen stammen.



Zeigt alle Attribute des gegebenen Elements an.

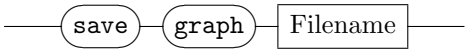


Zeigt den Wert des Attributs „AttributeName“ des gegebenen Elements an.

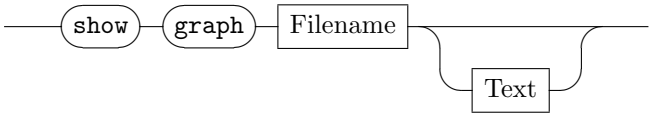


Gibt aus, ob das erste Element typkompatibel zum zweiten Element ist.

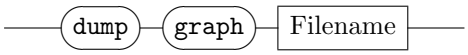
Graph-Ausgabe-Befehle



Schreibt den Graphen in Form eines GrShell-Skripts in die angegebene Datei.

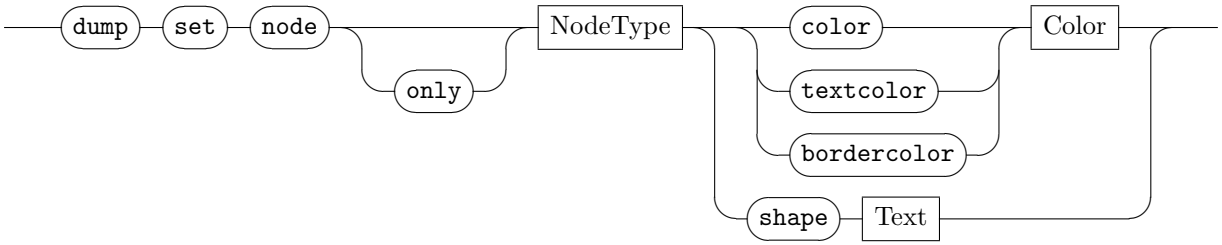


Schreibt den aktuellen Graphen im VCG-Format in eine temporäre Datei und übergibt sie dem in „Filename“ angegebenen Programm als (letzten) Parameter. Mittels des optionalen „Text“ können zusätzliche Parameter an das Programm übergeben werden.
Die temporäre Datei wird gelöscht, sobald das Programm beendet wird, sofern nicht GrShell vorher beendet wurde.

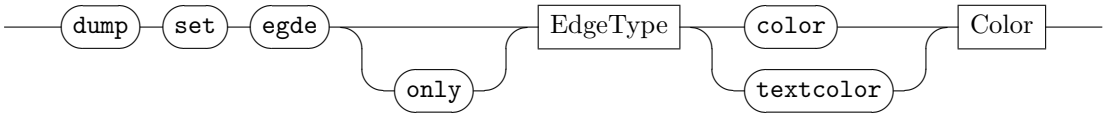


Schreibt den Graphen im VCG-Format in die angegebene Datei.

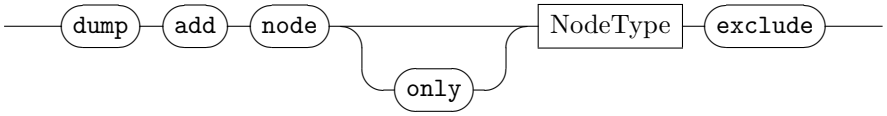
Mit den folgenden Befehlen kann die VCG-Ausgabe beeinflusst werden:



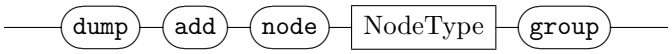
Setzt die Farbe, Textfarbe, Randfarbe bzw. Form aller Knoten des angegebenen Typs und, wenn „only“ nicht angegeben wurde, auch aller Untertypen. Folgende Formnamen werden unterstützt: box, triangle, circle, ellipse, rhomb, hexagon, trapeze, uptrapeze, lparallelogram und rparallelogram.



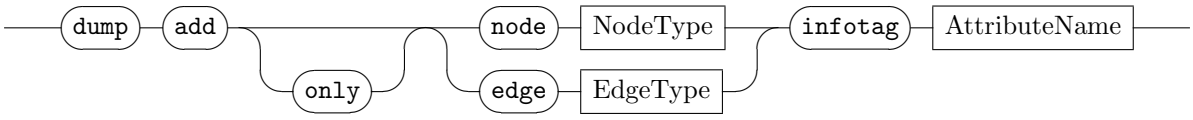
Setzt die Farbe bzw. Textfarbe aller Kanten des angegebenen Typs und, wenn „only“ nicht angegeben wurde, auch aller Untertypen.



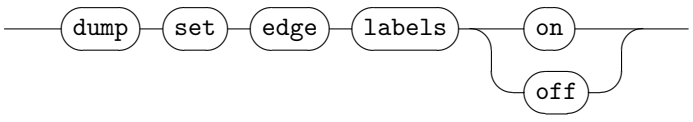
Schließt alle Knoten des angegebenen Typs und ihre inzidenten Kanten von der Ausgabe aus. Wenn „only“ nicht angegeben wird, gilt dies auch für alle Untertypen des gegebenen Typs.



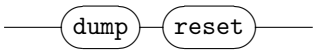
Deklariert den angegebenen Typ als neuen Gruppenknotentyp. Alle Nicht-Gruppenknoten, die auf einen Gruppenknoten verweisen, werden in dem Gruppenknoten gruppiert.



Deklariert ein Attribut des angegebenen Typs als „Infotag“. Infotags werden zusammen mit den Labels in die Knoten bzw. an die Kanten geschrieben. Wenn „only“ nicht angegeben wird, gilt dies auch für alle Untertypen des gegebenen Typs.

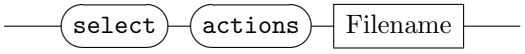


Legt fest, ob initial Kanten-Labels angezeigt werden sollen, oder nicht. (Standard ist „on“. Wird zur Zeit nicht von yComp unterstützt.)

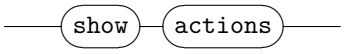


Stellt die Standardeinstellungen wieder her.

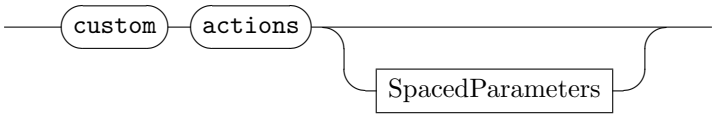
A.2.3 Actions-Befehle



Wählt einen Regelsatz aus. „Filename“ darf dabei sowohl auf eine .NET-Bibliothek (.dll) als auch eine C#-Sourcecode-Datei (.cs) verweisen, solange die Datei genau eine Unterklasse von LGSPActions enthält.



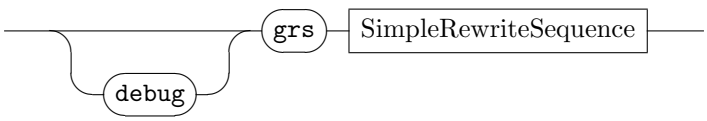
Listet alle Regeln zusammen mit ihren Parametern und Rückgabewerten auf.



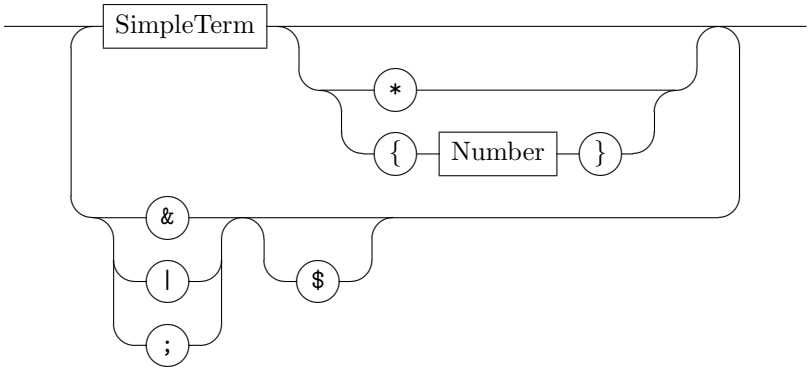
Führt einen backend-spezifischen Actionsbefehl aus. Werden keine Parameter angegeben, wird eine Liste der möglichen Befehle angezeigt.

Der Graphersetzungsbefehl

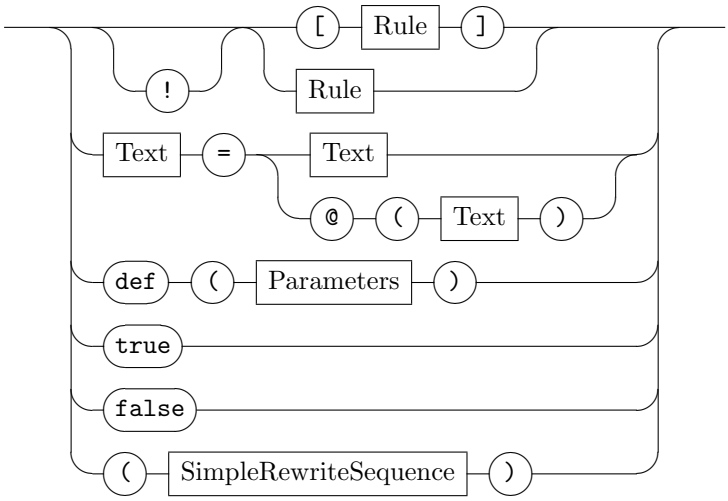
Grundsätzlich und ohne die Präzedenzen zu beachten sieht ein Graphersetzungsbefehl folgendermaßen aus:



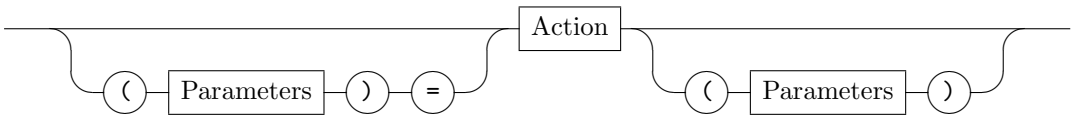
SimpleRewriteSequence



SimpleTerm



Rule



Seien s1 und s2 Graphersetzungssequenzen, v1 und v2 Variablennamen, n1 ein Graphelementname, $n \in \mathbb{N}_0$.

Rule:	Das erste gefundene Vorkommen des von der in „Rule“ spezifizierten Regel (Action) festgesetzten Muster wird ersetzt. Rule ist genau dann erfolgreich, wenn das Muster gefunden wurde.
[Rule]:	Alle Vorkommen des Musters werden ersetzt. Das Verhalten ist undefiniert, wenn bei einer Ersetzung ein anderes Vorkommen ungültig gemacht wird. [Rule] ist genau dann erfolgreich, wenn Das Muster gefunden wurde.
’!’-Regelprefix:	Die gefundenen Matches der entsprechenden Regel werden in VCG-Dateien geschrieben.
s1 & s2:	Transaktionelles Und: Zuerst wird s1 ausgeführt. Ist s1 erfolgreich, so wird s2 ausgeführt. Falls s2 nicht erfolgreich ist, werden alle Änderungen von s1 rückgängig gemacht. s1 & s2 ist genau dann erfolgreich, wenn s1 und s2 erfolgreich sind.
s1 | s2:	Exklusives Oder: Zuerst wird s1 ausgeführt. Ist s1 nicht erfolgreich, wird s2 ausgeführt. s1 | s2 ist genau dann erfolgreich, wenn s1 oder s2 erfolgreich ist.
s1 ; s2:	Konkatenation: Zuerst wird s1, dann s2 ausgeführt. s1 ; s2 ist genau dann erfolgreich, wenn s1 oder s2 erfolgreich ist.
’$’:	Hebt die Auswertungsreihenfolge des nachfolgenden Operators auf. Gleichverteilt wird die linke oder die rechte Seite des Operators als erster Teil der Sequenz aufgefasst.
s1*:	Führt s1 so oft hintereinander aus, bis s1 nicht mehr erfolgreich ist. s1* schlägt fehl, wenn s1 kein mal erfolgreich war.
s1{n}:	Führt s1 so oft hintereinander aus, bis s1 nicht mehr erfolgreich ist, maximal jedoch n-mal. s1{n} schlägt fehl, wenn s1 kein mal erfolgreich war.
def(Parameters):	Ist genau dann erfolgreich, wenn alle angegebenen Variablen belegt sind.
true:	Ist immer erfolgreich.
false:	Ist nie erfolgreich.
v1 = v2:	Der Variablen v1 wird der Wert von v2 zugewiesen. Ist v2 unbelegt (z.B. die Variable „null“), so ist v1 danach auch unbelegt.
v1 = @(n1):	Die Variable v1 wird auf das Graphelement mit dem Namen n1 gesetzt. Existiert ein solches Element nicht (mehr), so ist v1 danach unbelegt.

Dabei gilt folgende Präzedenzreihenfolge für die Verknüpfungen angefangen bei der höchsten: ’&’, ’|’, ’;’.

Bei Regeln, die Graphelemente zurückgeben, können diese Elemente optional mittels „(Parameters)=Action“ den entsprechenden Variablen in „Parameters“ zugewiesen werden. Regeln, die Graphelemente als Parameter verlangen, müssen diese auch mittels „Action(Parameters)“ in „Parameters“ übergeben werden. Werden unbelegte Variablen übergeben, wird für diese nach beliebigen, passenden Elementen gesucht. Da Parameter jedoch von der Suchplanerzeugung als „kostenlos“ angesehen werden, kann dies zu erheblichen Laufzeiteinbußen führen.

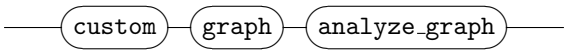
Wird „debug“ angegeben, gelangt man in den Einzelschrittmodus und kann mittels yComp [LGH+07] genau verfolgen, was während der Abarbeitung der angegebenen Ersetzungssequenz passiert. Die Ausführung kann dann in der GrShell mit diesen Befehlszeichen gesteuert werden:

’s’(tep):	Führt eine Regel komplett aus (Suchen und ersetzen).
’d’(etailed step):	Führt eine Regel komplett, aber in drei Teilen aus: Suchen, Ersetzungen markieren, Ersetzung ausführen.
’n’(ext):	Steigt in der Kantorowitsch-Baum-Darstellung der Ersetzungssequenz eine Stufe nach oben.
(step) ’o’(ut):	Lässt die Abarbeitung der Ersetzungssequenz solange weiterlaufen, bis die Scheife, in der sich die Abarbeitung befand, verlassen wurde. Befand sich die Abarbeitung in keiner Schleife, wird die komplette Sequenz abgearbeitet.
’r’(un):	Setzt die Abarbeitung bis zum Schluss fort.
’a’(bort):	Bricht die Abarbeitung der Ersetzungssequenz ab.

A.3 LGSP-Backend-Befehle

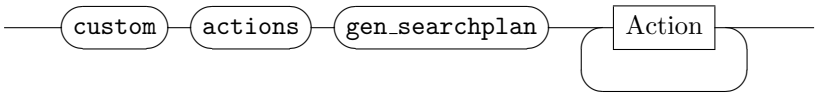
Das LGSP-Backend unterstützt folgende „custom“ Befehle:

A.3.1 Graphen-Befehle

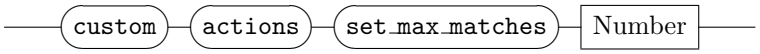


Analysiert den aktuellen Arbeitsgraphen. Diese Analysedaten stellen für die Erzeugung von Suchplänen wichtige Informationen zur Verfügung.

A.3.2 Actions-Befehle



Erzeugt anhand der zuletzt berechneten Analysedaten mit Hilfe einer Heuristik jeweils einen möglichst guten Suchplan für die angegebenen Regeln.



Setzt die maximale Anzahl der zu findenen Vorkommen von Mustern bei der Verwendung von [Rule] auf den gegebenen Wert. Ist der Wert kleiner oder gleich null, so wird nach allen Vorkommen gesucht.

Literaturverzeichnis

[Bat06] Batz, Gernot V.: An Optimization Technique for Subgraph Matching Strategies / Universität Karlsruhe, IPD Goos. Version: April 2006. http://www.info.uni-karlsruhe.de/papers/TR_2006_7.pdf. 2006 (2006-7). – Forschungsbericht

[CL65] Chu, Y.J. ; Liu, T.H.: On the shortest arborescence of a directed graph. In: *Science Sinica* 14 (1965), S. 1396–1400

[ERT99] Ermel, C. ; Rudolf, M. ; Taentzer, G.: The AGG Approach: Language and Environment. In: *[Roz99]* Bd. 2. 1999, S. 551–603

[Fuj06] Fujaba Developer Team: *Fujaba-Homepage*. http://www.fujaba.de/. Version: August 2006

[GBG+06] Geiss, Rubino ; Batz, Veit ; Grund, Daniel ; Hack, Sebastian ; Szalkowski, Adam M.: GrGen: A Fast SPO-Based Graph Rewriting Tool. In: Corradini, A. (Hrsg.) ; Ehrig, H. (Hrsg.) ; Montanari, U. (Hrsg.) ; Ribeiro, L. (Hrsg.) ; Rozenberg, G. (Hrsg.): *Graph Transformations - ICGT 2006*, Springer, 2006 (Lecture Notes in Computer Science), 383 – 397. – Natal, Brasilia

[Hac03] Hack, Sebastian: *Graphersetzung für Optimierungen in der Codeerzeugung*, Universität Karlsruhe, Diplomarbeit, 2003

[LGH+07] Leiss, Philipp ; Geiss, Rubino ; Hack, Sebastian ; Beck, Michael ; Kroll, Moritz: *yComp-Homepage*. http://www.info.uni-karlsruhe.de/software.php/id=6. Version: April 2007

[MB00] Marxen, Heiner ; Buntrock, Jürgen: *Old list of record TMs*. http://www.drb.insel.de/ heiner/BB/index.html. http://www.drb.insel.de/~heiner/BB/index.html. Version: August 2000

[Roz99] Rozenberg, G. (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, 1999

[Sch99] Schürr, A.: The PROGRES Approach: Language and Environment. In: *[Roz99]* Bd. 2. 1999, S. 487–550

[Var05] Varró, G.: *Graph transformation benchmarks page*. http://www.cs.bme.hu/~gervarro/benchmark/2.0/. Version: August 2005

[VFV04] Varró, G. ; Friedl, K. ; Varró, D.: Graph Transformations in Relational Databases. In: *Proc. GraBaTs 2004: Intl. Workshop on Graph Based Tools*, Elsevier, 2004

[VSV05] Varró, G. ; Schürr, A. ; Varró, D.: Benchmarking for Graph Transformation / Department of Computer Science and Information Theory, Budapest University of Technology and Economics. 2005. – Forschungsbericht

Abbildungsverzeichnis