Hochschule
Bonn-Rhein-Sieg
University
of Applied Sciences

Fachbereich Informatik
Department of Computer Science

Abschlussarbeit

im Studiengang Bachelor Informatik

Matrixbasierte Darstellung und Verarbeitung von Daten- und Wissensstrukturen

von

Andreas Schmitz

Erstprüfer	Prof. Dr. Martin E. Müller
Zweitprüfer	Prof. Dr. Alexander Asteroth

Eingereicht am

Eidesstattliche Erklärung

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

(Datum, Unterschrift)

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Vorwort

Diese Arbeit wurde im Rahmen des Bachelor-Studiengangs Informatik an der Hochschule Bonn-Rhein-Sieg verfasst.

An dieser Stelle möchte ich allen Personen danken, die mich bei der Erstellung dieser Arbeit und während des Studiums begleitet und unterstützt haben.

Mein besonderer Dank gilt meinem Erstprüfer, Herrn Prof. Martin E. Müller, für die gute Betreuung, die motivierenden Gespräche und die zahlreichen Anmerkungen und Ideen.

Ebenso gilt mein besonderer Dank meinem Zweitprüfer, Herrn Prof. Alexander Asteroth, für die schnelle und unkomplizierte Zusage und das Interesse an dieser Arbeit.

Weiterhin danke ich Peter Berger, ohne dessen Vorarbeiten diese Arbeit nicht möglich gewesen wäre sowie Boris Busche und Constantin Kirsch, die mir helfend beiseite Standen und immer ein offenes Ohr für mich hatten.

Ich danke allen Korrekturlesern für die gefundenen Fehler und die guten Ratschläge.

Des Weiteren danke ich meiner Familie und meinen Freunden, die mich auch an den stressigen Tagen ertragen konnten und mir Rückhalt gaben.

Mein besonderer Dank gilt meinen Eltern, die mich immer unterstützt haben und mir das Studium erst ermöglichten.

Kapitel 2

Einführung

2.1 Motivation und Problemstellung

Relationen stellen ein interessantes und vielseitiges Forschungsobjekt dar. Doch leider offenbaren sich Relationen nicht immer sehr verständlich, was vor allem an der sehr abstrakten Beschreibungssprache liegt. Die Erfassung der wesentlicher Strukturen und Beziehungen zwischen den Elementen einer Relation ist in der Regel nicht sehr einfach. Vor allem für Anfänger auf diesem Gebiet ist der Umgang mit Relationen nicht trivial.

Alternative Darstellungen von Relationen, wie Graphen oder boolesche Matrizen (siehe Abschnitt 3.1.2), beheben die Verständnis- und Darstellungsprobleme teilweise, führen aber zu einem nicht unerheblichen Erstellungsaufwand; erst recht mit Papier und Stift. In den Zeiten von günstigen und schnellen Computern ist es heute allerdings möglich beide Ansätze zu kombinieren und aus der abstrakten Beschreibung einer Relation eine verständliche Darstellung zu generieren.

Hier setzte Peter Berger 2011 mit seiner Bachelor-Abschlussarbeit an, indem er die Programmierbibliothek RelaFix sowie ein Kommandozeilen-Tool entwickelte, welche, durch Verwendung einer eigens entwickelten Sprache, die der mathematischen Beschreibung von Relationen sehr nahe kommt, Relationen erzeugen und verarbeiten kann. In Abschnitt 3.2 wird RelaFix genauer betrachtet. Ein erster Schritt war damit getan, doch fehlt es vor allem dem Kommandozeilen-Tool an Einfachheit und Benutzerfreundlichkeit.

Weiterhin gestaltet sich das Verändern und die Verarbeitung von Relationen mit herkömmlichen Mitteln sehr schwer. Problematisch scheint in diesem Zusammenhang vor allem die Frage, um welche Elemente eine Relation noch erweitert oder erleichtert werden muss, um eine bestimmte Eigenschaft zu erreichen. Das Lösen der Problematik würde es beispielsweise Erlauben, durch Rauschen verfälschte Relationen entzerren zu lassen. Zudem können anhand bestimmter Darstellungsformen, z.B. Graphen und boolesche Matrizen, relativ einfach Eigenschaften abgelesen werden. Diese Strukturen sind unter

Umständen nicht sichtbar, und können erst durch Umordnen der Elemente, z.B. Graphenknoten oder Matrix Zeilen und Spalten, offenbart werden. Auch hier könnten geeignete Algorithmen Abhilfe schaffen.

Zwar existieren zur Zeit schon Applikationen, die die genannten Probleme teilweise oder komplett lösen, z.B. die Applikation RELVIEW von Rudolf Berghammer (Behnke u. a., 1998), aber handelt es sich bei diesen in der Regel um schwer zu bedienende, komplexe Applikationen, die größtenteils weder Benutzer- noch Einsteigerfreundlichkeit sind oder nur für stolze Preise erworben werden können, wie z.B. Mathematica. Die hier aufklaffende Lücke war die wesentliche Motivation für die Entwicklung von RelaView.

2.2 Ziele der Arbeit

Ziel der vorliegenden Arbeit stellte die Konzeptionierung und Entwicklung einer freien, plattformübergreifenden Software zur Darstellung und Verarbeitung von Relationen dar, welche RelaView getauft wurde.

Im Laufe der Vorbereitungen wurden vier Schwerpunkte im Bezug auf die Relationen bestimmt, welche zu implementieren waren. Bei diesen handelt es sich um das Darstellen, Ändern, Vervollständigen und Sortieren von Relationen. Eine ausführliche Darstellung der kompletten Anforderungen findet sich in Abschnitt 4.

Besonders entscheidend war es, die bereits erwähnte Benutzer- und Einsteigerfreundlichkeit einer solchen Software zu erreichen und gleichermaßen einen ausreichend großen Funktionsumfang zu erzielen. Dazu gehörte vor allem die Möglichkeit der leichten und schnellen visuellen Interpretation der vorliegenden Informationen, weshalb für die Umsetzung einer solchen Software eine grafische Benutzeroberfläche als zwingend erforderlich galt.

Um den Nutzer nicht aus seiner gewohnten Arbeitswelt herauszuziehen, sollte die Applikation einen möglichst nativen Eindruck erwecken, und sich an die Gegebenheiten des verwendeten Betriebssystems anpassen.

Als essentielles Ziel galt es, die Software so zu entwickeln, dass diese sowohl in der Lehre als auch in der Forschung eingesetzt werden kann. Wie bereits in Abschnitt 2.1 dargelegt, ist der Umgang mit Relationen für Anfänger nicht immer leicht. Diesbezüglich sollte bei der Entwicklung darauf geachtet werden, eine Komplexität zu erreichen, die Studierenden einen einfachen Einstieg ermöglicht. Die Software sollte zudem in der Lage sein Studierende in ihrem Lernprozess zu unterstützen.

Weiterhin sollte die Software aber auch einen Umfang erreichen, der erfahrenen Anwendern und Experten einen Mehrwert bietet.

Kapitel 3

Einleitung

3.1 Mathematische Grundlagen

In diesem Abschnitt werden kurz Fakten zu Mengen und Relationen wiederholt.

3.1.1 Mengen

Im Rahmen dieser Arbeit sind lediglich endliche Mengen von Relevanz, weshalb sich bei den Definitionen und Erklärungen auf diese beschränkt wird.

Eine endliche Menge besitzt eine Bezeichnung sowie eine endliche Anzahl untereinander verschiedener Elemente - wie anhand folgender Beispiele zu erkennen. Eine Menge, die keine Elemente enthält wird mit $\emptyset$ bezeichnet und leere Menge genannt.

$$A = \{1, 2, 3, 4, 5, 6, 7\}$$
$$Vorlesungen = \{Mathematik, Programmieren, Recht\}$$
$$Konzerne = \{Apple, Google, HP, Microsoft\}$$

Als Teilmengen bezeichnet man Mengen, die einen Teil oder alle Elemente einer Menge enthalten. Dies wird durch das Teilmengensymbol $\subseteq$ beschrieben. Enthält eine Menge nur einen Teil der Elemente einer Menge, wird diese als echte Teilmenge bezeichnet. Für eine echte Teilmenge wird das Symbol $\subset$ verwendet. Um auszudrücken, dass es sich um keine Teilmenge handelt wird $\not\subset$ oder $\not\subseteq$ verwendet.

$$A = \{1, 2, 3, 4, 5, 6, 7\} \quad B = \{1, 2, 5\} \quad C = \{1, 2, 5, 8\}$$
$$B \subseteq A, \text{ sogar } B \subset A$$
$$C \not\subseteq A, B \subseteq C$$
$$A \subseteq A, \text{ aber } A \not\subset A$$

Operationen auf Mengen

Im Folgenden wird eine beschränkte Anzahl spezieller Mengen eingeführt. Die Beschränkung rührt daher, dass es genau diese Mengen sind, die im Rahmen dieser Arbeit relevant sind und in RelaFix / RelaView implementiert wurden. Da in RelaView eben jene spezielle Mengen aus einem Prozess bzw. Algorithmus entstehen, wird im Weiteren auch die Rede von Operationen sein. Die Operationen auf die sich hier beschränkt wird, sind:

- Kartesische Produkt
- Durchschnitt
- Vereinigung
- Differenz
- Komplement

Die Operationen kartesisches Produkt, Durchschnitt und Vereinigung werden hier als binäre Operationen eingeführt, können aber auch für eine beliebige Anzahl von Mengen definiert werden.

Kartesische Produkt $\times$

Das kartesische Produkt zweier Mengen ist die Kombination jedes Elementes einer Menge mit jedem der Anderen. Diese Kombination ist in einem 2-Tupel zusammengefasst, welches die Form (x, y) hat. Formal definiert ist das kartesische Produkt $\times$ wie folgt (Witt, 2010, S. 88):

$$A \times B = \{(a, b) \mid \forall a \in A, \forall b \in B\}$$

Durchschnitt $\cap$

Der Durchschnitt zweier Mengen ist definiert als die Menge der gemeinsamen Elemente dieser. Diese Menge wird auch als Schnittmenge $\cap$ bezeichnet (Witt, 2010, S. 65).

$$A \cap B = \{x \mid x \in A \wedge x \in B\}$$

Des Weiteren werden zwei Mengen, deren Schnittmenge die leere Menge ist, als disjunkt bezeichnet, oder auch elementfremd (vgl. Witt, 2010, S. 65).

Vereinigung $\cup$

Die Vereinigung zweier Mengen ist definiert als die Menge aller Elemente beider Mengen. Die Menge wird auch als Vereinigungsmenge $\cup$ bezeichnet (Witt, 2010, S. 65).

$$A \cup B = \{x \mid x \in A \vee x \in B\}$$

Differenz $\setminus$

Die Differenz $A \setminus B$ der Mengen A und B beinhaltet alle Elemente von A, außer denen, die auch in B enthalten sind (Witt, 2010, S. 66).

$$A \setminus B = \{x \mid x \in A \wedge x \notin B\}$$

Komplement $\overline{A}$

Das Komplement einer Menge A, mit $A \subseteq B$ ist definiert als die Differenz von B und A. Das Komplement von A enthält also alle Elemente aus B, außer denen, die in A enthalten sind (Witt, 2010, S. 66).

$$\overline{A} = B \setminus A = \{x \mid x \in B \wedge x \notin A\}, \text{ für } A \subseteq B$$

3.1.2 Relationen

Der Begriff der Relation stellt ein Verhältnis zwischen Mengen dar. Eine n-stellige Relation R ist eine Teilmenge des kartesischen Produkts der zugrundeliegenden Mengen $A_1, ..., A_n$. Die Relation stellt somit auch wieder eine Menge dar, auf welche die im vorherigen Abschnitt eingeführten Operationen analog angewandt werden können (Witt, 2010, S. 88-89).

$$R \subseteq A_1 \times ... \times A_n$$

bzw.

$$R \subseteq \{(x_1, ..., x_n) \mid x_i \in A_i, 0 \leq i \leq n\}$$

Im Rahmen dieser Arbeit sind vor allem binäre Relationen relevant, weshalb der Begriff Relation fortan, wenn nicht anders vereinbart, binäre Relationen meint. Definiert sind sie wie folgt:

$$R \subseteq \{(a, b) \mid a \in A, b \in B\}$$

Menge A wird als Ausgangsmenge und Menge B als Zielmenge bezeichnet.

Um darzustellen, dass ein Tupel Teil einer Relation ist - also ein Element der Ausgangsmenge mit einem der Zielmenge in Relation steht - wird neben der bekannten Schreibweise

$$(a, b) \in R, a \in A, b \in B$$

der Einfachheit die Infixschreibweise

$$aRb$$

verwendet (vgl. Witt, 2010, S. 90).

Weiterhin werden binäre Relationen, bei denen Ausgangs- und Zielmenge gleich sind, als homogen bezeichnet. Alle Relationen, die nicht homogen sind, heißen inhomogen.

Darstellung binärer Relationen

Im Weiteren wird ein kleines Beispiel (vgl. Witt, 2010, S. 89-90) verwendet, um die Anwendung der Definitionen zu zeigen sowie verschiedene Darstellungsarten von Relationen zu definieren, welche vor allem für spätere Kapitel bedeutend sind.

Es sei gegeben die homogene Relation R:

$$A = \{-3, -2, -1, 0, 1, 2, 3\}$$
$$R \subseteq A \times A, \text{ mit } R = \{(a, b) \in A \times A \mid a \cdot b > 2\}$$

Obwohl die dargelegte Definition bereits vollständig ausreicht, um R eindeutig zu beschreiben, mangelt es ihr an Anschaulichkeit, wie bereits in 2.1 erläutert.

Alternativ, und viel anschaulicher, präsentiert sich folgende Darstellung, welche alle 2-Tupel der Relation auflistet.

$$R = \{(-3, -3), (-3, -2), (-3, -1), (-2, -3), (-2, -2),$$
$$(-1, -3), (1, 3), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)\}$$

Nun sind alle enthaltenen Elemente der Relation direkt sichtbar, und müssen nicht erst „berechnet“ werden. Trotzdem mangelt es der Darstellung immer noch an Anschaulichkeit, und eventuelle Eigenschaften sind nicht direkt erkennbar. Für diesen Zweck besser geeignet ist eine Darstellung als Graph (vgl. Schmidt, 2010, S. 19 ff.), wie in Abbildung 3.1 zu sehen.

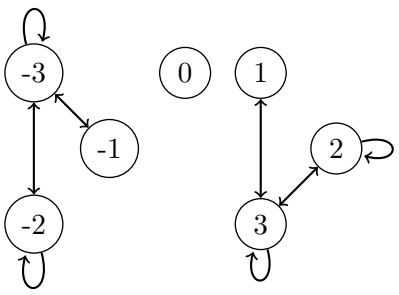

Abbildung 3.1: Graph Darstellung einer Relation

Die letzte hier aufgelistete Darstellungsart, welche zudem die für diese Arbeit wichtigste darstellt, ist die Matrix- oder Tabellendarstellung, auch boolesche Matrix genannt. Die boolesche Matrix, für Beispielrelation R, ist in Abbildung 3.2 dargestellt.

R	-3	-2	-1	0	1	2	3
-3	1	1	1	0	0	0	0
-2	1	1	0	0	0	0	0
-1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
2	0	0	0	0	0	1	1
3	0	0	0	0	1	1	1

Abbildung 3.2: Matrix Darstellung der Beispielrelation (Witt, 2010, S. 90)

Diese Darstellung hat den entscheidenden Vorteil, dass sich anhand der Einsen - man kann sich auch die Einsen durch einen schwarzen Block und die Nullen durch einen leeren Kasten substituiert vorstellen - nicht nur sofort alle in Relation stehende Elemente ablesen lassen, sondern bei bestimmten Matrixstrukturen auch sofort Eigenschaften der zugrundeliegenden Relation bestimmt werden können. So z.B. ist eine symmetrische Relation dadurch zu erkennen, dass die Matrix an ihrer Diagonalen spiegelsymmetrisch ist. Um ein Matrixelement anzugeben wird im Folgenden (x, y) für den Wert in Spalte x und Zeile y benutzt, welcher 1, für $(x, y) \in R$, oder 0, für $(x, y) \notin R$, sein kann.

Spezielle Relationen

An dieser Stelle sind einige spezielle Relationen hervorzuheben, welche sich durch besondere Strukturen auszeichnen und deshalb eine eigene Bezeichnung besitzen.

Die Nullrelation sowie die universelle Relation zeichnen sich, wie die Namen schon vermuten lassen, dadurch aus, dass diese keine bzw. alle Elemente des kartesischen Produktes enthalten (Schmidt, 2010, S. 37). In der Matrix Darstellung äußert sich das demzufolge dadurch, dass alle Matrixelemente der Nullrelation eine 0 und alle Matrixelemente der universellen Relation eine 1 enthalten.

$$R = \{\} \qquad \text{„Nullrelation“}$$
$$R = A \times B \qquad \text{„universelle Relation“}$$

Als weitere besondere Relation ist die Identitätsrelation zu nennen (vgl. Schmidt, 2010, S. 37). Bei dieser handelt es sich um eine homogene Relation, bei der alle Elemente mit sich selbst in Relation stehen - und auch nur diese.

$$R = \{(a, a) \mid a \in A\} \qquad \text{„Identitätsrelation“}$$

Die Identitätsrelation ist dadurch zu erkennen, dass ihre boolesche Matrix eine Diagonale aus Einsen von links oben nach rechts unten enthält und die restlichen Matrixelemente 0 sind, wie in Abbildung 3.3 zu erkennen.

R	a	b	c	d	e	f	g
a	1	0	0	0	0	0	0
b	0	1	0	0	0	0	0
c	0	0	1	0	0	0	0
d	0	0	0	1	0	0	0
e	0	0	0	0	1	0	0
f	0	0	0	0	0	1	0
g	0	0	0	0	0	0	1

Abbildung 3.3: Matrix Darstellung der Identitätsrelation

Operationen auf Relationen

Neben den bereits von Mengen bekannten Operationen auf Relationen, existieren weitere, relationsspezifische Operationen, von denen im Folgenden die Umkehrrelation sowie die Verkettung vorgestellt werden.

Umkehrrelation R^{-1}

Die Umkehrrelation R^{-1}, oder auch Inverse Relation, einer Relation $R \subseteq A \times B$ ist wie folgt definiert (vgl. Witt, 2010, S. 96-97):

$$R^{-1} = \{(b, a) \in B \times A \mid (a, b) \in R\}$$

bzw.

$$yR^{-1}x \text{ gdw. } xRy$$

Damit präsentiert sich die Umkehrrelation als die an der Diagonalen gespiegelte Relation. Was zudem bedeutet, dass die Umkehrrelation einer symmetrischen Relation gleich der eigentlichen Relation ist.

Verkettung $\circ$

Die Verkettung $R \circ S \subseteq A \times C$, oder auch Komposition, zweier binärer Relationen $R \subseteq A \times B$ und $S \subseteq B \times C$ ist wie folgt definiert (vgl. Witt, 2010, S. 97):

$$R \circ S = \{(a, c) \in A \times C \mid \exists b \in B : aRb \wedge bSc\}$$

Eigenschaft	Definition
Reflexivität	aRa
Irreflexivität	$aRb \rightarrow a \neq b$ bzw. $\forall a : \neg aRa$
Symmetrie	$aRb \rightarrow aRb$
Asymmetrie	$aRb \rightarrow \neg aRb$
Antisymmetrie	$aRb \wedge bRa \rightarrow a = b$
Transitivität	$aRb \wedge bRc \rightarrow aRc$
Difunktionalität	$aRb \wedge cRb \wedge cRw \rightarrow aRw$

Abbildung 3.4: Homogene Eigenschaften binärer Relationen

Homogene Eigenschaften binärer Relationen

Im Rahmen dieser Arbeit spielen weiterhin lediglich homogene Eigenschaften binärer Relationen eine Rolle, da in RelaFix zur Zeit nur homogene Eigenschaften implementiert sind. Deshalb wird an dieser Stelle nur darauf verwiesen, dass es zahlreiche weitere inhomogene als auch homogene Eigenschaften gibt, die keinesfalls unbdeutend sind, nur weil sie hier nicht aufgelistet wurden. Die hier eingeführten Eigenschaften sind in Abbildung 3.4 zu erkennen. Definition nach: (Müller, 2012, S. 22) und (Berghammer, 2008, S. 212) für $R \subseteq A \times A$, mit $\forall a, b, c, w \in A$.

Zusammengesetzte Eigenschaften

Aus den sechs eingeführten homogenen Eigenschaften lassen sich spezielle Relationen gruppieren. Hier eingeführt werden die Äquivalenzrelation, Halbordnung und Quasiordnung. In Abbildung 3.5 wird dargestellt, wie sich diese zusammensetzen.

	Äquivalenz-relation	Halbordnung	Quasiordnung
Reflexiv	X	X	X
Irreflexiv			
Symmetrisch	X		
Asymmetrisch			
Antisymmetrisch		X	
Transitiv	X	X	X
Difunktional			

Abbildung 3.5: Zusammengesetzte Eigenschaften

3.1.3 Äquivalenzrelationen

Als besondere Relation soll hier die Äquivalenzrelation herausgehoben werden. Wie anhand von Abbildunng 3.5 zu erkennen, ist eine Äquivalenzrelation

reflexiv, symmetrisch und transitiv. Diese drei Eigenschaften führen dazu, dass eine Äquivalenzrelation eine Partition auf der zugrundeliegenden Menge festlegt. Das bedeutet, dass eine Äquivalenzrelation die komplette Menge in disjunkte Teilmengen unterteilt. Jede dieser Mengen wird als Äquivalenzklasse bezeichnet (vgl. Witt, 2010, S. 93 ff.).

Die Matrix einer Äquivalenzrelation lässt sich durch Tauschen von Zeilen und Spalten - was nur die Darstellung, nicht aber die Relation, verändert - immer in eine Blockform überführen. In Blockform vorliegend, bildet die Matrix, von oben links nach unten rechts, diagonal angeordnete Quadrate aus, wobei jedes Quadrat eine Äquivalenzklasse repräsentiert. Ein Beispiel für eine Blockform ist in Abbildung 3.6 zu erkennen.

R	a	b	c	d	e	f	g
a	1	1	1	0	0	0	0
b	1	1	1	0	0	0	0
c	1	1	1	0	0	0	0
d	0	0	0	1	1	0	0
e	0	0	0	1	1	0	0
f	0	0	0	0	0	1	1
g	0	0	0	0	0	1	1

Abbildung 3.6: Matrix Darstellung einer Äquivalenzrelation in Blockform

Sortieren von Äquivalenzrelationen

Wie bereits im Abschnitt zuvor dargestellt, hat jede Äquivalenzrelation eine Matrix in Blockform - auch wenn diese erst durch Tauschen von Zeilen und Spalten zum Vorschein kommen kann. Abbildung 3.7 zeigt ein Beispiel für eine „ungeordnete“ Matrix einer Äquivalenzrelation.

R	a	b	c	d	e	f	g
a	1	0	0	1	0	0	1
b	0	1	0	0	0	0	0
c	0	0	1	0	1	0	0
d	1	0	0	1	0	0	1
e	0	0	1	0	1	0	0
f	0	0	0	0	0	1	0
g	1	0	0	1	0	0	1

Abbildung 3.7: Matrix Darstellung einer ungeordneten Äquivalenzrelation

Durch einfaches Tauschen von Zeilen und Spalten, kann die Matrix aus Abbildung 3.7 in eine Matrix in Blockform überführt werden, hier in Abbildung 3.8 dargestellt. Das hier angewandte Sortierverfahren wird später, in Abschnitt 5.6, vorgestellt.

R	a	d	g	b	e	c	f
a	1	1	1	0	0	0	0
d	1	1	1	0	0	0	0
g	1	1	1	0	0	0	0
b	0	0	0	1	0	0	0
e	0	0	0	0	1	1	0
c	0	0	0	0	1	1	0
f	0	0	0	0	0	0	1

Abbildung 3.8: Matrix Darstellung einer ungeordneten Äquivalenzrelation

3.2 RelaFix

RelaFix wurde 2012 von Peter Berger im Rahmen seiner Bachelor-Abschlussarbeit an der Hochschule Bonn-Rhein-Sieg, unter der Betreuung von Prof. Dr. Martin E. Müller, konzipiert und entwickelt. Nachzuschlagen in (Berger, 2012). RelaFix ist komplett in der Programmiersprache C entwickelt worden.

RelaFix ist eine C-Bibliothek, die das Erzeugen und Verarbeiten von Relationen erlaubt. RelaFix besitzt einen großen Umfang an Möglichkeiten Relationen zu erstellen, zu verändern oder auf ihre Eigenschaften hin zu überprüfen. Zusätzlich bietet RelaFix noch die Möglichkeit Operationen auf Relationen anzuwenden. Der Umfang umfasst alle Eigenschaften und Operationen, die in Abschnitt 3.1 eingeführt wurden. Ein weiterer wesentlicher Bestandteil von RelaFix ist ein Parser, welcher es erlaubt, anhand einer eigens durch Peter Berger entwickelten Sprache, RelaFix zu steuern. Dadurch wird es möglich Relationen in einfachen Textdateien zu beschreiben und einzulesen - sogar eine interaktive Eingabe über ein Kommandozeilen-Tool ist möglich. Das Kommandozeilen-Tool wird im Zusammenhang mit der Anbindung von RelaFix an RelaView, in Abschnitt 5.2, genauer erläutert.

In Abbildung 3.9 findet sich ein Beispiel für die Syntax von RelaFix. In diesem speziellen Beispiel wird eine Äquivalenzrelation dargestellt, welche durch eine boolesche Matrix kodiert wurde. Wie zu erkennen, ähnelt die Kodierung der Relation sehr stark der in Abschnitt 3.1.2 eingeführten Matrixdarstellung. Weiterhin besteht noch die Möglichkeit Operationen anzuwenden und Relationen durch Formeln auszudrücken, wie aus Abschnitt 3.1.2 bekannt. Intern rechnet RelaFix für eine Formel dann erneut eine boolesche Matrix aus, die für eine Darstellung verwendet werden kann.

3.3 Das Qt Framework

Die Basis der im Rahmen dieser Arbeit entwickelten Software bildet das Qt (Ausgesprochen: Cutie) Framework, in der Version 4.8. Qt ist ein in C++ entwickeltes, plattformübergreifendes Anwendungsframework, dessen

```
DOMAIN domain AS {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
RELATION equ: domain -> domain AS TABLE(
     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16,
0:   1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0,
1:   0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0,
2:   0  0  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0,
3:   0  0  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0,
4:   0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0,
5:   0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0,
6:   0  0  0  0  0  0  1  1  1  0  0  0  0  0  0  0  0,
7:   0  0  0  0  0  0  1  1  1  0  0  0  0  0  0  0  0,
8:   0  0  0  0  0  0  1  1  1  0  0  0  0  0  0  0  0,
9:   0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0,
10:  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0,
11:  0  0  0  0  0  0  0  0  0  0  0  1  1  1  1  0  0,
12:  0  0  0  0  0  0  0  0  0  0  0  1  1  1  1  0  0,
13:  0  0  0  0  0  0  0  0  0  0  0  1  1  1  1  0  0,
14:  0  0  0  0  0  0  0  0  0  0  0  1  1  1  1  0  0,
15:  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1,
16:  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1
);
equ;
```

Abbildung 3.9: RelaFix Beispiel: Äquivalenzrelation in Tabellenform

Hauptanwendung das Erstellen von Programmen mit grafischer Benutzeroberfläche ist. Es ist unter: http://qt.nokia.com zu finden. Aber auch Programme ohne grafische Benutzeroberfläche lassen sich erstellen. So basiert eine Vielzahl heutiger, moderner Programme auf dem Qt Framework. Einige bekannte Beispiele sind die Desktopumgebung KDE, der Media Player VLC oder Skype.

Der große Vorteil von Qt gegenüber anderen Frameworks ist, dass Qt native API Funktionen für die Erzeugung der Benutzeroberfläche nutzt und nicht das Aussehen und Verhalten eines Betriebssystems emuliert. Eine Qt Applikation profitiert somit von den Vorteilen nativer Applikationen und passt sich zu großen Teilen in die bekannte Benutzerumgebung ein. Weiterhin unterstützt Qt auch zahlreiche Eigenheiten der verschiedenen Betriebssysteme, welche ausführlich in der Dokumentation (Nokia Corporation, 2012a) beschrieben sind.

Qt ist frei und quelloffen und wird unter Anderem unter der Lesser General Public License (LGPL) vertrieben. Zur Zeit unterstützt Qt einen großen Teil der modernen Betriebssysteme, wie Microsoft Windows, Linux / BSD und Mac OS X. Auch einige mobile und embedded Betriebssysteme werden unterstützt. Da Qt ein essentieller Bestandteil von RelaView ist, ist an die-

ser Stelle eine genauere Betrachtung von Qt angemessen. Deshalb werden im Folgenden wesentliche Bestandteile des Qt Frameworks genauer beleuchtet.

3.3.1 Das Qt Build-System

Qt erweitert C++ um zahlreiche Funktionen. Eine Spracherweiterung von C++ führt dazu, dass die bereits erhältlichen Compiler den Quellcode nicht mehr kompilieren können. Um diesen Nachteil auszugleichen wurde der MOC (Meta Object Compiler) entwickelt. Der MOC erzeugt aus dem Qt C++ Quellcode Standard C++ Code, der mit jedem normalen C++ Compiler kompiliert werden kann. Beispiele finden sich in den folgenden Abschnitten.

Der aus C++ generierte Maschinencode ist plattformabhängig. Er kann also nicht einmal generiert und auf allen Plattformen ausgeführt werden wie beispielsweise Java. Das bedeutet, dass jedes mit Qt erstellte Programm für jedes Zielsystem separat kompiliert werden muss. Um das Erstellen und Verteilen von Qt Programmen zu erleichtern, wurde QMake entwickelt. QMake erzeugt aus einer QMake Projektdatei (.pro) - was eine Art Makefile darstellt - ein plattformabhängiges Makefile, welches zum Kompilieren des Quellcodes verwendet werden kann. In einer QMake Projektdatei sind alle Informationen enthalten, die benötigt werden, um das Programm zu erstellen. So zum Beispiel abhängige Bibliotheken, Release- und Debuginformationen, Zielordner sowie auch plattformabhängige Konfigurationen.

Neben dem MOC und QMake stellt Qt noch zahlreiche weitere Tools zur Verfügung. Der Qt Designer in etwa erlaubt das Erstellen von grafischen Benutzeroberflächen nach dem What-You-See-Is-What-You-Get (WYSIWYG) Prinzip. Die User-Interface Dateien (.ui), die der Qt Designers erzeugt, werden beim Kompilieren in C++ Quellcode umgewandelt, weshalb es zu keinen Geschwindigkeitsproblemen zur Laufzeit kommt und sich keinerlei Nachteile zur inline Programmierung der Benutzeroberfläche ergeben.

Ein weiteres wichtiges Tool des Qt-Frameworks ist der Qt Linguist. Qt Linguist umfasst Werkzeuge zum Erstellen und Integrieren verschiedener Sprachversionen in eine Qt Applikation. Innerhalb des Programmcodes wird das tr-Makro verwendet um ein Wort oder einen Satz für eine Übersetzung vorzubereiten. Mit weiteren Werkzeugen werden alle Quellcode Dateien ausgelesen, und es werden automatisch (.ts) Sprachdateien erzeugt, die durch einen Übersetzer an die gewünschte Landessprache angepasst werden können. Eine Qt Applikation kann nun so angepasst werden, dass sie zum Start automatisch die Betriebssystemsprache lädt oder sogar einen Wechsel zur Laufzeit ermöglicht. Interessant ist auch, dass bei einer nicht vorhandenen Sprachdatei der Text des tr-Makros verwendet wird und es so zu keinem Totalausfall kommen kann. Als Zeichenkodierung wird überall in Qt UTF-8 verwendet.

Zu guter Letzt sei das Ressourcen-System, als ein weiterer wichtiger Bestandteil von Qt, erwähnt. Dieses erlaubt es Dateien direkt in die ausführbare

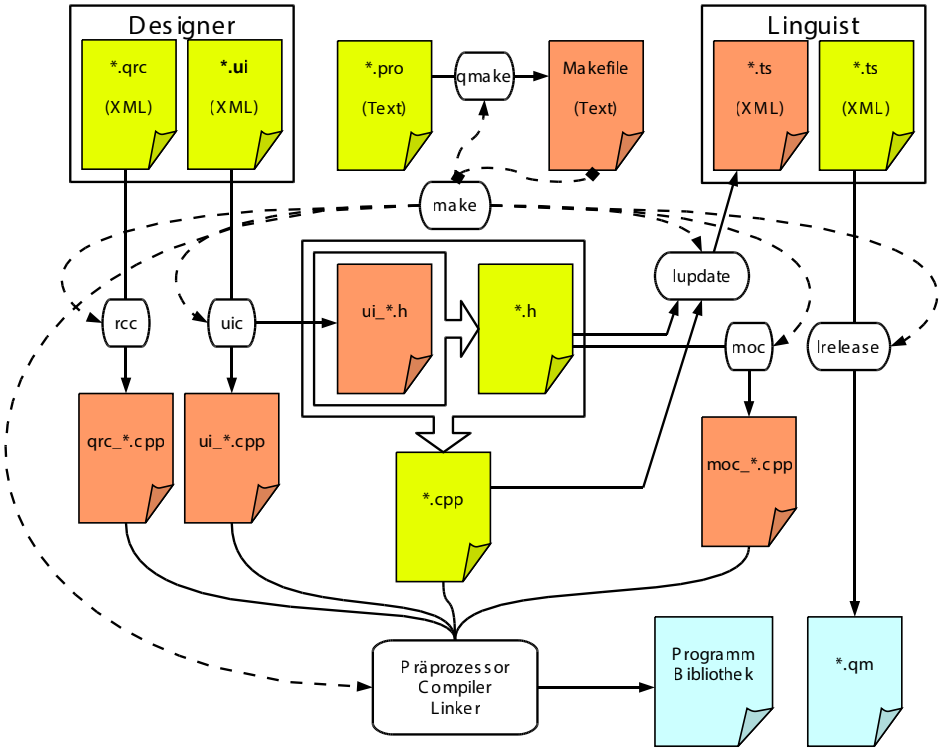

Abbildung 3.10: Das Qt Build-System (Wikimedia Foundation Inc., 2012)

Datei zu integrieren, was vor allem für Sprachdateien oder Icons sehr hilfreich ist. Hierzu werden die Pfade der zu integrierenden Ressourcen in (.qrc) Dateien kodiert. Innerhalb des Programmcodes kann dann zur Laufzeit auf die Ressourcen zugegriffen werden, und es ist garantiert, dass diese zur Verfügung stehen.

Das Zusammenwirken der genannten Tools und Dateien wird schematisch in Abbildung 3.10 dargestellt.

3.3.2 Objektmodell

Qt erweitert das C++ Objektmodell um zahlreiche Eigenschaften (Nokia Corporation, 2012b). Das oberste Objekt in der Vererbungshierarchie ist das `QObject`, von dem die meisten Klassen in Qt erben. Eine wichtige Eigenschaft des `QObjects` ist der Objektbaum. In den Objektbaum eines Objektes können andere Objekte eingehangen werden, die als Kinder bezeichnet werden. So lassen sich sehr gut Objekthierarchien abbilden, die gerade bei Benutzeroberflächen entstehen. Wenn nun ein Objekt zur Laufzeit gelöscht wird, werden automatisch alle Kinder des Objektes mit freigegeben. Dadurch

ergibt sich eine sehr intuitive und einfache Objekt- und Speicherverwaltung, welche zudem viele Fehler vermeiden kann. Als einen Anwendungsfall kann ein Fenster einer jeden Applikation herangezogen werden. Diese bestehen in der Regel aus vielen unterschiedlichen Elementen (Buttons, Slider, Textfelder, ...). Wenn nun das Fenster geschlossen wird, werden alle Elemente automatisch mit freigegeben. Das spart viel Zeit bei der Programmierung und beugt gegen potentielle Speicherlücken vor.

Das bereits erwähnte tr-Makro ist ebenfalls ein Teil des `QObjects`. Dadurch wird jedes Objekt, was von `QObject` erbt, sofort für eine mögliche Übersetzung vorbereitet.

Eine weitere wichtige Erweiterung ist das Event-System. Das Event-System ist Bestandteil von `QObject` und kapselt jegliche Form von Ereignissen des Betriebssystems. Beispiele sind Maus und Tastatur Events oder das Verändern der Größe eines Fensters.

Neben einem Mechanismus zum dynamischen Umwandeln von Qt Objekten und dem Signal und Slot Mechanismus, welcher in Abschnitt 3.3.3 genauer beleuchtet wird, sieht das Objektmodell noch zahlreiche weitere Eigenschaften vor. Diese sind für die vorliegende Arbeit allerdings weniger von Belang und können bei Bedarf in der Dokumentation (Nokia Corporation, 2012b) nachgeschlagen werden.

3.3.3 Signal-Slot-Konzept

Ein elementarer Bestandteil von Qt ist das Signal-Slot-Konzept (Nokia Corporation, 2012c). Dieses gehört nicht zum C++ Standard, weshalb hier der bereits erwähnte MOC zum Einsatz kommt. Das Signal-Slot-Konzept erlaubt allen von `QObject` erbenden Klassen einen mächtigen Kommunikationsmechanismus zwischen Objekten zu implementieren, welcher Gemeinsamkeiten mit dem Beobachtermuster (vgl. Gamma u. a., 1996, S. 287 ff.) sowie dem Callback Mechanismus hat, und gleichzeitig typsicher ist. Die Verwendung benötigt lediglich wenige Zeilen Code.

Bei Signals und Slots handelt es sich um Funktionsdeklarationen in der Header Datei einer C++ Klasse, welche durch einen speziellen **signals:** oder **slots:** Ausdruck angekündigt werden. Signals sind Benachrichtigungen, die Objekte der jeweiligen Klasse aussenden können. Slots hingegen sind Funktionen, die auf ein Signal hin aufgerufen, aber auch wie eine normale Member-Funktion verwendet werden können. Signale hingegen haben keinen Funktionsrumpf und können nur innerhalb der Klasse dazu verwendet werden ein Signal zu senden. Um auf einen Signalaufruf hin eine Funktion auszulösen, wird mittels der connect-Funktion - welche ihrerseits eine Member Funktion von `QObject` ist - eine Verbindung hergestellt. Die Nutzung des Signal-Slot-Konzepts wird in Abbildung 3.11 beispielhaft verdeutlicht.

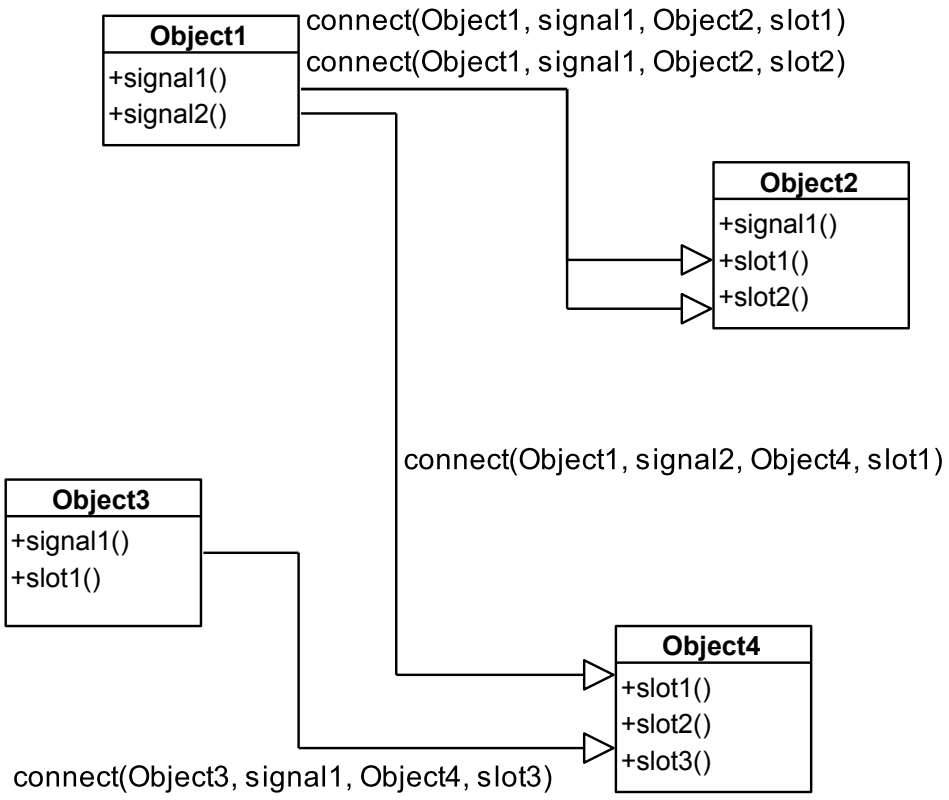

Abbildung 3.11: Das Qt Signal-Slot-Konzept (Nokia Corporation, 2012c)

Kapitel 4

Anforderungsanalyse

Im folgenden Kapitel werden die Anforderungen aufgelistet, die bei der Entwicklung von RelaView berücksichtigt werden sollten. Dabei ist der größte Teil der Anforderungen zu Beginn des Projektes entstanden. Durch einen iterativen Entwicklungsprozess wurden zahlreiche Anpassungen vorgenommen und Anforderungen hinzugefügt oder entfernt. Die groben Ziele wurden bereits in Abschnitt 2.2 dargelegt, weshalb nun lediglich eine Auflistung und Beschreibung der funktionalen und nichtfunktionalen Anforderungen stattfindet.

1. **Funktionale Anforderungen**

   1.1. **RelaFix**

      1.1.1. Aus RelaFix Code Dateien (.rfc) sollen Relationen geladen werden können

      1.1.2. Wenn RelaFix Code Dateien mehr als eine Relation enthalten, soll dem Nutzer die Wahl ermöglicht werden, welche dieser Relationen geladen werden sollen

      1.1.3. Für die Verarbeitung von Relationen soll die RelaFix Bibliothek herangezogen werden

   1.2. **Darstellung von Relationen**

      1.2.1. Eine Relation soll in Form einer booleschen Matrix dargestellt und verarbeitet werden

      1.2.2. Es sollen sowohl homogene als auch inhomogene Relationen dargestellt werden können

      1.2.3. Jede Relation soll in einem separaten Fenster dargestellt werden

      1.2.4. In die boolesche Matrix soll rein- und rausgezoomt werden können

      1.2.5. Die Eigenschaften einer Relation sollen auf einen Blick ersichtlich sein

1.2.6. Die Namen der Relation und der Mengen sollen dargestellt werden

1.2.7. Die in Relation stehenden Elemente sollen einsehbar sein

1.2.8. Äquivalenzrelationen sollen sich auf Anfrage zu ihrer Blockform umsortieren lassen

1.2.9. Zeilen und Spalten sollen sich händisch umordnen lassen

1.3. **Verarbeitung von Relationen**

1.3.1. Eine Relation soll durch das Anklicken eines Matrixelements verändert werden können, indem das entsprechende 2-Tupel der Relation hinzugefügt oder entfernt wird, woraufhin sich die Darstellung der Relation aktualisieren soll

1.3.2. Nicht erfüllte Relationseigenschaften sollen entweder durch das Entfernen oder Hinzufügen von Elementen vervollständigt werden können

1.3.3. Veränderungen an der Matrix / Relation sollen rückgängig gemacht werden können

1.4. **Weitere Funktionen**

1.4.1. Relationen und Mengen sollen in Form von RelaFix Code abgespeichert werden können

1.4.2. Neue Relationen sollen erstellt werden können

1.4.3. Bei der Erstellung neuer Relationen sollen alle bereits existierenden Mengen zur Auswahl stehen

1.4.4. Neue Mengen sollen angelegt werden können

1.4.5. Der Name einer Relation soll verändert werden können

1.4.6. Fehlermeldungen der RelaFix Bibliothek sollen einsehbar sein

1.4.7. Lange Ladezeiten sollen durch eine Fortschrittsanzeige visualisiert werden

2. **Nicht funktionale Anforderungen**

2.1. Auch für große Relationen ($> 100 \times 100$) soll die Skalierung der Matrix sowie eine Veränderung der Relation, auf durchschnittlichen Notebooks, nicht spürbar verzögert ablaufen

2.2. Erweiterungen von RelaFix um Eigenschaften und Operationen sollen möglichst einfach in RelaView integriert werden können

2.3. Die Software soll Windows, OSX und Linux/BSD unterstützen

2.4. Die Software soll an die Eigenheiten der Betriebssysteme angepasst sein (z.b. Standard-Tastenkombinationen)

2.5. Die Software soll unter den unterstützten Betriebssystemen nativ wirken

Kapitel 5

Systemdesign und Implementierung

In diesem Kapitel werden Designentscheidungen und Teile der Implementierung besprochen, die den wesentlichen Funktionsumfang von RelaView abdecken. Eine detaillierte Beschreibung der Implementierung übersteigt den Umfang dieser Arbeit um ein Vielfaches, weshalb auf die Erklärung von gewöhnlichen Algorithmen, wie etwa dem Undo/Redo Mechanismus, verzichtet wird. Als Hauptentwicklungsziel wurde die OSX Plattform angestrebt, wenn auch eine regelmäßige und ausführliche Überprüfung der Software unter Windows und Linux (Ubuntu und Debian) und teilweise FreeBSD stattfand. Als Entwicklungsumgebung wurde die Qt Creator Software in Verbindung mit dem CLang Compiler (http://clang.llvm.org) unter OSX, dem MinGW Compiler System (http://www.mingw.org) unter Windows und dem GCC (http://gcc.gnu.org) unter Linux verwendet sowie dem GCC Debugger (http://www.gnu.org/software/gdb). Die verwendete LGPL Version der Qt Library ist dynamisch gegen RelaView gelinkt, kann also auch im Nachhinein ausgewechselt werden. RelaView wird ebenfalls unter der LGPL veröffentlicht und wird, nach Abgabe und Veröffentlichung dieser Ausarbeitung, auf GitHub (https://github.com) veröffentlicht.

5.1 Klassenüberblick

Beginnen sollen die Ausführungen nun mit einem kurzen Überblick über die verschiedenen Klassen von RelaView. Im Wesentlichen besteht RelaView aus den folgenden fünf verschiedenen Arten von Klassen:

- Model
- View
- Controller

- Modifier
- UndoCommands

Zu den Model Klassen gehören alle Klassen, die Daten beinhalten oder zur Verfügung stellen. Dazu gehören die Klassen: `RelationData` und `DomainData`. `RelationData` und `DomainData` kapseln die RelaFix Strukturen `RF_RELATION` und `RF_DOMAIN` - welche RelaFix Relationen und Mengen abbilden - und stellen somit eine Schnittstelle zur RelaFix Bibliothek her. Dadurch wird eine Überprüfung von Werten sowie das Verwenden des Signal-Slot-Konzepts ermöglicht. Zudem wird die Anwendung des objektorientierter Paradigmas von C++ gewährleistet.

Unter den View Klassen befinden sich alle Klassen, die für die Darstellung von Fenstern und Dialogen verantwortlich sind und dem Nutzer eine Interaktion mit der Software ermöglichen. Das Herzstück der View Klassen ist die `MainWindow` Klasse, welche das Hauptfenster der Applikation darstellt. Abbildungen dieser finden sich in Kapitel 6. Weiterhin sei die `MatrixView` Klasse erwähnt, welche ein Unterklasse der `QGraphicsView` Klasse ist, und genauer in Abschnitt 5.5 beleuchtet wird. Außer ein paar Änderungen bezüglich des Drag und Drop Verhaltens, unterscheidet sich die `MatrixView` nicht von der `QGraphicsView`, weshalb weiterhin auch nur von der `QGraphicsView` Klasse die Rede ist. Obwohl zahlreiche weitere View Klassen existieren - tatsächlich machen die View Klassen die größte Menge des Quellcodes aus - sollen diese hier unerwähnt bleiben.

Bei den Controller Klassen handelt es sich um die verwaltenden Strukturen von RelaView. Es handelt sich um die Klassen und Dateien: `DataManager`, fileloader.c, `ModificationFactory` sowie `SequenceCounter`. Die `DataManager` Klasse ist einer der erwähnenswertesten und ist verantwortlich für das Vorhalten von Mengen und Fehlern sowie für die Kommunikation mit dem Kommandozeilen-Tool von Peter Berger, zur Ansteuerung des RelaFix Parsers. Das Kommandozeilen-Tool befindet sich, in einer angepassten und veränderten Form, in der fielloader.c Datei. Der `DataManager` dient dazu Relationen nach dem Ablauf des Parsers in einer Liste zu puffern, bis der Nutzer die Entscheidung getroffen hat, welche der vom Parser erkannten Relationen benutzt werden sollen. Danach wird der Besitz an der Relation an die `MainWindow` Klasse abgegeben und die Liste für das nächste Laden geleert. In Abschnitt 5.2 wird das Zusammenwirken der Komponenten genauer erläutert. Die `ModificationFactory` Klasse ist verantwortlich für die Anbindung aller RelaFix Operationen und Eigenschaften, die aus RelaView zu bedienen sind. Sie wird genauer in Abschnitt 5.4 betrachtet. Als letzte Klasse der Controller Klassen sei die `SequenceCounter` Klasse erwähnt. Diese stellt eine Zwischenschicht zwischen den eigentlichen Relationsdaten und der Darstellung dar, und ermöglicht das Umordnen von Zeilen und Spalten der Matrix. In Abschnitt Abschnitt 5.6.2 wird dies genauer erläutert.

Als Modifier Klassen werden all jene Klassen verstanden, die für die Kapselung einzelner RelaFix Eigenschaften und Operationen verantwortlich sind. Eine ausführliche Erklärung der Klassen findet sich ebenfalls in Abschnitt 5.4.

Zu guter Letzt seien die Klassen genannt, die unter der Kategorie UndoCommands zu führen sind. All diese Klassen erben von der Klasse `QUndoCommand`, welche ihrerseits Teil des Undo/Redo Frameworks von Qt sind. Das Undo/Redo Framework von Qt implementiert das allseits bekannte Befehlsmuster (vgl. Gamma u. a., 1996, S. 273 ff.). Die Klassen kapseln alle Veränderungen an RelaView, wie z.B. das Ändern eines Elementes oder das Verwenden einer Operation. Innerhalb der Klassen befinden sich undo und redo Funktionen, welche für die Funktion Rückgängig und Wiederherstellen in RelaView herangezogen werden.

Weiterhin sei erwähnt, dass in allen Klassen von RelaView das, bereits in Abschnitt 3.3.1 erwähnte, tr-Makro für die Darstellung von Text verwendet wird - ausgenommen der Fehlermeldungen von RelaFix. Dadurch wird es möglich RelaView für nahezu jede Sprache zu übersetzen. Zur Zeit unterstützt RelaView Englisch und Deutsch. Beim Start erkennt RelaView automatisch die Sprache des Betriebssystems und versucht für diese eine Übersetzung zu laden. Wird keine Übersetzung gefunden, wird automatisch Englisch verwendet.

5.2 Anbindung von RelaFix

Die Anbindung einer C++ Applikation an RelaFix hat sich als etwas trickreich herausgestellt. Wie bereits in Abschnitt 3.2 dargelegt, ist in RelaFix ein Parser integriert, um aus Eingaben - zum Beispiel in Form einer .rfc Datei - Relationen zu erzeugen. Im Parser sind bestimmte C-Funktionen angegeben, welche aufgerufen werden, um Relationen, Mengen oder Fehler anzuzeigen. Das bedeutet, dass jedes Programm, welches RelaFix und die Funktionalität des Parsers nutzen möchte, eben jene C-Funktionen zur Verfügung stellen muss.

Ein Beispielprogramm, welches die Anbindung an RelaFix demonstriert, wurde ebenfalls von Peter Berger entwickelt. Dieses Kommandozeilen-Tool wurde in großen Teilen auch zur Anbindung von RelaView an RelaFix verwendet. Zur Verdeutlichung, wird in Abbildung 5.1 die Interaktion der einzelnen Softwarekomponenten in einem Sequenzdiagramm verdeutlicht. In diesem konkreten Beispiel, wird eine .rfc Datei geöffnet, in einem Fenster dargestellt und anschließend wieder geschlossen.

Wie zu erkennen, werden für das Laden von Relationen in RelaView die Funktionen des Kommandozeilen-Tool (showRelation(), showError() und showDomain()) verwendet, welche die Relationen und Mengen eigentlich bloß auf der Kommandozeile abbilden, hier aber zusätzlich dem `DataManager`

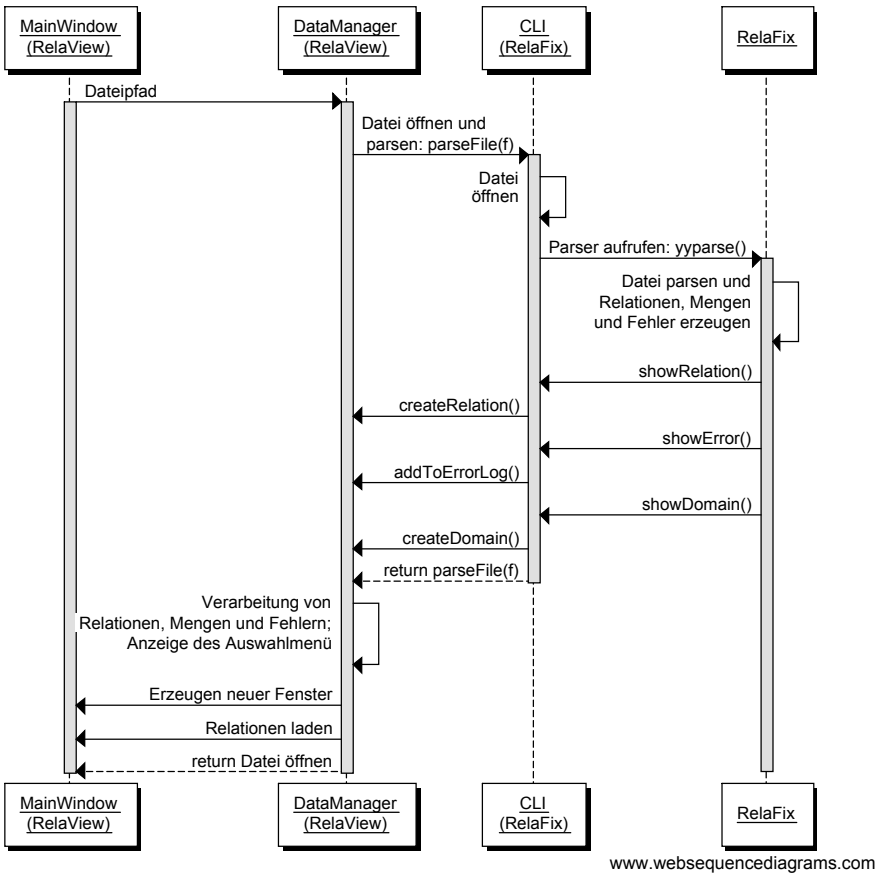

Abbildung 5.1: Anbindung von RelaFix an RelaView

die entsprechenden Daten übergeben. Das Aufrufen der show-Funktionen geschieht aber nicht durch die bloße Deklaration einer Menge oder Relation in einer RelaFix Code Datei, sondern erfordert einen Aufruf. Aus diesem Grund muss jede Relation, die RelaView laden können soll, durch den Ausdruck `relationsname;` in einer RelaFix Code Datei angekündigt werden, wie in Zeile 22 in Abbildung 3.9 zu erkennen.

Weiterhin sei erwähnt, dass RelaFix statisch gegen RelaView gelinkt ist, also nicht, ohne RelaView neu zu Kompilieren, ausgetauscht werden.

5.3 Algorithmen zum Vervollständigen von Eigenschaften

Ein elementarer Bestandteil von RelaView stellt die Vervollständigung von Relationseigenschaften dar. In diesem Abschnitt werden die von RelaView

verwendeten Verfahren zur Vervollständigung, die im Rahmen dieser Arbeit entwickelt und in RelaFix integriert worden, dargestellt und erläutert. Es wird beschrieben, wie die Vervollständigung der jeweiligen Eigenschaften erreicht wird, für die konkreten Algorithmen wird, aus Platzgründen, auf den Quellcode verwiesen, wenn auch für ausgewählte Beispiele der Quellcode vorgestellt wird. Des Weiteren soll an dieser Stelle angemerkt werden, dass es kein „Patentrezept“ für die Vervollständigung gibt. Es existieren in einigen Fällen mehrere Möglichkeiten eine Eigenschaft hervorzubringen, wovon die hier vorgestellten Algorithmen nur eine Teilmenge bilden.

Der Ausgang für die Entwicklung der Algorithmen waren die Implementierungen der RelaFix Algorithmen, zum Überprüfen der jeweiligen Eigenschaft. Zu Beginn eines jeden Vervollständigungsalgorithmus wird der RelaFix Algorithmus zum Überprüfen der jeweiligen Eigenschaft ausgeführt. Der Rückgabewert dieser wird daraufhin ausgewertet. Wenn ein Fehler aufgetaucht ist - z.B. weil die Relation inhomogen ist, die Eigenschaft aber nur für homogene Relationen gilt - oder die Relation bereits die Eigenschaft erfüllt terminiert der Vervollständigungsalgorithmus mit dem Rückgabewert des Fehlers. Im anderen Fall läuft der Algorithmus weiter und führt sich am Ende noch einmal selber aus, um erneut die anfängliche Überprüfung durchzuführen. Dies wird so lange wiederholt, bis dieser erfolgreich oder mit einem Fehler terminiert. Ein Beispiel für dieses Vorgehen wird in Abbildung 5.2 demonstriert.

Zusätzlich besitzen einige der Funktionen einen Parameter vom Typ bool, der das interne Verhalten eines Algorithmus bestimmt; ob zum Beispiel Elemente aus der Relation entfernt oder ergänzt werden. Genaueres wird in der Beschreibung der einzelnen Algorithmen besprochen.

Reflexivität

Um Reflexivität zu erzielen ist es lediglich nötig die Relation mit der Identitätsrelation zu Vereinigen, beziehungsweise der Relation alle 2-Tupel der Form (x, x) hinzuzufügen. Der Algorithmus hierfür wird in Abbildung 5.2 aufgelistet.

Irreflexivität

Irreflexivität kann erreicht werden, indem von einer Relation die Identitätsrelation abgezogen wird $(R \setminus I_R)$. Anhand der Matrix kann sich auch vorgestellt werden, dass alle Matrixelemente der Diagonalen auf 0 gesetzt werden.

Symmetrie

Um eine Relation symmetrisch werden zu lassen, ist es notwendig, dass die Relation an ihrer Diagonalen spiegelsymmetrisch wird. Dies kann erzielt wer-

```
int rf_relation_gain_reflexive(RF_RELATION* relation)
{
    int res = rf_relation_is_reflexive(relation);
    if (res != 0)
        return res;

    int x, y, width, result;

    width = rf_table_get_width(relation->table);
    for (x = 0, y = 0; x < width; x++, y++)
    {
        result = rf_table_get_bit(relation->table, x, y);
        if (result == 0)
            rf_table_set_bit(relation->table, x, y, 1);
    }

    return rf_relation_gain_reflexive(relation);
}
```

Abbildung 5.2: C-Algorithmus zum Erreichen von Reflexivität

den, indem die obere Ecke der Matrix betrachtet und jedes Matrixelement (x, y) darauf untersucht wird, ob es den selben Wert wie sein Spiegelpaar (y, x) hat. Ist dies nicht der Fall, werden beide Matrixelemente entfernt oder hinzugefügt, was abhängig davon ist, ob der Nutzer sich für das Auffüllen der Relation oder Entfernen von Elementen entschieden hat. In Abbildung 5.3 ist die verwendete Implementierung angegeben.

Asymmetrie

Für Asymmetrie wurde ebenfalls der Algorithmus von RelaFix zum Überprüfen auf Asymmetrie als Ausgangspunkt der Entwicklung genommen. Der Algorithmus besitzt zwei Punkte, an denen eine Vervollständigung angesetzt werden muss.

Am ersten Punkt überprüft der Algorithmus, ob alle Diagonalelemente (x, x) enthalten sind. Ist dies der Fall, gilt die Relation als nicht asymmetrisch. Im Algorithmus zum Vervollständigen werden an dieser Stelle somit einfach alle Elemente (x, x) entfernt.

Am zweiten Punkt wird überprüft, ob die gegenüberliegenden Matrixelemente (x, y) und (y, x) den gleichen Wert besitzen. Ist dies der Fall terminiert der Überprüfungsalgorithmus erneut, und die Relation gilt als nicht asymmetrisch. Um Asymmetrie zu erreichen, müssen die beiden Elemente also unterschiedlich sein. An diesem Punkt stehen zahlreiche Möglichkeiten zur Verfügung, denn die Frage ist, welches Element wird hinzugefügt oder entfernt, um die Eigenschaft der Asymmetrie zu erreichen. Es wurde sich

```
int rf_relation_gain_symmetric(RF_RELATION* relation, RF_BOOL
    fill)
{
    int res = rf_relation_is_symmetric(relation);
    if (res != 0)
        return res;

    int x, y, width, result_1, result_2;
    width = rf_table_get_width(relation->table);
    for (y = 0; y < width - 1; y++)
        for (x = y + 1; x < width; x++)
        {
            result_1 = rf_table_get_bit(relation->table, x, y);
            result_2 = rf_table_get_bit(relation->table, y, x);
            if (result_1 != result_2){
                if (fill){
                    rf_table_set_bit(relation->table, x, y, 1);
                    rf_table_set_bit(relation->table, y, x, 1);
                }else{
                    rf_table_set_bit(relation->table, x, y, 0);
                    rf_table_set_bit(relation->table, y, x, 0);
                }

            }
        }
    return rf_relation_gain_symmetric(relation, fill);
}
```

Abbildung 5.3: C-Algorithmus zum Erreichen von Symmetrie

dazu entschieden, zwei Varianten zu implementieren. In der ersten Variante werden für alle Matrixelemente (x, y) und (y, x), die den gleichen Wert besitzen, die Elemente (x, y) auf 1 und die Elemente (y, x) auf 0 gesetzt. In der zweiten Variante genau umgekehrt. Das bedeutet, dass entweder die untere Dreiecksmatrix ergänzt wird oder aber die Obere. Die Wahl obliegt wie schon bei der Symmetrie dem Nutzer. Der komplette Algorithmus ist in Abbildung 5.4 aufgelistet.

Antisymmetrie

Um eine Relation antisymmetrisch werden zu lassen, darf kein Matrixelement (x, y) ein Spiegelpaar (y, x) mit dem gleichen Wert besitzen, es sei denn $x = y$. Daraus folgt, dass beim Auftauchen eines Matrixelementes mit $(x, y) \wedge (y, x)$ und $x \neq y$ entweder (x, y) auf 0 oder (y, x) auf 0 gesetzt werden muss. Wie schon die Male davor, entscheidet auch in diesem Fall der Nutzer darüber.

```
int rf_relation_gain_asymmetric(RF_RELATION* relation, RF_BOOL
    upper)
{
    int res = rf_relation_is_asymmetric(relation);
    if (res != 0)
        return res;

    int x, y, width, result_1, result_2;
    width = rf_table_get_width(relation->table);
    for (y = 0; y < width; y++)
        for (x = y; x < width; x++)
        {
            if (x == y)
            {
                result_1 = rf_table_get_bit(relation->table, x,
                    y);
                if (result_1 == 1){
                    rf_table_set_bit(relation->table, x, y, 0);
                }
                else if (result_1 == 0)
                    continue;
            }
            else
            {
                result_1 = rf_table_get_bit(relation->table, x,
                    y);
                result_2 = rf_table_get_bit(relation->table, y,
                    x);

                if (result_1 == result_2){
                    if (upper){
                        rf_table_set_bit(relation->table, x, y,
                            0);
                        rf_table_set_bit(relation->table, y, x,
                            1);
                    }else{
                        rf_table_set_bit(relation->table, x, y,
                            1);
                        rf_table_set_bit(relation->table, y, x,
                            0);
                    }
                }
            }
        }
    return rf_relation_gain_asymmetric(relation, upper);
}
```

Abbildung 5.4: C-Algorithmus zum Erreichen von Asymmetrie

Transitivität

Vergleichsweise simpel zeigt sich das Vervollständigungsverfahren der Transitivität. Die Transitivität besagt, dass wenn eine Relation die Tupel (x, y) und (y, z) beinhaltet, auch x mit z in Relation stehen muss (xRz). Wenn dies nicht der Fall ist, wird entweder das Tupel (y, z) entfernt, oder aber das Tupel (x, z) hinzugefügt, je nachdem, ob der Nutzer Elemente entfernen oder hinzufügen möchte.

Difunktionalität

Das wohl komplexeste Vervollständigungsverfahren ist das der Difunktionalität. Der Algorithmus sucht nach allen y, die mit x in Relation stehen (yRx) sowie nach allen Elementen z, die ebenfalls mit x in Relation stehen (zRx). Im nächsten Schritt werden alle Elemente w gesucht, die mit z in Verbindung stehen (zRw). Um Difunktionalität erfüllen zu können, muss jetzt y ebenfalls mit w in Relation stehen (yRw). Ist dies nicht der Fall, existieren zwei Möglichkeiten die Relation difunktional werden zu lassen. Wenn der Algorithmus dazu aufgerufen wurde Elemente hinzuzufügen, wird y mit w in Relation gestellt (yRw), also das Tupel (y, w) der Relation hinzugefügt. Soll eine Vervollständigung durch entfernen erfolgen, wird das Tupel (z, w) entfernt. Abbildung 5.5 stellt dies anschaulich dar. Die gepunktete Linie stellt die Verbindung dar, die hinzugefügt wird und die gestrichelte die, die entfernt wird. Wie der Algorithmus vorgeht wird durch den Nutzer bestimmt.

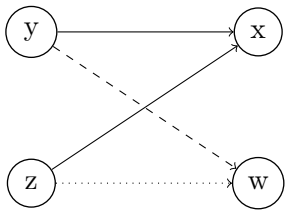

Abbildung 5.5: Verfahren zum Erreichen von Difunktionalität

Zusammengesetzte Relationen

Wie bereits in Abschnitt 3.1.2 dargestellt, setzten sich die Eigenschaften Äquivalenz, Halbordnung und Qusiordnung aus anderen Eigenschaften zusammen. Nicht anders verhält es sich bei der Vervollständigung einer Relation zu den entsprechenden Eigenschaften. Abbildung 5.6 demonstriert dies am Beispiel des Algorithmus zum Erzielen von Äquivalenz.

```
int rf_relation_gain_equivalent(RF_RELATION* relation, RF_BOOL
    fill)
{
    int res = rf_relation_is_equivalent(relation);
    if (res != 0)
        return res;

    rf_relation_gain_reflexive(relation);
    rf_relation_gain_symmetric(relation, fill);
    rf_relation_gain_transitive(relation, fill);

    return rf_relation_gain_equivalent(relation, fill);
}
```

Abbildung 5.6: C-Algorithmus zum Erreichen von Äquivalenz

5.4 Dynamische Anbindung von RelaFix Funktionen

Wie bereits im Einführungskapitel 3.1 festgestellt, existieren zahlreiche weitere Relationseigenschaften und Operationen für homogene und inhomogene Relationen, die noch nicht Einzug in RelaFix und RelaView gefunden haben. Deshalb wurde bei der Entwicklung darauf geachtet, eine potentielle Erweiterung von RelaFix, um weitere Eigenschaften und Operationen, möglichst schnell und unkompliziert in RelaView einfließen lassen zu können. Aus diesem Grund wurde die Anbindung an RelaFix dynamisch gestaltet, sodass lediglich wenige Zeilen Code für die Erweiterung von RelaView nötig sind.

Die Umsetzung der dynamischen Anbindung ist in Abbildung 5.7 stark vereinfacht in Form eines UML2 Klassendiagramms dargestellt. Um die Übersicht zu wahren, wurde auf eine vollständige Darstellung der Klassen verzichtet. Das bedeutet konkret, dass Parameter von Funktionen und Funktionszeigern durch ein $*$ Symbol ausgetauscht wurden.

Bei der Gestaltung der Anbindung wurden sechs verschiedene Arten von Eigenschaften und Operationen ausfindig gemacht, welche den Funktionsumfang der RelaFix Bibliothek abdeckt und gleichzeitig eine potentielle Erweiterung um weitere Eigenschaften und Operationen berücksichtigt. Die Eigenschaften und Operationen werden jeweils in einem Objekt gekapselt, welches einen Namen, einen anzuzeigenden Text und einen oder mehrere Funktionszeiger auf RelaFix C-Funktionen besitzt.

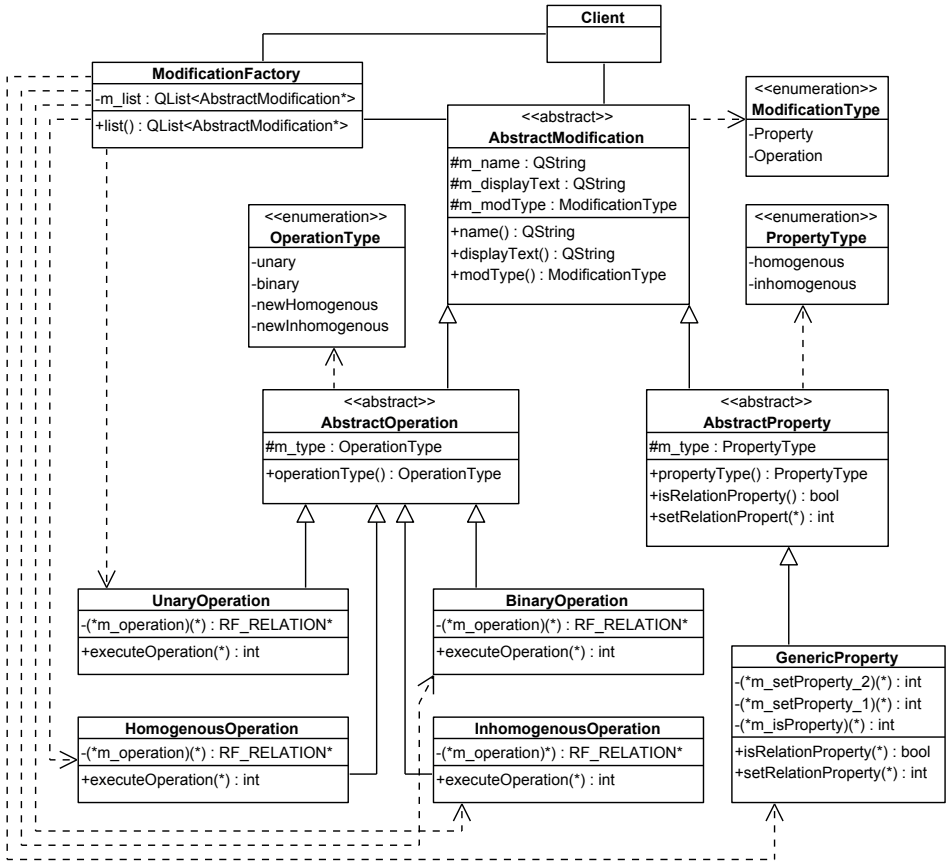

Abbildung 5.7: Dynamische Eigenschaften

In der obersten Klasse der dargestellten Hierarchie befindet sich die `AbstractModification` Klasse, welche grundlegend zwischen den zwei Arten, Operation und Eigenschaft, unterscheidet. Diese existieren auch jeweils als abstrakte Unterklasse: `AbstractOperation` und `AbstractProperty`. Als konkrete Implementierung der `AbstractOperation` liegen die `UnaryOperation` und `BinaryOperation` vor, welche unäre und binäre Operationen auf Relationen kapseln, wie z.B. Komplement, Inverse oder Durchschnitt. Die Klassen `HomogenousOperation` und `InhomogenousOperation` stellen Operationen zum Erstellen neuer Relationen dar. Beispiele hierfür sind das Erstellen einer inhomogenen Nullrelation oder einer homogenen Identitätsrelation. Die einzige konkrete Implementierung der `AbstractProperty` Klasse ist die `GenericProperty` Klasse, welche alle Funktionen für Eigenschaften kapselt. Konkret besitzt die `GenericProperty` Klasse einen Funktionszeiger auf eine C-Funktion, welche überprüft, ob eine Eigenschaft erfüllt ist, und eine C-Funktion, welche zum Erfüllen der Relationseigenschaft herangezogen wird. Durch Member Funk-

tionen der Klassen können die Funktionszeiger dereferenziert und ausgeführt werden, was dem Nutzer in RelaView an verschiedenen Stellen möglich ist.

Ziel dieser aufwendigen Konstruktion war die Zusammenfassung aller RelaFix Funktionsanbindungen in einer Klasse, welche jedes Funktionsobjekt einmalig erzeugt, verwaltet und anderen RelaView Objekten zur Verfügung stellt. Diese Aufgabe wird von der `ModificationFactory` Klasse übernommen, welche eine rein statische Klasse ist. Sie ähnelt in ihrem Aufbau dem Abstrakten Fabrik Entwurfsmuster (vgl. Gamma u. a., 1996, S. 107 ff.). Die `ModificationFactory` erstellt bei der Klassenerzeugung alle Eigenschaften und Operationen und speichert diese in einer Liste, die nach außen hin zugreifbar ist, und von allen anderen RelaView Objekten benutzt werden kann. Das hat zur Folge, dass bei einer Erweiterung von RelaFix lediglich zwei Zeilen Code pro Erweiterung geschrieben werden müssen, um RelaView komplett anzupassen.

Um während der Laufzeit feststellen zu können, ob ein Objekt aus der Liste aller Modifikationen eine Eigenschaft, eine Operation oder sogar von einem bestimmten Untertyp ist, steht in C++ der `dynamic_cast` Mechanismus zur Verfügung. Dieser überprüft mittels RTTI Typen, zu welcher Klasse ein Objekt gehört. Dieses ist eine sehr zeitintensive Angelegenheit (vgl. Stephens u. Turkanis, 2006, S. 325 f.). Aus diesem Grund wurden die verschiedene Klassentypen durch eine `Enumeration` abgebildet. So muss zur Laufzeit nur eine deutlich performantere `dynamic_cast` Operation ausgeführt werden.

5.5 Relationsmatrix

Der zentrale Punkt der Software ist die Darstellung der Relationsmatrix. Anstatt die Matrix durch Einsen und Nullen aufzufüllen, wurden gefüllte (Eins) und nicht ausgefüllte Rechtecke (Null) gewählt. Der entscheidende Vorteil dieses Ansatzes besteht darin, dass die Elemente für das menschliche Auge leichter zu unterscheiden sind, und auch noch zu erkennen sind, wenn die Relation größere Ausmaße annimmt oder sehr klein dargestellt wird.

Anforderung 2.1. (auch für große Relationen eine performante Interaktion zu garantieren) zu erreichen stellte sich als größeres Problem heraus als zuvor angenommen. Der trivialste Ansatz für die Realisierung innerhalb des Qt Frameworks war die Verwendung der `QGraphicsView` sowie der `QGraphicscene` Klassen, welche für den Zweck entwickelt wurde mehrere 10000 `QGraphicItems` (Rechtecke,Kreise, Linien,...) aufzunehmen und zu verwalten. Dabei beinhaltet die `QGraphicScene` die eigentlichen Objekte und die `QGraphicsView` Klasse stellt die Szene dar. Einer der großen Vorteile dieses Ansatzes besteht darin, dass dadurch eine Menge Anforderungen auf ein mal erschlagen werden können. So bietet die `QGraphicsView` Klasse schon einen Mechanismus zum herein- und herauszoomen in die Szene,

wodurch automatisch alle Elemente der Szene skaliert werden. Wird sogar soweit hereingezoomt, dass die Darstellung der Szene größer wird als die Anzeigefläche, werden automatisch Scrollleisten eingeblendet. Darüber hinaus können der `QGraphicsScene` Objekte der `QGraphicsObject` Klasse hinzugefügt werden, welche von `QObject` erben und so die Verwendung des Signal-Slot-Konzepts erlauben. Diese Möglichkeiten führten zur Entwicklung der `MatrixElement` Klasse, dessen Objekte ein Matrixelement darstellen. Ein `MatrixElement` Objekt besitzt, neben der Darstellung als Rechteck, eine Position (x,y Koordinate) innerhalb der Matrix sowie ein Signal, welches ausgelöst wird, wenn das Element in der Darstellung angeklickt wird. Das Signal überträgt daraufhin die Koordinaten des angeklickten Elementes, und der daraufhin ausgeführte Slot kümmert sich um die Veränderung der Relationsdaten.

Dieser auf den ersten Blick kompliziert erscheinende Ansatz ist in Wirklichkeit sehr schlicht und simpel, vor allem im Vergleich zu einer eigenen Implementierung der Matrix und seiner Elemente. Demzufolge ist es bedauernswert, dass sich dieser Ansatz auf einem normalen Notebook als sehr unperformant herausstellte. Sobald eine Relation mehr als 10.000 Elemente besitzt, wird die Software nahezu unbenutzbar. Eine Erweiterung um eine Hardwarebeschleunigung durch OpenGL hat die Matrix nur unwesentlich beschleunigt.

Aus diesem Grund wurden in der Entwicklungsphase zwei weitere Ansätze ausprobiert. Zuerst wurde versucht eine reine OpenGL Implementierung zu entwickeln, da sich herausstellte, dass die OpenGL Hardwarebeschleunigung der `QGraphicsView` nicht die volle Performance der Hardware ausnutzen kann, da zur Laufzeit erst alle `QGraphicsItems` in OpenGL umgewandelt werden müssen, was zu einem erheblichen Geschwindigkeitseinbruch führt. Für RelaView wurde eine reine OpenGL Implementierung der Matrix allerdings nie vollständig implementiert, da sich die inline Verwendung von OpenGL, zu diesem Zeitpunkt, noch in der Testphase befand und nicht sehr stabil war. Erste Versuche wirkten allerdings vielversprechend, und ließen darauf hoffen, dass in Zukunft eine derartige Implementierung möglich wird - auch wenn die Implementierung aller Anforderungen, die bereits von `QGraphicsView` bereitgestellt werden, sehr aufwendig ausfallen wird.

Der letzte Ansatz bestand darin, die Matrix auf einem Bild zu rendern und dieses Bild zu puffern. Veränderungen, wie beispielsweise eine Skalierung, konnten so sehr schnell und performant auf dem gepufferten Bild ausgeführt werden. Änderungen, die die Matrix veränderten führten zum Erstellen eines neuen Bildes. Dieser Ansatz funktionierte ausgezeichnet, scheiterte aber daran, dass in Qt nicht bestimmt werden kann, wann das Skalieren eines Bildes oder das Verändern der Fenstergröße zu Ende sind. Diese beiden Events wären notwendig, um an diesem Zeitpunkt das Bild der Matrix zu aktualisieren, da sonst nur ein unscharfes, skaliertes Bild dargestellt wird.

Aus diesem Grund wurde sich bei der Entwicklung dazu entschieden die erste Implementierung beizubehalten und die Schwerpunkte der restlichen Entwicklungszeit auf die Umsetzung der restlichen Anforderungen sowie das Testen der Software zu konzentrieren.

5.6 Verfahren zum Sortieren von Äquivalenzrelationen

Wie bereits in Abschnitt 3.1.3 angekündigt, folgt nun die Erläuterung des Verfahrens, mit welchen in RelaView Äquivalenzrelationen sortiert werden. Da die Recherchen im Rahmen dieser Arbeit leider keine Algorithmen hervorgebracht haben, die das Sortieren von Äquivalenzrelationen ermöglichen, wurde der nachfolgende Algorithmus entwickelt. An diesem Punkt soll angemerkt werden, dass nicht ausgeschlossen werden kann, dass dieser Algorithmus bereits zuvor entdeckt wurde, noch dass er stabil ist oder beweisbar sicher funktioniert. Nachfolgend wird der Algorithmus erläutert und die Komplexitätsklasse bestimmt.

Der erste Teil des Algorithmus ist die Hauptschleife, dargestellt in (Algorithmus 1), welche die Matrixdiagonale von oben links nach unten rechts entlangläuft.

Algorithm 1: Verfahrens zum Sortieren einer Äquivalenzrelation - Hauptschleife

Data: boolesche Matrix einer Äquivalenzrelation
Result: Äquivalenzrelation in Blockform

```
height = Anzahl der Zeilen/Spalten der Matrix;
dia = 0;
while dia < height do
    sortColumn(dia);
    sortRow(dia);
    dia++;
end
```

Nun werden an jedem Punkt auf der Diagonalen Spalten, danach Zeilen so getauscht, dass auf einem Block von Einsen nur noch Nullen Folgen. Das Tauschen wird durch das Spalten-Sortierverfahren (Algorithmus 2), welches die Spalte des aktuellen Diagonalpunktes sortiert, sowie das Zeilen-Sortierverfahren erreicht. Der Zeilen-Sortieralgorithmus verhält sich analog, weshalb im Folgenden nur der Spalten-Sortieralgorithmus erklärt wird.

Ausgehend von einem Punkt auf der Diagonalen, betrachtet der Spalten-Sortieralgorithmus die restlichen Punkte darunter. Der Punkt an dem als erstes eine 0 auftaucht, wird in der `firstNullpos` Variablen vorgehalten

Algorithm 2: Verfahrens zum Sortieren einer Äquivalenzrelation - Spalten-Sortieralgorithmus (columnSort)

```
Data: boolesche Matrix einer Äquivalenzrelation
Result: Sortierte Spalte an übergebener Position
Input: Position der zu sortierenden Spalte

height = Zeilenhöhe;
diagonalStart = Position der zu sortierenden Spalte;
run = diagonalStart;
bit = 0;
firstNullpos = -1;
while run < height do
    x = diagonalStart;
    y = run;
    bit = Matrixelement an der Stelle x,y (0 oder 1);
    if (bit == 0) && (firstNullpos == -1) then
        firstNullpos = run;
    end
    if bit == 1 && firstNullpos ≠ -1 then
        Tausche Zeile(firstNullpos, run);
        firstNullpos++;
    end
    run++;
end
```

und dient zum Tauschen. Wenn bei der Betrachtung der restlichen Punkte auf dem Weg nach unten eine Eins auftaucht, wird die Zeile mit der vorgehaltenen getauscht, die `firstNullpos` Variabel wird um 1 erhöht und der Algorithmus folgt weiter der Zeile mach unten. Es wird so lange getauscht, bis das Ende der Spalte erreicht ist. Nach Terminierung des Spalten-Sortieralgorithmus folgt der Zeilen-Sortieralgorithmus, woraufhin das Hauptprogramm die Diagonale ein Element weiter wandert, bis der Algorithmus alle Punkte auf der Diagonalen durchlaufen hat. Dadurch werden alle Einsen blockweise zusammengefasst und die Äquivalenzrelation erhält die typische Blockform.

5.6.1 Bestimmung der Komplexitätsklasse

Betrachtet wird nun die Komplexität des Algorithmus. Die folgenden Schlussfolgerungen können anhand von Abbildung 5.8 nachvollzogen werden. Algorithmus 1 besucht jeden Punkt an der Diagonalen. Dabei handelt es sich bei einer homogenen Relation, dessen Menge N Elemente besitzt, um N Punkte. Weiterhin werden bei jeder Runde 2 Algorithmen aufgerufen (Spalten- und Zeilen-Sortierverfahren). Diese besitzen beide die gleiche Laufzeit, welche von Runde zu Runde um eins sinkt, da der Diagonalen im 1. Algorithmus nach unten gefolgt wird. Die beiden Algorithmen starten also jeweils mit einer Laufzeit von N, gefolgt von $N-1$ bis hin zu 1. Dieses Schema entspricht der Gaußchen Summenformel: $\frac{n(n+1)}{2}$. Da es sich um 2 Aufrufe handelt, ergibt sich eine Laufzeit von $2 \cdot \frac{n(n+1)}{2}$ woraus $n(n+1)$ und damit n^2+n folgt. Die Laufzeit n^2+n gehört zur Komplexitätsklasse $O(n^2)$.

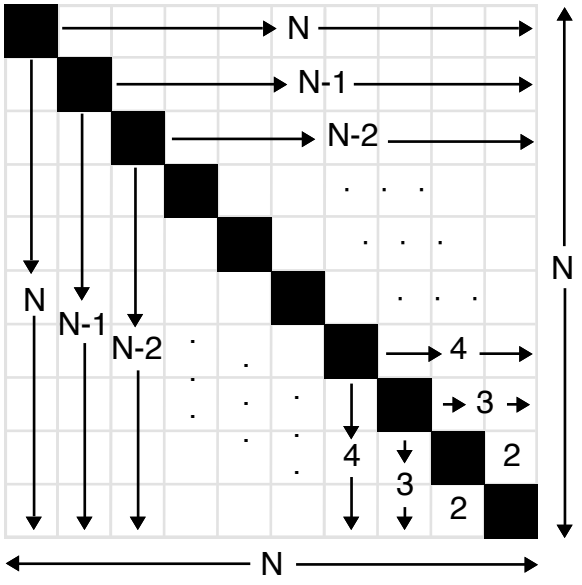

Abbildung 5.8: Verfahren zum Sortieren einer Äquivalenzrelation

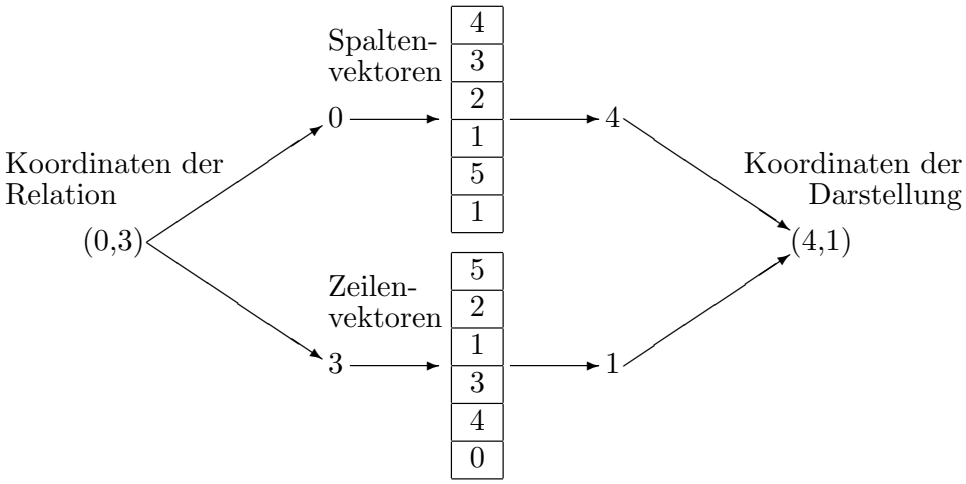

Abbildung 5.9: Funktionsweise der SequenceConverter Klasse

5.6.2 Bemerkung zur Integration in das Programm

Prinzipiell ist ein Tauschen von Zeilen und Spalten in RelaFix nicht vorgesehen. Ein Tauschen von Zeilen oder Spalten wäre in RelaFix nur möglich, wenn die Reihenfolge der Elemente der aufspannenden Mengen geändert werden würde, da von diese Reihenfolge abhängig ist, wie eine Relation dargestellt wird. In RelaFix verhalten sich zwei Mengen, die zwar die gleichen Elemente enthalten, aber in einer anderen Reihenfolge vorliegen, wie zwei unterschiedliche Mengen. Das bedeutet, dass z.B. Operationen, die auf der Gleichheit der Mengen basieren (z.B. die Verkettung), nicht mehr funktionieren würden. Viel mehr würde das auch bedeuten, dass alle anderen Relationen, die eine solch veränderte Menge nutzen würden, auf einmal auch anders dargestellt werden würden, da in RelaFix eine Menge von mehreren Relationen benutzt werden kann.

Aus diesem Grund wurde eine Zwischenschicht zwischen Darstellung der Daten in der `MatrixView` Klasse und den Relationsdaten in der `RelationData` Klasse eingeführt. Weiterhin stellt dies auch rein logisch eine gute Entscheidung dar, da so die Anordnung logisch von den eigentlichen Daten getrennt ist. Diese Zwischenschicht wird von der `SequenceConverter` Klasse realisiert. In dieser wird die Reihenfolge der Matrixzeilen und Matrixspalten durch zwei Arrays abgebildet. Das bedeutet, dass innerhalb der Arrays die eigentliche Position einer Zeile oder Spalte kodiert ist. Dies hat den großen Vorteil, dass sich die Darstellung durch einfaches umordnen der Werte in einem Array verändern lässt. Abbildung 5.9 verdeutlicht das geschilderte Zusammenspiel beispielhaft.

Wenn nun eine Relation dargestellt werden soll, werden die Koordinaten (x, y) eines jeden Elementes dem `SequenceConverter` übergeben. Dieser ermittelt daraufhin die Position (x', y'). Dazu wird der Wert, der im Array der Spaletenvektoren an Position x steht als eigentliche Koordinate x' verwendet. Dies geschieht analog für den Wert y'.

Kapitel 6

Softwareüberblick

In diesem Kapitel wird RelaView kurz vorgestellt und der Umgang mit der Software anhand von Abbildungen erläutert. Zu Beginn demonstriert Abbildung 6.1 das Hauptfenster von RelaView wie es unter OSX anzutreffen ist. In dieser speziellen Abbildung wurde die Äquivalenzrelation aus Abbildung 3.9 geladen. In der linken Hälfte befindet sich die Relation in Matrixdarstellung, die rechte zeigt die Eigenschaften der Relation. RelaView ist eine dokumentenbasierte Applikation, was bedeutet, dass für jede darzustellende Relation ein eigenes Fenster geöffnet wird. Das Verhalten ist bereits von Editoren bekannt. Weiterhin zeigen die Abbildung 6.2 und 6.3 das Aussehen der Benutzeroberfläche von RelaView unter Windows 7 und Linux/BSD mit einer Gnome Oberfläche.

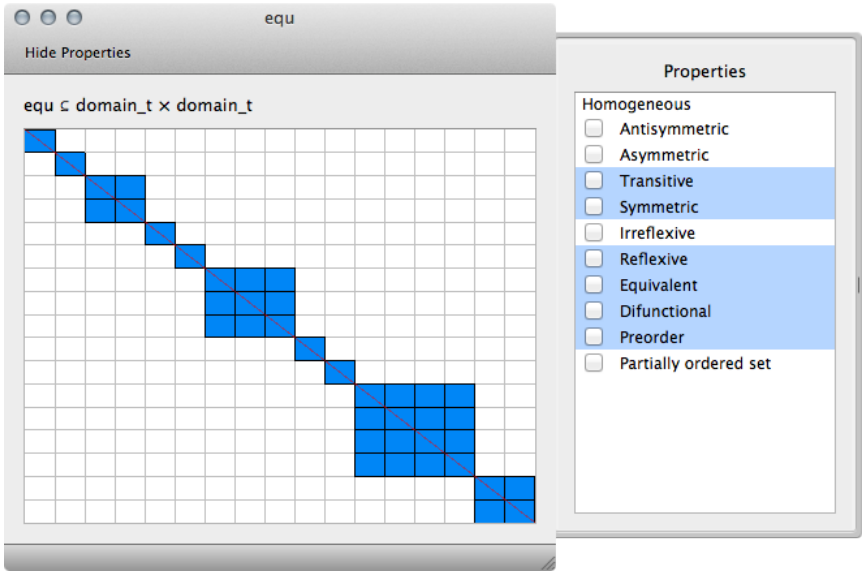

Abbildung 6.1: RelaView (Mac OSX)

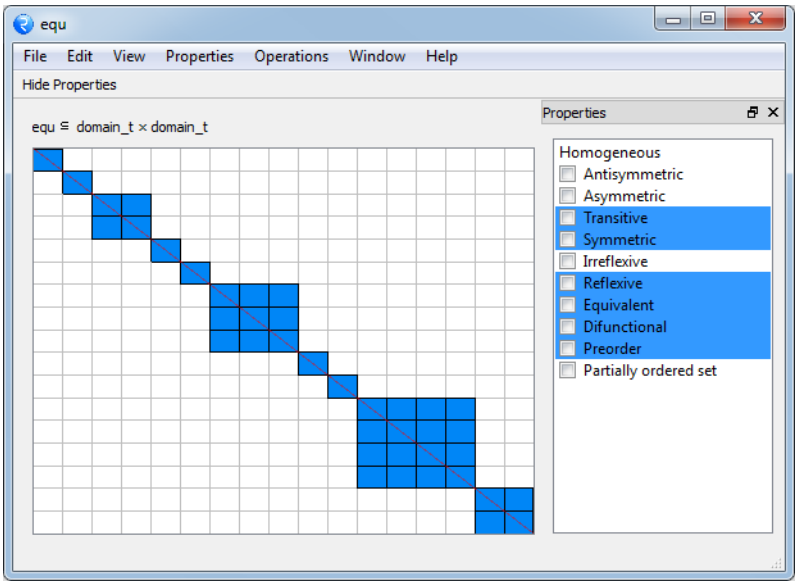

Abbildung 6.2: RelaView (Windows)

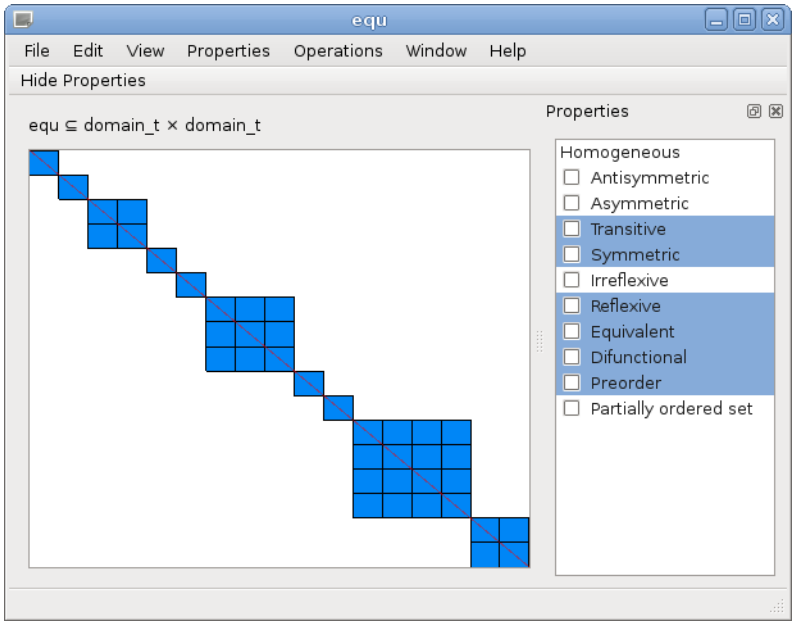

Abbildung 6.3: RelaView (Linux/BSD mit Gnome)

RelaView ist sehr schlicht gehalten und ist intuitiv zu bedienen. Wenn der Mauszeiger über ein Element der Matrix ruht, wird das entsprechende 2-Tupel in der Statusleiste und als Tooltip dargestellt. Wird auf ein Element geklickt, verändert sich die zugrundeliegende Relation sowie die Darstellung.

Der Grund, warum die Mengen der Relation nicht, wie aus einer Tabelle bekannt, an den Seiten aufgeführt werden, leitet sich aus der Möglichkeit ab, in die Relation rein- und rauszuzoomen.

Als nächstes sei die Menüzeile (Abbildung 6.4) erwähnt, durch welche der größte Teil der Funktionen von RelaView zu bedienen ist.

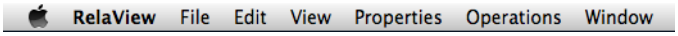

Abbildung 6.4: RelaView - OSX Menüleiste

Unter dem Menüpunkt *Properties* (Abbildung 6.5) befinden sich die Funktionen, um die Eigenschaften einer geladenen Relation zu vervollständigen. Wenn eine Relation eine Eigenschaft bereits erfüllt oder das Vervollständigen einer Eigenschaft auf die Relation gar nicht möglich ist, wird der Menüpunkt deaktiviert. Eine Vervollständigung ist z.B dann nicht möglich, wenn die geladene Relation inhomogen ist, es sich bei der zur vervollständigende Eigenschaft aber um eine homogene Eigenschaft handelt. Ein wichtiges Element ist der im Menü *Properties* auswählbare Menüpunkt *Fill*. Wenn dieser aktiviert ist, werden die Algorithmen zur Vervollständigung, die aus dem Menü ausgeführt werden dazu angeleitet der Relation Elemente hinzuzufügen. Wenn der Menüpunkt nicht selektiert ist, werden stattdessen Elemente entfernt. Da einige der in Abschnitt 5.3 vorgestellten Algorithmen nicht zwischen Hinzufügen oder Entfernen unterscheiden, schaltet das *Fill* bei diesen Algorithmen zwischen oberer und unterer Dreiecksmatrix hin und her. Dieses Verhalten hat sich als übersichtlicher bewiesen, als einen weiteren Schalter für diese Funktionalität zu implementieren, auch wenn es auf den ersten Blick wenig intuitiv wirkt. Hier kann in späteren Versionen eine verbesserte Lösung gefunden werden.

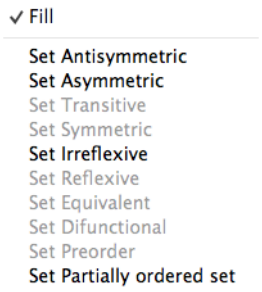

Abbildung 6.5: RelaView - Properties Drop-Down Menü

Unter dem Menüpunkt *Operations* (Abbildung 6.6) finden sich alle relationsverändernde Operationen. Während die Operationen in der oberen Hälfte des Separators, namentlich *Converse* und *Complememt*, die aktuelle Relation direkt verändern, führen die binären Operationen *Concatenate...*, *Intersect...* und *Union...* zum Auftauchen eines Dialoges, in dem die 2. Rela-

tion ausgewählt werden kann, die für die gewünschten Eigenschaft benötigt wird. Die resultierende Relation ersetzt dann die erste Relation. Wenn eine Operation nicht ausgeführt werden kann, wird ein Eintrag in das Fehlerprotokoll geschrieben.

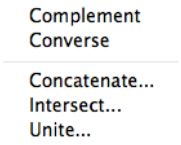

Abbildung 6.6: RelaView - Operations Drop-Down Menü

Das Fehlerprotokoll, zu sehen in Abbildung 6.7, protokolliert eine Vielzahl auftauchender Fehler von RelaView und RelaFix und wird automatisch angezeigt - insofern dies nicht durch die in der Abbildung zu erkennende Checkbox unterbunden wird. Das Fehlerprotokoll kann zudem durch *Window* → *Error Log* angezeigt werden. Dabei werden lediglich Fehler protokolliert, die dem Nutzer einen Mehrwert bieten und zu keinem Zeitpunkt Software- oder Speicherfehler. Der Ansatz des Fehlerprotokolls wurde an Stelle von Pop-up Fehlermeldungen gewählt, um den Nutzer nicht in seinem Arbeitsfluss zu unterbrechen. Lediglich kritische Fehler, wie etwa ein fehlgeschlagener Speicherversuch, werden dem Nutzer durch Pop-up Fenster mittgeteilt.

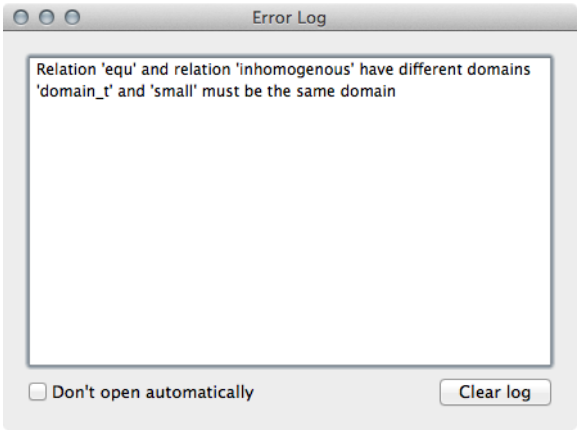

Abbildung 6.7: RelaView - Fehlerbericht

Die im Hauptfenster rechts dargestellte Auflistung von Eigenschaften erfüllt zwei Zwecke. Sie listet alle Eigenschaften, die von RelaView und RelaFix unterstützt werden, für eine Relation auf und zeigt durch eingefärbten Hintergrund an, welche dieser Eigenschaften erfüllt sind. Als Füllfarbe wird die Highlight-Farbe des Betriebssystems verwendet. Unterschiede, die bei der Auflistung der Eigenschaften auftauchen können, werden durch den Ver-

gleich der homogenen Relation aus Abbildung 6.1 und der inhomogenen Relation aus Abbildung 6.8 deutlich.

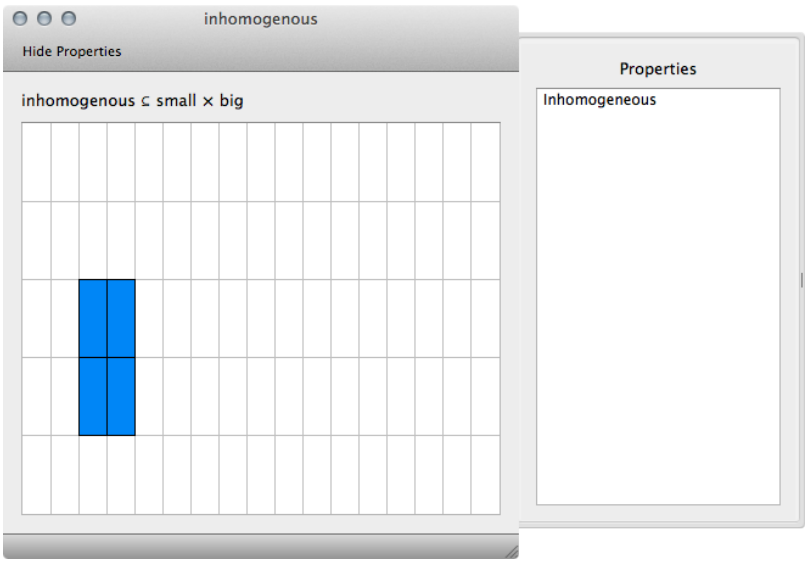

Abbildung 6.8: RelaView - inhomogene Relation

Weiterhin ermöglicht die Auflistung eine Mehrfachauswahl von Eigenschaften, die beim weiterem Verändern der Matrix erzielt werden sollen. Wenn Eigenschaften ausgewählt sind, werden beim „Wegklicken“ eines Matrixelementes die ausgewählten Eigenschaften dazu veranlasst Elemente zu entfernen. Wenn Elemente der Matrix hinzugefügt werden, führt dies auch zum Hinzufügen von Elementen beim Ablauf der Vervollständigungsalgorithmen. Dadurch ergibt sich ein sehr angenehmes und intuitives Verhalten für die Veränderung der Relation. Abbildung 6.9 demonstriert das zuvor beschriebene Verhalten. Dort wurde die Eigenschaft *Equivalent* ausgewählt, und durch einen Klick auf ein Matrixelement erweitert.

Unter dem Menüpunkt *View* (Abbildung 6.10) finden sich alle Funktionen zur Veränderung der Ansicht von RelaView. Unter dem Menüpunkt *Show Properties* bzw. *Hide Properties* kann das Fenster der Eigenschaften angezeigt oder ausgeblendet werden. Die Menüpunkte *Matrix Zoom In*, *Matrix Zoom Out* und *Matrix Fit to Scene* verändern die Darstellung der Relation im Fenster, während *Equivalent Sort* eine Äquivalenzrelation in ihre Blockform überführt. *Reset Order* macht die Umordnung der Zeilen und Spalten wieder rückgängig.

Zur Demonstration des Verfahrens zur Sortierung einer Äquivalenzrelation wurde die Relation aus Abbildung 6.9 sortiert. Das Ergebnis ist in Abbildung 6.11 dargestellt.

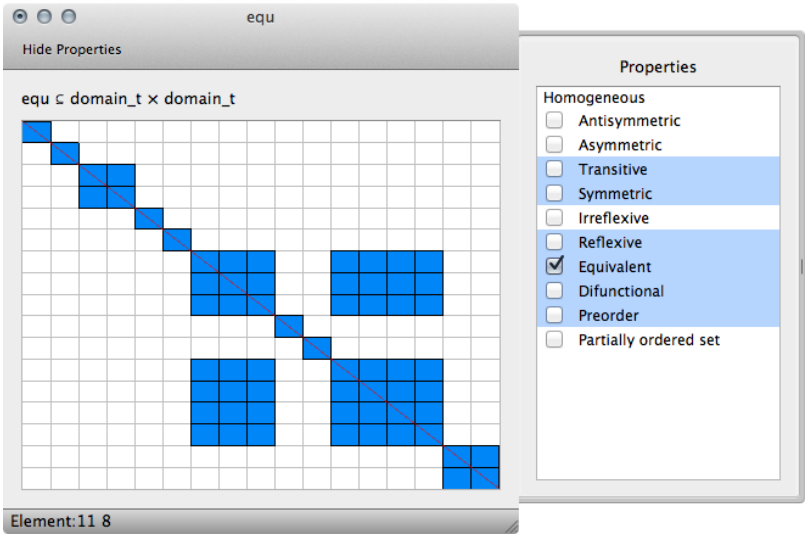

Abbildung 6.9: Relaview - Vervollständigung zur Äquivalenzrelation

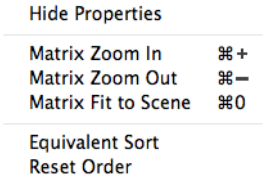

Abbildung 6.10: RelaView - View Drop-Down Menü

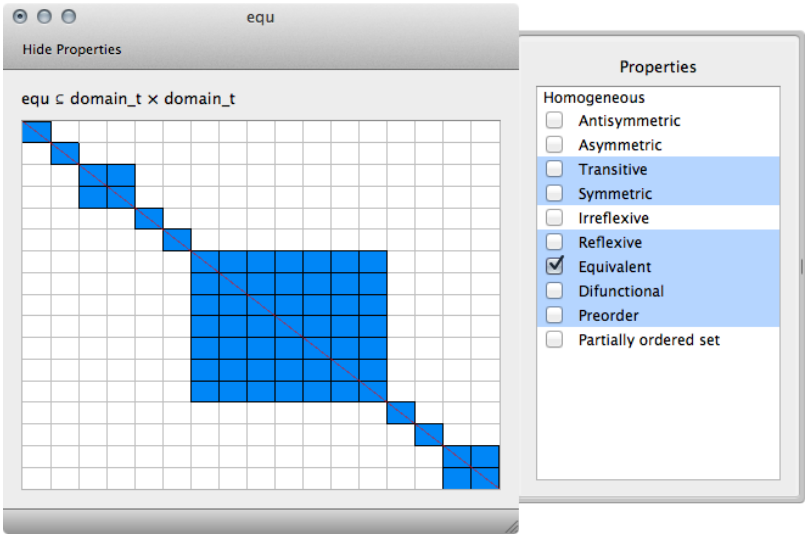

Abbildung 6.11: RelaView - Sortierte Äquivalenzrelation

Weiterhin existieren in RelaView zahlreiche weitere Funktionen. So können Relationen zum Beispiel als Bilddatei exportiert, mit einer Liste der Eigenschaften ausgedruckt oder wieder als RelaFix Code abgespeichert werden. Letzteres ist auch für Mengen möglich. Zusätzlich bietet RelaFix eine komfortable Möglichkeit Mengen und Relationen zu erstellen, wodurch der Einsatz eines zusätzlichen Editors unnötig wird. In Abbildung 6.12 wird der Dialog dargestellt, welcher das Anlegen neuer Mengen ermöglicht. Wahlweise können die Elemente durch Komma separiert angegeben oder von 0 aufsteigend bis zum angegebenen Wert durchnummeriert werden.

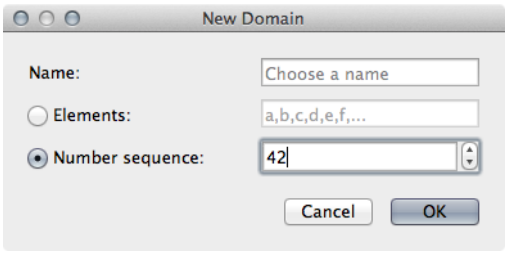

Abbildung 6.12: RelaView - Dialog zum Anlegen von Mengen

Neue Relationen werden durch den in Abbildung 6.13 dargestellten Dialog angelegt. Dieser ermöglicht es homogene oder inhomogene Relationen anzulegen und einen Startzustand festzulegen. Für inhomogene Relationen stehen die Nullrelation und die universelle Relation zur Verfügung. Homogene Relationen können darüber hinaus noch als Identitätsrelation sowie obere und untere Dreiecksmatrix angelegt werden. Als zugrundeliegende Mengen können alle zur Laufzeit des Programms erstellten sowie geladenen Mengen ausgewählt werden.

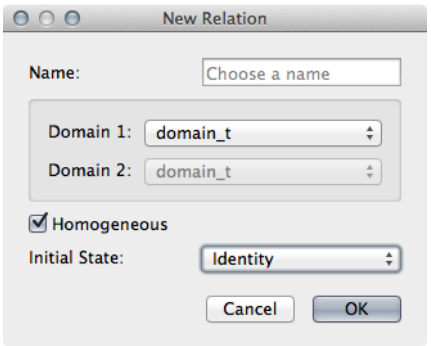

Abbildung 6.13: RelaView - Dialog zum Anlegen von Relationen

Kapitel 7

Ergebnisanalyse

Abschließend soll in diesem Kapitel analysiert werden, ob die erbrachten Ergebnisse den Anforderungen genügen. Anschließend wird betrachtet, in wie weit sich RelaView und RelaFix noch verbessern und erweitern lassen.

7.1 Abgleich der Anforderungen mit dem Ergebnis

Beim Abgleich der Anforderungen mit den demonstrierten Fähigkeiten von RelaView muss festgestellt werden, dass die Anforderungen zu einem großen Teil durch RelaView erfüllt werden. Des Weiteren kann aber auch festgestellt werden, dass einige Anforderungen nicht erfüllt wurden, welche nun hier genauer betrachtet werden.

Wie schon bereits in Abschnitt 5.5 erwähnt gilt Anforderung 2.1. als nicht erfüllt, da sich die `QGraphicsView` als nicht nicht performant genug herausgestellt hat, um auf herkömmlichen Notebooks mit mehr als 10.000 Elemente umgehen zu können.

Weiterhin gilt die manuelle Sortierung von Zeilen und Spalten (Anforderung 1.2.9.) als nicht erfüllt. Für die Realisierung dieser Funktion konnte kein zufriedenstellendes Konzept für die grafische Benutzeroberfläche gefunden werden, welche einfach und intuitiv zu bedienen gewesen wäre. Zwar existierten vielversprechende Ideen, der Aufwand um diese umzusetzen stand aber nicht im Verhältnis zum Nutzen und würde den zeitlichen Rahmen dieser Arbeit sprengen. Wenn zukünftig ein Konzept zur Verfügung stehen wird, ist eine Integration problemlos zu bewältigen, da die Mechanik zum Tauschen von Zeilen und Spalten bereits für das Sortierverfahren der Äquivalenzrelation implementiert wurde.

7.2 Fehler und Probleme

Während der Entwicklungsphase sind wie zu erwarten war zahlreiche Fehler aufgetaucht, die alle durch Assertions, Debugging und Code Analyse behoben oder umgangen werden konnten. Dennoch kann nicht ausgeschlossen werden, dass weitere Programmfehler in RelaView stecken. Neben Fehler in dem Qt Framework selbst, besteht die Möglichkeit, dass schwer zu erkennende Fehler oder Speicherlücken existieren. Letztere wurden versucht durch das Tool-Suite Valgrind (http://valgrind.org) auf ein Minimum zu reduzieren. Eine vollständige Abwesenheit von Speicherlücken kann, auch durch den Einsatz solcher Tools, nicht garantiert werden, da diese bestimmte Typen von Speicherfehler nicht finden können. Qt Fehler traten ebenfalls einige Male auf und wurden durch Fehlerberichte im Internet bestätigt. Die betroffenen Stellen sind auskommentiert und mit einem Link auf den Fehlerbericht sowie einer kurzen Beschreibung des Problems markiert. Die Fehler wurden soweit wie möglich umgangen. Doch war das leider nicht immer möglich. So existiert nach wie vor ein Fehler unter OSX, der das Eigenschaften-Fenster manchmal nicht schließt, wenn ein Hauptfenster geschlossen wird.

7.3 Ausblick

Wie bereits gezeigt wurde sind einige Teile von RelaView von vornherein so entwickelt worden, dass Erweiterungen in RelaFix - insbesondere Eigenschaften und Operationen - unkompliziert eingebunden werden können.

Es ist offensichtlich, dass in diesem Projekt noch ein großes Potential steckt und zahlreiche mögliche Erweiterungen das Programm zu einem nützlichen Tool für Lehre und Forschung werden lassen können.

Weiterhin soll im Anschluss an diese Arbeit die Entwicklung und Erweiterung von RelaView fortgesetzt werden, da die Überzeugung vorherrscht, dass es sich lohnt zusätzliche Zeit in ein solches Projekt zu investieren. Weiterhin ist es sehr wahrscheinlich, dass über die Zeit Fehler auftauchen werden, welche es zu beheben gilt.

Äußerst störend ist vor allem die schlechte Performance von RelaView im Umgang mit „größeren“ Relationen, welche es zu verbessern gilt. Es ist davon auszugehen, dass sich dieses Problem durch einen mittelgroßen Aufwand in den Griff bekommen lässt, auch wenn dafür zahlreiche Änderungen notwendig sind.

Neben Ausbesserungen von Fehlern und Erweiterungen um Eigenschaften würde eine Erweiterung der Fehlerprotokollierung eine sinnvolle Tätigkeit darstellen. Weiterhin wäre eine Einführung in die Nutzung des Programms bzw. Hilfedateien angebracht. Diese könnten so angelegt werden, dass auch themenfremde Nutzer einen guten Einstieg in die Materie erhalten. Inter-

essant wäre auch eine Erweiterung um eine LaTeX-Export Funktion, welche aus Relationen oder Mengen direkt LaTeX Abbildungen erstellt.

Des Weiteren ist eine manuelle Umordnung von Zeilen und Spalten eine nützliche Erweiterung, die möglichst bald integriert werden sollte.

Zudem gibt es in RelaView einige verwirrende Stellen. So zum Beispiel die Steuerung der Vervollständigungsalgorithmen durch den *Fill* Schalter im Menüpunkt *Properties*. Eine Konfiguration für jede einzelne Eigenschaft würde dieses Problem lösen, müsste aber übersichtlich dargestellt und integriert werden.

Schlussendlich wäre eine noch detailliertere Visualisierung von Relationen ein wünschenswertes Ziel. Es wäre zum Beispiel denkbar, die Äquivalenzklassen einer Äquivalenzrelation unterscheidbar darzustellen. Auch wäre es sicherlich hilfreich eine Tabelle einblenden zu können, welche die Definitionen der unterschiedlichen Eigenschaften und Operationen aufzeigt und es unter Umständen sogar ermöglicht, in RelaView Beispiele für diese laden zu lassen.

Kapitel 8

Fazit

In der vorliegenden Arbeit wurde die Konzeptionierung und Entwicklung einer freien, plattformübergreifenden Software zur Darstellung und Verarbeitung von Relationen dokumentiert, welche RelaView getauft wurde. Dazu wurden wesentliche Motivationsgründe aufgeführt und die Ziele der Entwicklung abgesteckt. Nach einem Überblick über die notwendigen mathematischen Grundlagen wurde das Applikations-Framework Qt vorgestellt, welches für die Entwicklung von RelaView verwendet wurde. Anschließend wurde die C-Bibliothek RelaFix vorgestellt, welche von RelaView zum Laden und Verarbeiten von Relationen und Mengen genutzt wird.

Weiterhin wurden detaillierte Anforderungen an RelaView entwickelt und besondere Teile der Implementierung von RelaView vorgestellt. Besonders hervorzuheben sind die entwickelten Algorithmen zum Vervollständigen von Relationseigenschaften, welche in die RelaFix Bibliothek integriert wurden. Des Weiteren wurde ein Verfahren zum Sortieren von Äquivalenzrelationen entwickelt und vorgestellt.

Weiterhin umfasst die vorliegende Arbeit eine Vorstellung der Funktionen von RelaView, welche durch Abbildungen der Applikation unterstützt wurde.

Abschließend wurde eine Ergebnisanalyse geführt, welche die gestellten Anforderungen mit den eigentlichen Ergebnissen gegenüberstellte sowie Probleme und Fehler von RelaView aufzeigte. Abgeschlossen wurde die Analyse durch einen Ausblick, welcher mögliche Erweiterungen von RelaFix und RelaView aufzeigte.

Literaturverzeichnis

[Behnke u. a. 1998] Behnke, Ralf ; Berghammer, Rudolf ; Meyer, Erich ; Schneider, Peter: RELVIEW — A system for calculating with relations and relational programming. In: Astesiano, Egidio (Hrsg.): *Fundamental Approaches to Software Engineering* Bd. 1382. Springer Berlin / Heidelberg, 1998, S. 318–321

[Berger 2012] Berger, Peter: *Entwicklung eines Programms und zugehöriger Sprache zur Analyse von Relationen*, Hochschule Bonn-Rhein-Sieg, Bachelor-Abschlussarbeit, 2012

[Berghammer 2008] Berghammer, R.: *Ordnungen, Verbände und Relationen mit Anwendungen.* Vieweg+Teubner Verlag, 2008

[Gamma u. a. 1996] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; John, Vlissides: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software.* 6. Auflage. Bonn : Addison-Wesley, 1996

[Müller 2012] Müller, Martin E.: *Relational Knowledge Discovery.* Cambridge University Press, 2012

[Nokia Corporation 2012a] Nokia Corporation: *Qt Library 4.8 Documentation.* http://qt-project.org/doc/qt-4.8/. Version: 2012, Abruf: 9.8.2012

[Nokia Corporation 2012b] Nokia Corporation: *Qt Library 4.8 Documentation - Object Model.* http://qt-project.org/doc/qt-4.8/object.html. Version: 2012, Abruf: 20.8.2012

[Nokia Corporation 2012c] Nokia Corporation: *Qt Library 4.8 Documentation - Signals and Slots.* http://qt-project.org/doc/qt-4.8/signalsandslots.html. Version: 2012, Abruf: 19.8.2012

[Schmidt 2010] Schmidt, Gunther: *Relational Mathematics.* Cambridge University Press, 2010 (Encyclopedia of Mathematics and Its Applications)

[Stephens u. Turkanis 2006] Stephens, D.R. ; Turkanis, J.: *C++ Kochbuch.* O'Reilly, 2006

[Wikimedia Foundation Inc. 2012] Wikimedia Foundation Inc.: *Qt Build-System*. http://de.wikipedia.org/wiki/Qt_(Bibliothek). Version: 2012, Abruf: 20.8.2012

[Witt 2010] Witt, Kurt-Ulrich: Mathematische Grundlagen für die Informatik. (2010)