Masterarbeit

CHASE und BACKCHASE:

Entwicklung eines Universal-Werkzeugs für eine Basistechnik der Datenbankforschung

eingereicht von: Martin Jurklies

eingereicht am: 10. September 2018

Gutachter: Prof. Dr. rer. nat. habil. Andreas Heuer
PD Dr.-Ing. habil. Meike Klettke

Zusammenfassung

Der CHASE-Algorithmus ist ein seit vielen Jahren in der Datenbanktheorie eingesetztes Verfahren, welches mitunter in den Bereichen der semantischen Optimierung von Anfragen, Reformulierung von Anfragen auf Sichten, Datenintegration, konzeptionellen Datenbankentwurf und Provenance-Management eingesetzt wird. Während es viele Tools gibt, die den CHASE in jeweils einem der genannten Bereiche umsetzen, existiert bislang keines, das den CHASE auf mehrere Bereiche anwendbar macht. Diese Arbeit stellt das Gerüst eines solchen Tools vor, das die Theorie des CHASE nahezu eins-zu-eins umsetzt und diese einfach anwendbar macht. Es kann den Standard-CHASE auf eine Datenbankinstanz mit Integritätsbedingungen anwenden und daraus eine Instanz erstellen, die die Integritätsbedingungen erfüllt. Ausgehend davon kann das Tool als Grundlage für die Anwendung des CHASE auf die verschiedenen Szenarien dienen.

Abstract

The CHASE-algorithm is used in database theory for many years and it finds application in different fields, e.g. semantic query optimization, query reformulation for using views, data integration, conceptional database design and provenance management. While there are many tools that implement the CHASE for one of these applications each, there doesn't exist one that make use of the CHASE for more of them. This thesis' objective is to develop such a potential tool that translates the theory of the CHASE nearly one-to-one and that makes it easily applicable. It can apply the Standard-CHASE on a database instance with integrity constraints and creates an instance satisfying the constraints. With that the tool can be a basis for the application of the CHASE in the different scenarios.

Inhaltsverzeichnis

1. Einleitung

In der Theorie wird der CHASE-Algorithmus als ein universal anwendbares Werkzeug beschrieben, das auf viele scheinbar verschiedene Probleme angewandt werden kann. Es gibt viele Tools, die den CHASE jeweils auf bestimmte Klassen von Problemen anwenden können und auf diese jeweils zugeschnitten haben. Jedoch ist aktuell kein Tool präsent, welches den CHASE als universalen Algorithmus für ebendiese verschiedenen Anwendungen einsetzt. Ziel dieser Arbeit ist es, ein solches universales Tool zu entwickeln, mit welchem der CHASE auf diese Anwendungen angewandt werden kann. Dabei spielt die Konzeption der Handhabung verschiedener den Anwendungen entsprechender Objekte eine Rolle, sodass dem Tool mitunter sowohl Datenbestände als auch Anfragen als Eingabe übergeben werden können. Dazu werden bereits bestehende Techniken und Tools des CHASE betrachtet und dabei abgewägt, in welchem Maße sich diese Tools als Basis für die Entwicklung eines solchen universalen Werkzeugs nutzen lassen können.

1.1. Einführung

Der CHASE-Algorithmus ist eine Grundtechnik in der Datenbankforschung, welche sich seit der Entwicklung und Veröffentlichung 1979 in [ABU79, MMS79] stetig weiterentwickelt hat. Die Idee des CHASE war dabei ursprünglich das Testen von Abhängigkeiten in Datenbanken beim Datenbankentwurf, wobei Tableaus das Modell dargestellt haben, auf dem hauptsächlich gearbeitet wurde. Mit gegebenen Integritätsbedingungen in Form von *funktionalen Abhängigkeiten* (FDs) und *Verbundabhängigkeiten* (JDs) wurden dann mithilfe des CHASE eventuell weitere Abhängigkeiten impliziert und gebildet. Mittlerweile hat der CHASE Anwendung in weiteren Bereichen wie der Anfrageoptimierung oder der Datenintegration gefunden, in denen dieser auf Datenbanken, Schemata oder Anfragen angewandt werden kann. Eine kompakte Übersicht zum CHASE wird in Tabelle 1.1 gegeben.

Idee: Der CHASE arbeitet eine Menge von Abhängigkeiten $\star$ in ein Objekt $\bigcirc$ ein, sodass $\star$ implizit in $\bigcirc$ enthalten ist:

$$\text{chase}_{\star}(\bigcirc) = \circledast$$

	$\star$	$\bigcirc$	Ergebnis	Ziel	Tool
0.	Abhängigkeiten	DB-Schema[1]	DB-Schema mit Integritätsbedingungen	optimierter DB-Entwurf	
I.	Abhängigkeiten	Anfragen	Anfragen	Semantische Optimierung	PDQ
II.	Sichten	Anfragen	Anfragen auf Sichten	AQuV	ProvCB
II'.	Operationen	Anfragen	Anfragen auf Operationen	AQuO	
III.	s-t tgd's, egd's, tgd's	Quell-DB	Ziel-DB	Datenaustausch, Datenintegration	Llunatic, ChaseFUN
IV.	tgd's, egd's	DB	modifizierte DB	Cleaning	Llunatic, ChaseFUN
V.	tgd's, egd's	unvollständige DB	Anfrageergebnis	sichere Antworten	
VI.	s-t tgd's, egd's, tgd's	DB	Anfrageergebnis	invertierbare Auswertung	

Tabelle 1.1. Überblick von CHASE-Anwendungen und zugehörigen Parametern

Die Punkte I., II., III. und VI. sind dabei die für das Ziel dieser Arbeit interessanten Anwendungen, wobei III. eher als Vorbild für VI. dient.

[1] PJ-Mapping

Es haben sich neben verschiedenen Anwendungsbereichen auch unterschiedliche Varianten des CHASE ergeben, die jeweils ein unterschiedliches Verhalten in der Terminierung, Effizienz und Lösungsgröße aufweisen. Diese werden hier kurz beschrieben und später in Abschnitt 2.3 genauer behandelt.
Der *Standard-CHASE* ist die Variante des CHASE, um die es in dieser Arbeit hauptsächlich gehen wird. Dieser führt sogenannte CHASE-Schritte aus. Ein solcher CHASE-Schritt ist das Durchführen von später genauer erklärten Maßnahmen, die dafür sorgen, dass eine Datenbankinstanz oder eine Anfrage bestimmten mitangegeben Integritätsbedingungen genügt. Den Standard-CHASE zeichnet aus, dass auf die Notwendigkeit der Ausführung eines CHASE-Schrittes vorher geprüft wird.
Der *Core-CHASE* führt die CHASE-Schritte parallel aus und reduziert das Ergebnis anschließend auf den Kern.
Der *Oblivious CHASE*, der sich in *Naive Oblivious CHASE* und *Skolem-Oblivious CHASE* unterteilt, hat die Überprüfung auf die Notwendigkeit eines CHASE-Schrittes nicht. Das hat zur Folge, dass der Oblivious CHASE eventuell nicht terminiert, da CHASE-Schritten wiederholt angewendet werden können, um bestimmte Integritätsbedingungen zu erfüllen, obwohl die Instanz oder Anfrage diesen bereits genügen. Der Skolem-Oblivious CHASE umgeht dieses Problem, indem sogenannte *Skolemfunktionen* stellvertretend für generierende Variablen oder Nullwerte eingesetzt werden, deren Funktionswerte von bereits gegebenen Werten abhängig sind. Auf diese Weise werden nicht wiederholt neue Tupel oder Teilanfragen mit völlig neuen Variablen oder Nullwerten generiert, sondern durch die Skolemfunktion eindeutig gemacht.
Die Terminierung des CHASE ist also in bestimmten Fällen und Varianten nicht gewiss. Es ist sogar unentscheidbar zu bestimmen, ob der CHASE bei einer gegebenen Abhängigkeitenmenge für alle Instanzen terminiert [DNR08]. Jedoch gibt es Situationen und Bedingungen, bei denen diese Problematik entscheidbar wird oder zumindest mit einer größeren Sicherheit bestimmt werden kann, dass der CHASE terminieren wird. Eine Ordnung für die Terminierung der einzelnen vorgestellten CHASE-Varianten sieht etwa wie folgt aus:

$$\text{Naive Oblivious CHASE} \preceq \text{Skolem-Oblivious CHASE} \preceq \text{Standard-CHASE} \preceq \text{Core-CHASE}$$

Einige Kriterien für eine Terminierung sind beispielsweise schwache Azyklizität von tgd's [FKMP05, GMS12], Hierarchisierung (Stratification) [GMS12], Sicherheit (Safety) [GMS12]. Die genannten Kriterien zur Terminierung des CHASE dienen hier allerdings nur der Übersicht. Eine Erklärung dieser befindet sich in [GMS12].
Bestimmte Eigenschaften des CHASE sind nur in bestimmten Fällen gegeben. So ist die *Konfluenz* nur gegeben, wenn es sich bei den Integritätsbedingungen um FDs oder JDs handelt. Bei den allgemeineren Bedingungen wie tgd's und egd's ist der CHASE jedoch nicht konfluent. Konfluenz sagt aus, dass immer dasselbe Ergebnis erreicht wird unabhängig von der Reihenfolge der angewandten Schritte, um auf das Ergebnis zu kommen.
Die in Tabelle 1.1 enthaltenen Anwendungen, umgesetzten Tools und darunter aufgelisteten CHASE-Varianten werden im Laufe der Arbeit noch genauer behandelt. Zunächst einmal folgt nach der Aufgabenstellung dieser Arbeit der Beispieldatensatz, welcher in Beispielen zu bestimmten Definitionen in Kapitel 2 und im Beispielprogramm des hier entwickelten CHASE-Tools Anwendung findet.

1.2. Aufgabenstellung

Für die Anwendungsprobleme I., II., III. und VI. in Tabelle 1.1 soll ein grundlegendes CHASE- und BACKCHASE-Tool konzipiert und prototypisch implementiert werden. Dem Tool sollen sowohl Datenbestände als auch Anfragen übergeben werden können. Die Schnittstellen sollen sich dabei an relationale

Tabellen und ein Kern-SQL anlehnen. Für die Anwendungsprobleme II., III. und VI. wird dem CHASE-Prozess noch ein zweiter CHASE-Prozess, der sogenannte BACKCHASE-Prozess, hinzugefügt, der die Ermittlung der gesuchten Ergebnisanfrage erst ermöglicht. Es ist möglich, dass sich für die Implementierung auf ein Teilproblem in Form von einfachen Daten, einfachen Korrespondenzen oder einfachen Anfragen beschränkt werden kann.

1.3. Beispieldatensatz

Der Beispieldatensatz entstammt aus [Aug17] und besteht aus fünf Schemata mit zugehörigen Relationen. Der hier in der Einleitung verwendete Datensatz ist dabei nur ein Ausschnitt des vollständigen Satzes, welcher in Anhang A zu finden ist.

Der Datensatz stellt eine fiktive Datenbank des Vorlesungsangebots des Instituts für Informatik und Elektrotechnik der Universität Rostock dar. Er besteht aus Tabellen für angebotene Vorlesungen, teilnehmende Studenten, eingetragene Prüfungsnoten sowie die Zugehörigkeit der Vorlesungen zu bestimmten Modulen und haltenden Dozenten.

Jede Tabelle enthält eine ID-Spalte, um die darin enthaltenen Tupel eindeutig zu identifizieren und direkt referenzieren zu können. Die Domänen einiger Attribute sind Enumerationen, d.h., sie sind nur auf bestimmte Werte eingeschränkt wie beispielsweise die Noten oder Semester. Andere Domänen wurden so für das Verständnis des Beispiels eingeschränkt, dass diese hier endlich sind, jedoch in einer realen Anwendung unendlich sein können. So können beispielsweise die Namen von Studenten weitestgehend beliebige Strings oder die Modulnummern wie die Matrikelnummern natürliche Zahlen sein.

Der Beispieldatensatz ist nun folgendermaßen aufgestellt:

- STUDENTEN = { $\text{ID}_{\text{Studenten}}$, Matrikelnummer, Name, Vorname, Studiengang }
  - dom($\text{ID}_{\text{Studenten}}$) := { $S_i \mid i \in \mathbb{N}$ und fortlaufend nummeriert }
  - dom(Matrikelnummer) := $\mathbb{N}$ = { 1, 2, 3, ... }
  - dom(Name) := { Fieber, Sonnenschein, Mueller, Johansen, Miller, Mustermann, Johannes }
  - dom(Vorname) := { Fabian, Sarah, Max, Mira, Johannes, Mia, Paul }
  - dom(Studiengang) := { Informatik, Mathematik, Elektrotechnik, ITTI }
- NOTEN = { ID_{Noten}, Modulnummer, Matrikelnummer, Semester, Note }
  - dom(ID_{Noten}) := { $N_i \mid i \in \mathbb{N}$ und fortlaufend nummeriert }
  - dom(Modulnummer) := { $xyz \mid xyz$ fortlaufend nummeriert mit $x, y, z \in \{0, ..., 9\}$ und $xyz \neq 000$ }
  - dom(Matrikelnummer) := $\mathbb{N}$
  - dom(Semester) := { WS 14/15, SS 15, WS 15/16, SS 16, WS 16/17, SS 17 }
  - dom(Note) := { 1.0, 1.3, 1.7, 2.0, 2.3, 2.7, 3.0, 3.3, 3.7, 4.0, 5.0 }
- MODULE = { $\text{ID}_{\text{Module}}$, Modulnummer, Titel, Vertiefung }

  - dom($\text{ID}_{\text{Module}}$) := { $M_i \mid i \in \mathbb{N}$ und fortlaufend nummeriert }
  - dom(Modulnummer) := { $xyz \mid xyz$ fortlaufend nummeriert mit $x, y, z \in \{0, ..., 9\}$ und $xyz \neq 000$ }
  - dom(Titel) := { DBIII, Mathematik 1, MeKa, Individuelles Wissensmanagement, DWBI, Kognitive Systeme, TRDB, Systembiologie, NEidI }
  - dom(Vertiefung) := { Informationssysteme, Modelle und Algorithmen, Smart Computing, Visual Computing, Wirtschaftsinformatik, xxx }

- DOZENTEN = { $\text{ID}_{\text{Dozenten}}$, Modulnummer, Dozent }
  - dom($\text{ID}_{\text{Dozenten}}$) := { $D_i \mid i \in \mathbb{N}$ und fortlaufend nummeriert }
  - dom(Modulnummer) := { $xyz \mid xyz$ fortlaufend nummeriert mit $x, y, z \in \{0, ..., 9\}$ und $xyz \neq 000$ }
  - dom(Dozent) := { Dozent A, Dozent B, Professor A, Professor B, Professor C, Professor D, Professor E }
- TEILNEHMER = { $\text{ID}_{\text{Teilnehmer}}$, Modulnummer, Matrikelnummer }
  - dom($\text{ID}_{\text{Teilnehmer}}$) := { $T_i \mid i \in \mathbb{N}$ und fortlaufend nummeriert }
  - dom(Modulnummer) := { $xyz \mid xyz$ fortlaufend nummeriert mit $x, y, z \in \{0, ..., 9\}$ und $xyz \neq 000$ }
  - dom(Matrikelnummer) := $\mathbb{N}$

Die Tabellen sind folgendermaßen aufgebaut:

$\text{ID}_{\text{Studenten}}$	Matrikelnummer	Name	Vorname	Studiengang
S_1	1	Fieber	Fabian	Lehramt Informatik
S_2	2	Sonnenschein	Sarah	Mathematik
S_3	3	Müller	Max	Elektrotechnik
S_4	4	Müller	Mira	Informatik
S_5	5	Johansen	Johannes	Informatik
S_6	6	Miller	Mia	Informatik
S_7	7	Mustermann	Max	Elektrotechnik
S_8	8	Joahnnes	Paul	ITTI

Tabelle 1.2. Beispielrelation von STUDENTEN

ID_{Noten}	Modulnummer	Matrikelnummer	Semester	Note
N_2	001	2	SS 16	1.7
N_6	002	2	WS 14/15	1.3
N_{14}	004	2	WS 16/17	3.0

Tabelle 1.3. Beispielrelation von NOTEN aus Tabelle A.2 eingeschränkt auf Matrikelnummer 2

1.4. Aufbau der Arbeit

Es werden im Folgenden zuerst für diese Arbeit notwendige grundlegende Begriffe definiert. Angefangen wird mit Begriffen, die auch schon bereits aus Vorlesungen der Universität Rostock bekannt sind. Danach folgen welche, die für das Verständnis des CHASE-Algorithmus' notwendig sind.
Im darauffolgenden Kapitel werden der aktuelle Stand der Forschung betrachtet und bereits bestehende

$\mathbf{ID_{Module}}$	**Modulnummer**	**Titel**	**Vertiefung**
M_1	001	DBIII	Informationssysteme
M_2	002	Mathematik 1	xxx
M_3	003	MeKa	Smart Computing
M_4	004	Individuellesa Wissensmanagement	Informationssysteme
M_5	005	DWBI	Wirtschaftsinformatik
M_6	006	Kognitive Systeme	Smart Computing
M_7	007	TRDB	Informationssysteme
M_8	008	Systembiologie	Modelle und Algorithmen
M_9	009	NEidI	xxx

Tabelle 1.4. Beispielrelation von MODULE

$\mathbf{ID_{Dozenten}}$	**Modulnummer**	**Dozent**
$D_{1.1}$	001	Professor A
$D_{1.2}$	001	Dozent A
D_2	002	Professor B
D_3	003	Professor C
D_4	004	Professor D
D_5	005	Dozent B
D_6	006	Professor D
D_7	007	Professor A
D_8	008	Professor E
D_9	009	Professor A

Tabelle 1.5. Beispielrelation von DOZENTEN

$\mathbf{ID_{Teilnehmer}}$	**Modulnummer**	**Matrikelnummer**
T_2	001	2
T_6	002	2

Tabelle 1.6. Beispielrelation von TEILNEHMER aus Tabelle A.5 eingeschränkt auf Matrikelnummer 2

Techniken und Tools für den CHASE kurz vorgestellt. Am Ende des Kapitels wird dann entschieden, ob und welche Tools sich für eine Weiterentwicklung und Anpassung für eine weitergehende Anwendung des CHASE auf mehrere Probleme eignen. Anschließend werden das Konzept und die Implementierung des entwickelten CHASE-Tools vorgestellt. Es wird darauf eingegangen, wie die Theorie und die vorgestellten Grundlagen in das Tool Einzug gefunden haben und warum diese auf diese Weise umgesetzt wurden.
Im Fazit wird schließlich zusammengefasst, was umgesetzt werden konnte und welche Probleme noch bestehen bzw. mit welchen das Tool noch nicht umgehen kann.

2. Begriffe und Definitionen

In diesem Kapitel werden Begriffe definiert, die für das Verständnis der Arbeit notwendig sind. Viele der Definitionen entsprechen dabei weitestgehend denen, die in sonstiger Literatur aus diesem Gebiet vorkommen [Heu17, BKM+17, GMS12]. Als Grundlage wird zuerst das Relationenmodell definiert, wozu Begriffe der Schemata, Instanzen und Integritätsbedingungen gehören. Anschließend werden benötigte Operationen des Relationenmodells vorgestellt. Die Begriffe davon stammen dabei aus [Heu17]. In Abschnitt 2.3 wird schließlich alles relevante zum CHASE-Algorithmus geklärt. Einige Begriffe und Symbole aus diesen Definitionen, welche überwiegend aus [BKM+17, GMS12] stammen, wurden allerdings so abgeändert, dass diese der Verwendung des Lehrstuhls DBIS der Universität Rostock entsprechen.

2.1. Relationenmodell

2.1.1. Schemata und Instanzen

Definition 2.1 (Relationenschema)**.** Seien $\mathcal{A}$ eine Menge von **Attributen** und $\mathcal{R}$ eine Menge von **Relationensymbolen** oder **-namen**. Dann ist $R := \{A_1, ..., A_n\}$ mit $R \in \mathcal{R}$ und $A_i \in \mathcal{A}$ für alle $1 \leq i \leq n$ ein **Relationenschema**.

Definition 2.2 (Domäne)**.** Eine **Domäne** $\mathrm{dom}(A)$ eines Attributes A ist eine Abbildung $\mathrm{dom}\colon \mathcal{A} \to \mathit{Const}$, wobei Const eine Menge von Konstanten ist, und bezeichnet den Wertebereich von A.

Definition 2.3 (Relation)**.** Sei R ein Relationenschema mit den Attributen $A_1, ..., A_n$. Dann heißt $r(R)$ **Relation** über R und ist definiert als eine Menge von totaldefinierten Abbildungen

$$t\colon R \to \bigcup_{A \in R} \mathrm{dom}(A)$$

mit $t(A) \in \mathrm{dom}(A)$, die als **Tupel** bezeichnet werden. Für eine Menge $X \subseteq R$ von Attributen ist die zugehörige Domäne entsprechend der Vereinigung der Domänen aller Attribute von X und wird als X-**Wert** eines Tupels t mit $t|_X$ bezeichnet.

Definition 2.4 (Datenbankschema)**.** Eine Menge von Relationenschemata $S := \{R_1, ..., R_n\}$ mit $n \in \mathbb{N}$ wird als **Datenbankschema** bezeichnet.

Definition 2.5 (Datenbank(-instanz))**.** Sei S ein Datenbankschema mit den Relationenschemata $R_1, ..., R_n$. Dann ist $d(S)$ eine **Datenbank** über S und bezeichne die Menge von Relationen $d := \{r_1, ..., r_n\}$, wobei $r_i(R_i)$ für alle $1 \leq i \leq n$ gilt.

Beispiel 2.1. *Für die eben definierten Begriffe wäre der Beispieldatensatz aus Tabelle 1.2 eine Relation mit*

$$\mathit{STUDENTEN} = \{\mathit{ID}_{\mathit{Studenten}}, \mathit{Matrikelnummer}, \mathit{Name}, \mathit{Vorname}, \mathit{Studiengang}\}$$

als Relationenschema. Die Tupel von STUDENTEN entsprechen den Zeilen von Tabelle 1.2. Eines davon wäre beispielsweise $t_2(ID_{Studenten}) = S_2$, $t_2(Matrikelnummer) = 2$, $t_2(Name) =$ *Sonnenschein,* $t_2(Vorname) =$ *Sarah und* $t_2(Studiengang) =$ *Mathematik. Mit den restlichen Tabellen zusammen, die jeweils ebenfalls Relationen darstellen, ergäbe dies eine Datenbank mit dem Datenbankschema*

$$S = \{STUDENTEN, NOTEN, MODULE, DOZENTEN, TEILNEHMER\}.$$

2.1.2. Integritätsbedingungen

Als Integritätsbedingungen werden in der Datenbanktheorie all jene Bedingungen bezeichnet, die erfüllt sein müssen, damit Datenbanksysteme konsistent sind. Das heißt, alle in der Datenbank gespeicherten Daten sind korrekt. Falls Aktionen ausgeführt werden sollen, die eine Integritätsbedingung verletzen würden, wird die Ausführung einer solchen Aktion vom Datenbanksystem verhindert.
Die gängigsten und häufigsten Integritätsbedingungen sind *Schlüssel*, die sich in Primärschlüssel und Fremdschlüssel unterteilen, und verschiedene Arten von Abhängigkeiten zwischen Attributen wie etwa *funktionale Abhängigkeiten*, *Verbundabhängigkeiten* oder *mehrwertige Abhängigkeiten*. Allgemeiner aufgefasst sind das *tuple generating dependencies* und *equality generating dependencies*, welche in Abschnitt 2.3 für den CHASE-Algorithmus definiert werden. Operationen des Relationenmodells, die für die Definitionen in diesem Abschnitt gebraucht werden, sind in Abschnitt 2.2 zu finden.

Definition 2.6 (Schlüssel)**.** Für eine Relation $r(R)$ heißt eine Menge $K := \{B_1, ..., B_n\} \subseteq R$ **identifizierende Attributmenge** mit

$$\forall t_1, t_2 \in r[t_1 \neq t_2 \Rightarrow \exists B \in K : t_1(B) \neq t_2(B)].$$

Ein **Schlüssel** ist bzgl. $\subseteq$ eine minimale, identifizierende Attributmenge und jedes Attribut eines Schlüssels ist ein **Primattribut**.

Beispiel 2.2. *Für die STUDENTEN-Relation wäre ein Schlüssel die "Matrikelnummer". In den Beispielrelationen sind alle Attributnamen, die ein Schlüssel für die jeweilige Relation darstellen, unterstrichen. Bei mehreren unterstrichenen Attributnamen bilden diese zusammen den Schlüssel.*

Definition 2.7 (Funktionale Abhängigkeit/ Functional Dependency)**.** Eine **funktionale Abhängigkeit** (**FD**) ist für eine Relation $r(R)$ mit $X, Y \subseteq R$ ein Ausdruck der Form $X \to Y$. Das heißt, die Attributmenge X bestimmt eindeutig die Attributmenge Y. Die Relation r genügt einer FD $X \to Y$ also genau dann, wenn $|\pi_Y(\sigma_{X=c}(r))| \leq 1$ für alle $c \in \text{dom}(X)$ gilt.

Beispiel 2.3. *Eine FD wäre "Titel* $\to$ *Vertiefung" in der MODULE-Relation. Jedes Attribut von "Titel" bestimmt eindeutig die Vertiefung, zu der es gehört.*

Definition 2.8 (Verbundabhängigkeit/ Join Dependency)**.** Eine **Verbundabhängigkeit** (**JD**) ist für ein Datenbankschema $S = \{R_1, ..., R_n\}$ über $\mathcal{U} := \bigcup_{1 \leq i \leq n} R_i$ ein Ausdruck der Form $\bowtie [R_1, ..., R_n]$. Eine Relation $r(\mathcal{U})$ genügt einer JD $\bowtie [R_1, ..., R_n]$ genau dann, wenn

$$r = \pi_{R_1}(r) \bowtie ... \bowtie \pi_{R_n}(r)$$

gilt.

Beispiel 2.4. *Die Relation r aus Tabelle 2.2 genügt der JD*

$$\bowtie [(Modulnummer, Semester), (Modulnummer, Dozent)] :$$

Modulnummer	Semester
001	SS 16
002	WS 15/16
003	WS 16/17
004	WS 16/17
005	SS 17
006	SS 17
007	SS 16
008	
009	SS 15
009	SS 16

Modulnummer	Dozent
001	Professor A
001	Dozent A
002	Professor B
003	Professor C
004	Professor D
005	Dozent B
006	Professor D
007	Professor A
008	Professor E
009	Professor A

Tabelle 2.1. $\pi_{\{\text{Modulnummer, Semester}\}}(r)$ (links)
$\pi_{\{\text{Modulnummer, Dozent}\}}(r)$ (rechts)

Modulnummer	Semester	Dozent
001	SS 16	Professor A
001	SS 16	Dozent A
002	WS 15/16	Professor B
003	WS 16/17	Professor C
004	WS 16/17	Professor D
005	SS 17	Dozent B
006	SS 17	Professor D
007	SS 16	Professor A
008		Professor E
009	SS 15	Professor A
009	SS 16	Professor A

Tabelle 2.2. Ergebnis r von $\pi_{\{\text{Modulnummer, Semester}\}}(r) \bowtie \pi_{\{\text{Modulnummer, Dozent}\}}(r)$

Definition 2.9 (Mehrwertige Abhängigkeit/ Multivalued Dependency)**.** Eine **mehrwertige Abhängigkeit** (**MVD**) ist für eine Relation $r(R)$ mit $X, Y \subseteq R, Z := R - (X \cup Y)$ ein Ausdruck der Form $X \twoheadrightarrow Y$. Die Relation r genügt einer MVD $X \twoheadrightarrow Y$ genau dann, wenn

$$\forall t_1, t_2 \in r[(t_1 \neq t_2 \wedge t_1(X) = t_2(X)) \Rightarrow \exists t_3 \in r\colon t_3(X) = t_1(X) \wedge t_3(Y) = t_1(Y) \wedge t_3(Z) = t_2(Z)]$$

gilt.

Definition 2.10 (Inklusionsabhängigkeit/ Inclusion Dependency)**.** Eine **Inklusionsabhängigkeit** (**ID**) ist für Relationen $r_1(R_1), r_2(R_2) \in d(S), X \subseteq R_1, Y \subseteq R_2$ ein Ausdruck der Form $R_1[X] \subseteq R_2[Y]$. Die Datenbank d genügt einer IND $R_1[X] \subseteq R_2[Y]$ genau dann, wenn

$$\pi_X(r_1) \subseteq \pi_Y(r_2)$$

gilt.

Bemerkung. Falls eine Instanz I oder Relation r einer Menge von Abhängigkeiten $\mathcal{B}$ genügt, wird dies mit $I \vDash \mathcal{B}$ oder $r \vDash \mathcal{B}$ ausgedrückt. Analog gilt dies für einzelne Abhängigkeiten $b \in \mathcal{B}$.

2.2. Operationen des Relationenmodells

Definition 2.11 (Projektion)**.** Die **Projektion** π einer Relation $r(R)$ auf $X \subseteq R$ wählt die X-Werte von Tupeln t aus r aus, die im Index des Projektionsoperators angegebenen sind. Formal ausgedrückt:

$$\pi_X(r) := \{t(X) \mid t \in r\}.$$

Beispiel 2.5. *Die Projektion angewandt auf die STUDENTEN-Relation auf "Studiengang" ergibt*

$$\pi_{Studiengang}(STUDENTEN) = \{(Lehramt\ Informatik), (Mathematik), (Elektrotechnik), (Informatik), (ITTI)\}.$$

Definition 2.12 (Selektion)**.** Die **Konstanten-Selektion** von $r(R)$ nach $X\theta c$ mit $X \subseteq R$, c X-Wert und $\theta \in \{<, \leq, =, \geq, >, \neq\}$ wählt die Tupel t aus r aus, deren X-Werte durch c unter einem angegebenen Vergleichsoperator θ eingeschränkt ist. Formal ausgedrückt:

$$\sigma_{X\theta c}(r) := \{t \mid t \in r \wedge t(X)\theta c\}.$$

Die **Attribut-Selektion** von $r(R)$ nach $X\theta Y$ mit $X, Y \subseteq R$ und $\theta \in \{<, \leq, =, \geq, >, \neq\}$ ist definiert als

$$\sigma_{X\theta c}(r) := \{t \mid t \in r \wedge t(X)\theta t(Y)\}.$$

Eine **Selektion** ist eine Konstanten-Selektion oder eine Attribut-Selektion. Für beide Selektionen gilt, dass $\theta \in \{=, \neq\}$, falls die Menge der X-Werte und Y-Werte nicht linear geordnet ist. Mehrere Selektionen können zu allgemeinen Selektionsbedingungen mittels $\wedge$, $\vee$, $\neg$ oder Klammerung (und) formuliert werden.

Beispiel 2.6. *Tabelle 1.3 ist die Konstanten-Selektion von Tabelle A.2 nach "Matrikelnummer = 2".*

Definition 2.13 (Verbund)**.** Der **natürliche Verbund** (kurz: **Verbund**) von $r_1(R_1)$ und $r_2(R_2)$ ist definiert als

$$r_1 \bowtie r_2 := \{t \mid t \text{ ist Tupel über } R_1 \cup R_2 \wedge [\forall i \in \{1, 2\} \exists t_i \in r_i \colon t_i = t(R_i)]\}.$$

Im Falle von $R_1 = R_2$ wird $\bowtie$ zum mengentheoretischen Durchschnitt und im Falle von $R_1 \cap R_2 = \{\}$ wird $\bowtie$ zum **kartesischen Produkt**.

Definition 2.14 (Mengenoperationen)**.** Die Mengenoperationen **Vereinigung**, **Differenz** und **Durchschnitt** sind für $r_1(R)$ und $r_2(R)$ definiert als

$$r_1 \cup r_2 := \{t \mid t \in r_1 \vee t \in r_2\}$$
$$r_1 - r_2 := \{t \mid t \in r_1 \wedge t \notin r_2\}$$
$$r_1 \cap r_2 := \{t \mid t \in r_1 \wedge t \in r_2\}.$$

Definition 2.15 (Umbenennung)**.** Die **Umbenennung** β von einem Attribut A zu einem Attribut B in $r(R)$ ist für $A \in R$, $B \notin R - \{A\}$, $R' := (R - \{A\}) \cup \{B\}$, $\mathrm{dom}(A) = \mathrm{dom}(B)$ definiert als

$$\beta_{B \leftarrow A}(r) := \{t' \mid \exists t \in r \colon t'(R - \{A\}) = t(R - \{A\}) \wedge t'(B) = t(A)\}.$$

2.3. CHASE-Algorithmus

Definition 2.16 (Konstanten, Variablen, Nullwerte)**.** Bezeichne *Const* eine unendlich abzählbare Menge von **Konstanten**, *Null* eine unendlich abzählbare Menge von markierten **Nullwerten** und *Var* eine

unendlich abzählbare Menge von **Variablen**.

Definition 2.17 (Werte und Terme)**.** Ein **Wert** bezeichnet ein Element aus *Const* oder ein Element aus *Null*. Ein **Term** bezeichnet einen **Wert** oder ein Element aus *Var*.

Definition 2.18 (erweitertes Tupel)**.** Ein **erweitetes Tupel** ist ein Ausdruck der Form $\mathbf{t} = (t_1, ..., t_n)$, wobei die t_i Terme für alle $1 \leq i \leq n$ sind. Der Einfachheit halber wird $\mathbf{t}$ auch als Menge von Termen betrachtet, weshalb $t \in \mathbf{t}$ eine korrekte Schreibweise darstellt, falls t in $\mathbf{t}$ vorkommt.

Definition 2.19 (Atomare Formel/ Atom)**.** Ein **relationales Atom** ist eine Ausdruck der Form $R(\mathbf{t})$, wobei $R \in \mathcal{R}$ ist, $t_i \in \mathbf{t}$ Terme sind für alle $1 \leq i \leq n$ und $n \in \mathbb{N}$ die Stelligkeit von R ist.
Ein **Gleichheitsatom** ist ein Ausdruck der Form $t_1 = t_2$, wobei t_1 und t_2 Terme sind.

Bemerkung. Relationale Atome werden auch repräsentativ für Relationenschemata genutzt.

Definition 2.20 (Fakt)**.** Ein **Fakt** ist ein Atom ohne Variablen.

Definition 2.21 (erweiterte Instanz)**.** Eine **erweiterte Instanz** I ist eine Menge relationaler Fakten $R(\mathbf{t}) \in I$ oder $\mathbf{t} \in I(R)$.

Bemerkung. Erweiterte Tupel und Instanzen können im Gegensatz zu den vorigen Definitionen von Tupeln und Instanzen Nullwerte enthalten. Da für die Arbeit mit dem CHASE die erweiterten Tupel und Instanzen benötigt werden, werden für den Rest dieser Arbeit nur noch diese verwendet. Außerdem werden ab sofort die erweiterten Tupel und Instanzen der Einfachheit halber nur noch "Tupel" und "Instanzen" genannt.

Beispiel 2.7. *Es wird wieder der Beispieldatensatz aus Kapitel 1 betrachtet. Relationale Atome, welche zugleich Fakten sind, wären beispielsweise*

$$STUDENTEN(S_6, 6, Miller, Mia, Informatik),$$
$$NOTEN(N_{14}, 004, 2, WS\ 16/17, 3.0).$$

Definition 2.22 (tuple generating dependency)**.** Eine **tuple generating dependency** (**tgd**) ist ein Ausdruck der Form

$$\forall \mathbf{x}\colon (\varphi(\mathbf{x}) \rightarrow \exists \mathbf{y}\colon \psi(\mathbf{x}, \mathbf{y})),$$

wobei $\varphi(\mathbf{x})$ und $\psi(\mathbf{x}, \mathbf{y})$ Konjunktionen von relationalen Atomen sind, sodass jede Variable $\mathbf{x}$ in mindestens einem der Atome in $\varphi(\mathbf{x})$ und jede der Variablen in $\mathbf{x}$ und $\mathbf{y}$ in mindestens einem der Atome in $\psi(\mathbf{x}, \mathbf{y})$ vorkommt.
Eine **source-to-target tuple generating dependency** (**s-t tgd**) ist eine tgd, wobei $\varphi(\mathbf{x})$ eine Konjunktion von atomaren Formeln über einem Quellschema S und $\psi(\mathbf{x}, \mathbf{y})$ eine Konjunktion von atomaren Formeln über einem Zielschema T ist.
Der Teil φ wird auch als **Rumpf** und ψ als **Kopf** der tgd bezeichnet.
Für tgd's und s-t tgd's ist keines der Atome ist ein Gleichheitsatom.

Definition 2.23 (equality generating dependency)**.** Eine **equality generating dependency** (**egd**) ist ein Ausdruck der Form

$$\forall \mathbf{x}\colon (\varphi(\mathbf{x}) \rightarrow (x_1 = x_2)),$$

wobei $\varphi(\mathbf{x})$ eine Konjunktion von relationalen Atomen ist, sodass jede Variable x in mindestens einem der Atome in $\varphi(\mathbf{x})$ vorkommt, und x_1 und x_2 Variablen aus $\mathbf{x}$ sind.
Der Teil φ wird auch als **Rumpf** und $(x_1 = x_2)$ als Kopf der egd bezeichnet.
Alle Atome des Kopfes sind Gleichheitsatome.

Bemerkung. Obwohl in der Definition hier und in der meisten Literatur der Kopf einer egd nur als ein Gleichheitsatom dargestellt wird, werden hier auch egd's mit einer Konjunktion von Gleichheitsatomen im Kopf verwendet.

Bemerkung. FD's und IND's sind Spezialfälle von egd's und JD's und MVD's sind Spezialfälle von tgd's.

Bemerkung. Variablen werden unterteilt in zwei Typen von Variablen: die gegebenen Variablen, welche in Integritätsbedingungen im Rumpf gegeben sind, und die existenzquantifizierten Variablen, welche im Kopf von tgd's neu hinzukommen. Die gegebenen Variablen entsprechen hier den in [Heu17] in Tableaus verwendeten ausgezeichneten Variablen und die existenzquantifizierten Variablen entsprechen den nichtausgezeichneten Variablen.

Bemerkung. Die Variablen- und Nullwertbezeichnungen in den Beispielen dieses Kapitels folgen dem Schema

$$\upsilon_{\text{Attributbezeichnung}_{\text{Indexnummer}}},$$
$$\varepsilon_{\text{Attributbezeichnung}_{\text{Indexnummer}}},$$
$$\eta_{\text{Attributbezeichnung}_{\text{Indexnummer}}}.$$

Dabei wird υ bei gegebenen Variablen in tgd's und egd's verwendet, ε bei existenzquantifizierten Variablen in tgd's und egd's und η bei Nullwerten in Instanzen. Der Übersichtlichkeit halber werden die Attributbezeichnungen gegebenenfalls abgekürzt. Die zusätzlichen Indexnummern werden verwendet, falls mehrere verschiedene Variablen oder Nullwerte mit derselben Attributbezeichnung verwendet werden.

Beispiel 2.8. *Ein Beispiel für eine tgd ist*

$$\begin{aligned}&\mathit{STUDENTEN}(\upsilon_{id_{stud}}, \upsilon_{matnr}, \upsilon_{nachn}, \upsilon_{vorn}, \upsilon_{stuga})\\ &\wedge\ \mathit{TEILNEHMER}(\upsilon_{id_{teiln}}, \upsilon_{modnr}, \upsilon_{matnr})\\ &\rightarrow \exists \varepsilon_{id_{noten}}, \varepsilon_{sem}, \varepsilon_{note}\colon \mathit{NOTEN}(\varepsilon_{id_{noten}}, \upsilon_{modnr}, \upsilon_{matnr}, \varepsilon_{sem}, \varepsilon_{note})\end{aligned}$$

Das folgende Beispiel für eine egd simuliert die Schlüsselfunktion für "Matrikelnummer" in Tabelle 1.2:

$$\begin{aligned}&\mathit{STUDENTEN}(\upsilon_{id_{stud_1}}, \upsilon_{matnr}, \upsilon_{nachn_1}, \upsilon_{vorn_1}, \upsilon_{stuga_1})\\ &\wedge\ \mathit{STUDENTEN}(\upsilon_{id_{stud_2}}, \upsilon_{matnr}, \upsilon_{nachn_2}, \upsilon_{vorn_2}, \upsilon_{stuga_2})\\ &\rightarrow \upsilon_{id_{stud_1}} = \upsilon_{id_{stud_2}}, \upsilon_{nachn_1} = \upsilon_{nachn_2}, \upsilon_{vorn_1} = \upsilon_{vorn_2}, \upsilon_{stuga_1} = \upsilon_{stuga_2}\end{aligned}$$

Bemerkung. Die Allquantoren vor dem Rumpf wurden der Einfachheit halber weggelassen.

Die eben definierten Integritätsbedingungen werden in eine gegebene Instanz oder Anfrage bei einer Anwendung des CHASE eingearbeitet. Dabei werden durch die tgd's diese Strukturen verändert. Damit das Ergebnis einer CHASE-Anwendung noch einen Bezug zur ursprünglichen gegebenen Struktur hat, werden nur solche Veränderungen durchgeführt, die die Äquivalenz zwischen den Strukturen von vor und nach der Anwendung beibehalten. Dazu werden im folgenden Homomorphismen und Trigger definiert gefolgt von den Definitionen der CHASE-Varianten.

Definition 2.24 (Homomorphismus)**.** Seien I_1 und I_2 zwei Instanzen über demselben Datenbankschema mit Werten in $\mathit{Const} \cup \mathit{Null}$. Ein **Homomorphismus** $h\colon I_1 \rightarrow I_2$ ist eine Abbildung von $\mathit{Const}(I_1) \cup \mathit{Null}(I_1)$ nach $\mathit{Const}(I_2) \cup \mathit{Null}(I_2)$, sodass gilt:

(1) $\forall c \in \mathit{Const}(I_1)\colon h(c) = c$,

(2) $\forall R_i(a_1, ...a_n) \in I_1\colon R_i(h(a_1), ..., h(a_n)) \in I_2$.

I_1 und I_2 sind **äquivalent**, was mit $I_1 \leftrightarrow I_2$ bezeichnet wird, falls es zwei Homomorphismen $h\colon I_1 \to I_2$ und $h'\colon I_2 \to I_1$ gibt.

Beispiel 2.9. *Gegeben seien zwei Instanzen*

$$I_1 = \mathit{STUDENTEN}(S_3, 3, \mathit{Müller}, \eta_{vorn}, \mathit{Elektrotechnik}),$$
$$I_2 = \mathit{STUDENTEN}(S_3, 3, \mathit{Müller}, \mathit{Max}, \mathit{Elektrotechnik}),$$

dann gibt es einen Homomorphismus von I_1 auf I_2, der den Nullwert η_{vorn} auf "Max" und die Konstanten auf sich selbst abbildet.

Definition 2.25 (Trigger)**.** Sei I eine Instanz und $b \in \mathcal{B}$ eine tgd oder egd. Dann ist ein **Trigger** für b in I ein Homomorphismus $h\colon \varphi(\mathbf{x}) \to I$ mit $\varphi(\mathbf{x})$ als Rumpf von b.
Ein **aktiver Trigger** für b in I ist ein Trigger h mit:

- falls b tgd ist, existiert keine Erweiterung von h zu einem Homomorphismus von $\psi(\mathbf{x}, \mathbf{y}) \to I$
- falls b egd ist, ist $h(x_1) \neq h(x_2)$, wobei x_1 und x_2 im Kopf von b vorkommen.

I genügt $b \Leftrightarrow \nexists$ aktiver Trigger für b in I.

Beispiel 2.10. *Gegeben sei die Instanz*

$$\begin{aligned} I = {} & \mathit{STUDENTEN}(S_3, 3, \mathit{Müller}, \mathit{Max}, \mathit{Elektrotechnik}), \\ & \mathit{TEILNEHMER}(T_7, 002, 3), \\ & \mathit{TEILNEHMER}(T_{21}, 007, 3) \end{aligned}$$

und die tgd

$$\begin{aligned} b = {} & \mathit{STUDENTEN}(\upsilon_{id_{stud}}, \upsilon_{matnr}, \upsilon_{nachn}, \upsilon_{vorn}, \upsilon_{stuga}) \\ & \wedge \mathit{TEILNEHMER}(\upsilon_{id_{teiln}}, \upsilon_{modnr}, \upsilon_{matnr}) \\ & \rightarrow \exists \varepsilon_{id_{noten}}, \varepsilon_{sem}, \varepsilon_{note}\colon \mathit{NOTEN}(\varepsilon_{id_{noten}}, \upsilon_{modnr}, \upsilon_{matnr}, \varepsilon_{sem}, \varepsilon_{note}). \end{aligned}$$

I wird dann erweitert auf die Instanz I', indem der aktive Trigger für b in I angewandt wird:

$$\begin{aligned} I' = {} & \mathit{STUDENTEN}(S_3, 3, \mathit{Müller}, \mathit{Max}, \mathit{Elektrotechnik}), \\ & \mathit{TEILNEHMER}(T_7, 002, 3), \\ & \mathit{TEILNEHMER}(T_{21}, 007, 3), \\ & \mathit{NOTEN}(\eta_{ID_{noten_1}}, 002, 3, \eta_{sem_1}, \eta_{note_1}). \end{aligned}$$

Da der Trigger immer noch aktiv ist, wird dieser erneut angewandt:

$$\begin{aligned} I' = {} & \mathit{STUDENTEN}(S_3, 3, \mathit{Müller}, \mathit{Max}, \mathit{Elektrotechnik}), \\ & \mathit{TEILNEHMER}(T_7, 002, 3), \\ & \mathit{TEILNEHMER}(T_{21}, 007, 3), \\ & \mathit{NOTEN}(\eta_{ID_{noten_1}}, 002, 3, \eta_{sem_1}, \eta_{note_1}) \\ & \mathit{NOTEN}(\eta_{ID_{noten_2}}, 007, 3, \eta_{sem_2}, \eta_{note_2}). \end{aligned}$$

Jetzt ist kein aktiver Trigger mehr vorhanden und I' erfüllt somit b.

Definition 2.26 (CHASE-Schritt)**.** Sei I eine Instanz.
(1) Seien b eine tgd $\forall \mathbf{x}\colon (\varphi(\mathbf{x}) \to \exists \mathbf{y}\colon \psi(\mathbf{x}, \mathbf{y}))$ und h ein Homomorphismus von $\varphi(\mathbf{x})$ nach I, sodass es

keine Erweiterung von h zu einem Homomorphismus h' von $\varphi(\mathbf{x}) \wedge \psi(\mathbf{x}, \mathbf{y})$ nach I gibt. Dann kann b auf I mit dem Homomorphismus h **angewandt** werden.
Sei I' die Vereinigung von I mit der Menge von Fakten, die entstanden ist durch:

(1) Erweiterung von h zu h', sodass jeder Variablen in $\mathbf{y}$ ein neuer markierter Nullwert zugewiesen wird,

(2) anschließende Anwendung von h' auf ψ, sodass $I' = I \cup h'(\psi(\mathbf{x}, \mathbf{y}))$.

I' ist dann das Ergebnis der Anwendung von b auf I mit h und wird bezeichnet als $I \xrightarrow{b,h} I'$.

(2) Seien b eine egd $\forall \mathbf{x}\colon (\varphi(\mathbf{x}) \rightarrow (x_1 = x_2)$ und h ein Homomorphismus von $\varphi(\mathbf{x})$ nach I, sodass $h(x_1) \neq h(x_2)$. Dann kann b auf I mit dem Homomorphismus h **angewandt** werden.
Es werden nun zwei Fälle unterschieden:

- Falls sich sowohl $h(x_1)$ als auch $h(x_2)$ in *Const* befinden, dann ist das Ergebnis der Anwendung von b auf I mit h ein Fehlschlag und wird bezeichnet als $I \xrightarrow{b,h} \perp$.
- Andererseits sei $I' = I$ und seien $h(x_1)$ und $h(x_2)$ folgendermaßen definiert:
  - falls eines eine Konstante c ist, wird der markierte Nullwert überall, wo er vorkommt, durch c ersetzt;
  - falls beides markierte Nullwerte sind, wird einer der beiden Nullwerte überall, wo er vorkommt, durch den anderen vorzugsweise mit dem kleineren Index ersetzt.

  I' ist dann das Ergebnis der Anwendung von b auf I mit h und wird bezeichnet als $I \xrightarrow{b,h} I'$.

$I \xrightarrow{b,h} I'$ (wobei $I' = \perp$ sein kann) wird nun als **CHASE-Schritt** bezeichnet.

In einfacheren Worten ausgedrückt wird ein CHASE-Schritt mit einer Abhängigkeit b auf eine Instanz I immer dann ausgeführt, falls I den Rumpf von b aber nicht den Kopf erfüllt. Es existiert also ein aktiver Trigger für b in I.
Abbildung 2.1 zeigt, wie die Ersetzung von Termen im CHASE-Schritt durch egd's gehandhabt wird. Diese Termhierarchie wird auch später in der Implementierung des CHASE-Tools verwendet. Zu sehen ist dabei, dass Konstanten die höchste Wertigkeit haben und jede andere Art Term auf eine Konstante abgebildet werden kann. Konstanten können nicht auf andere Terme sondern nur auf sich selbst abgebildet werden. Nullwerte können auf andere Nullwerte oder auf Konstanten abgebildet werden. Jede Variable kann auf einen Nullwert oder auf eine Konstante abgebildet werden. Gegebene Variablen können nicht auf andere gegebene oder existenzquantifizierte Variablen abgebildet werden. Existenzquantifizierte Variablen hingeben können auf jeden anderen Term abgebildet werden.

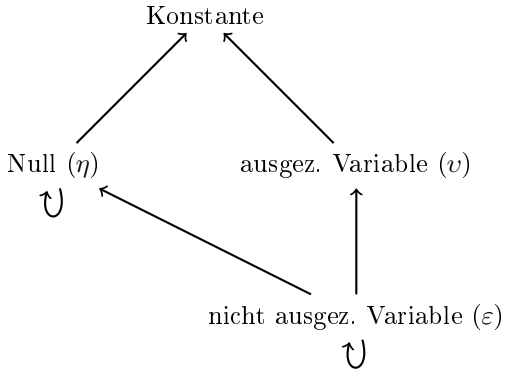


Abbildung 2.1. Termhierarchie

Definition 2.27 (Standard-CHASE)**.** Seien $\mathcal{B}$ eine Menge von tgd's und egd's und I eine Instanz. Eine **CHASE-Sequenz** von I mit $\mathcal{B}$ ist eine (endliche oder unendliche) Folge von CHASE-Schritten $I_i \xrightarrow{b_i,h_i} I_{i+1}$ mit $i \in \mathbb{N}_0$, $I = I_0$ und $b_i \in \mathcal{B}$.
Ein **endlicher CHASE** von I mit $\mathcal{B}$ ist eine endliche CHASE-Sequenz $I_i \xrightarrow{b_i,h_i} I_{i+1}$ mit $0 \leq i < n$ und der Voraussetzung, dass entweder

(a) $I_n = \perp$ oder

(b) es gibt kein $b_i \in \mathcal{B}$ und keinen Homomorphismus h_i, sodass b_i auf I_n mit h_i angewandt werden kann.

I_n ist dann das Ergebnis des endlichen CHASE. Im Falle von (a) handelt es sich dabei um einen **fehlgeschlagenen** oder **gescheiterten endlichen CHASE** und im Falle von (b) um einen **erfolgreichen endlichen CHASE**.
Die Anwendung des CHASE auf eine Instanz I mit Abhängigkeiten $\mathcal{B}$ wird mit $chase(I, \mathcal{B})$ bezeichnet.

Beispiel 2.11. *Der CHASE-Algorithmus ist die Anwendung von aktiven Triggern. So wird der CHASE in Beispiel 2.10 auf die Instanz I mit der Integritätsbedingung b angewandt. Die resultierende Instanz der CHASE-Anwendung erfüllt dann b und es muss nicht weiter geCHASEt werden.*

Der CHASE-Algorithmus wird in Algorithmus 1 als Pseudocode dargestellt, welcher aus [BKM+17] entnommen wurde und für diese Arbeit als Grundlage für die Entwicklung des CHASE-Tools dienen wird. Die Funktion des Algorithmus' wird nun anhand des Pseudocodes erklärt.
Dem Algorithmus wird eine zu CHASEnde Instanz I und die reinzuCHASEnden Integritätsbedingungen $\mathcal{B}$ als Eingabeparameter übergeben. Zuerst wird eine temporäre Instanz I' als I initialisiert. Diese stellen die neuen Fakten am Ende jeder Iteration mit allen Integritätsbedingung dar. Außerdem wird noch eine weitere leere Instanz N erstellt, die später alle durch tgd's generierten Tupel enthält.
Für jede Integritätsbedingung $b \in \mathcal{B}$ wird überprüft, ob der Rumpf von b durch I erfüllt wird (Zeilen 5 - 7), jedoch nicht der Kopf (Zeile 7 durch Überprüfung auf aktiven Trigger). Außerdem muss mindestens ein Atom aus $h(p_1(\mathbf{x}))$ in I' enthalten sein, um sicherzugehen, dass neue hinzugekommene Fakten mitverantwortlich für die Ausführung von Triggern sind, sodass nicht noch einmal ausschließlich alle alten Fakten die Integritätsbedingungen triggern.
Anschließend wird die Integritätsbedingung b zwischen einer tgd und einer egd unterschieden (Zeilen 8 und 12). Im Fall der tgd werden alle existenzquantifizierten Variablen neuer Tupel auf neue markierte Nullwerte abgebildet (Zeile 9) und die so entstehenden Tupel schließlich der temporären Instanz N hinzugefügt. Somit enthält N alle neuen durch tgd's generierten Tupel nach einer Iteration mit allen Integritätsbedingungen.
Im Fall der egd wird der Algorithmus mit einem Fehlschlag abgebrochen, falls die gleichzusetzenden Variablen verschieden und zugleich Konstanten sind (Zeilen 13 - 15). Andernfalls wird die "größere" der Variablen durch die "kleinere" ersetzt (Zeilen 16 - 19) [BKM+17]. Die Ersetzung entspricht hierbei der aus Definition 2.26, in der markierte Nullwerte durch Konstanten oder durch andere Nullwerte mit einem kleineren Index ersetzt werden.
Sobald die vorigen Schritte mit jeder Integritätsbedingung von $\mathcal{B}$ durchgeführt wurden, wird die Variablenersetzung μ von den Zeilen 17 und 18 auf I und N angewandt. Dies sorgt für eine Ersetzung der Nullwerte, sodass die egd's erfüllt werden. Davon wird dann I aus dem Resultat entfernt, sodass nur noch neue, vorher nicht in I enthaltene Fakten übrigbleiben, was schließlich I' zugewiesen wird. Nun enthält I' alle neuen Fakten einer Iteration: die durch N neuen gewonnenen tgd-Tupel und die durch μ veränderten Tupel. Die Termabbildungen μ werden dann erneut auf I angewandt, um gegebenenfalls nochmal Nullwerte in I durch andere Nullwerte oder Konstanten zu ersetzen. Abschließend wird I aktualisiert, indem I mit den neuen Fakten I' vereinigt wird. Jetzt wurde jeder aktive Trigger jeder Integritätsbedingung

Algorithmus 1 : Standard-CHASE

Input : Instanz I, Menge von tgd's und egd's $\mathcal{B}$
Output : Instanz I mit reingeCHASEten Integritätsbedingungen $\mathcal{B}$
oder FAIL bei nichterfüllter egd

```
1  Function chase(I, B)
2      I' := I
3      while I' ≠ {} do
4          N := {}, μ := {}
5          foreach b ∈ B mit Rumpf p1(x) do
6              foreach Trigger h für b in I mit h(p1(x)) ∩ I' ≠ {} do
7                  if h ist aktiver Trigger für b in μ(N ∪ I) then
8                      if b = ∀x p1(x) → ∃y p2(x, y) ist tgd then
9                          h' := h ∪ {y → v} mit v neue Nullwerte
10                         N := N ∪ h'(p2(x, y))
11                     end
12                     else if b = ∀x p1(x) → xi = xj ist egd then
13                         if (h(xi) ≠ h(xj)) ∧ ({h(xi), h(xj)} ⊆ Const) then
14                             FAIL
15                         end
16                         else
17                             ν := {max(h(xi), h(xj)) ↦ min(h(xi), h(xj))}
18                             μ := μ ∘ (ν ∘ μ)
19                         end
20                     end
21                 end
22             end
23         end
24         I' := μ(N ∪ I) − I
25         I := μ(I) ∪ I'
26     end
27     return I
28 end
```

einmal auf I angewandt und I entsprechend kumulativ verändert (neue Tupel durch tgd's, Gleichsetzungen bestimmter Werte in Tupeln durch egd's).
Es werden nun alle Integritätsbedingungen durch I ohne die neue hinzugekommenen Fakten erfüllt. Die nächste Iteration wird nun versuchen, die Integritätsbedingungen durch I inklusive der neuen Fakten zu erfüllen. Dabei können wieder neue Fakten entstehen. Dieser gesamte Vorgang wird solange wiederholt, bis die temporäre Instanz I' durch den letzten Schritt leer geworden ist, was bedeutet, dass keine neuen Fakten seit der letzten Iteration hinzugekommen sind. Das finale Ergebnis des Algorithmus ist dann die veränderte Instanz I.

Weitere CHASE-Varianten

Als Nächstes werden einige zusätzliche Definitionen beschrieben, die nicht notwendig für das Ziel dieser Arbeit sind, jedoch noch in weitere Anwendungsgebiete des CHASE hereinreichen und optional für die Umsetzung des CHASE-Tools sind. Dazu gehören noch zwei weitere Varianten des CHASE, die eine Abwandlung des eben vorgestellten CHASE-Algorithmus' darstellen. Der Core-CHASE ist dabei die Anwendung des CHASE mit einer parallelen Ausführung der CHASE-Schritte und einer anschließenden Berechnung eines eindeutigen Kerns, auf den die resultierende Instanz reduziert werden soll. Der darauffolgende Oblivious CHASE ist der CHASE ohne eine Überprüfung auf vorhandene aktive Trigger.

Definition 2.28 (Kern(-instanz)/ Core)**.** Ein Homomorphismus $h\colon I \to I'$ mit $I' \subseteq I$ und $h(x) = x$ für alle x in I' wird als **Reduktion** bezeichnet und I' ist eine **Reduktion von** I. Eine Reduktion ist **echt**, falls sie nicht surjektiv ist. Eine Instanz ist ein **Kern**, falls sie keine echten Reduktionen hat. Ein **Kern einer Instanz** I ist eine Reduktion von I, welche zugleich ein Kern ist, und wird mit $\mathrm{core}(I)$ bezeichnet. Kerne einer Instanz sind bis auf Isomorphie eindeutig.

Definition 2.29 (Paralleler CHASE-Schritt, Core-CHASE-Schritt)**.** Sei $\mathcal{B}$ eine Menge von tgd's und egd's und seien I, I', J und K Instanzen über demselben Datenbankschema, dann bezeichnet $I \xrightarrow{\mathcal{B}} J$ einen **parallelen CHASE-Schritt**, wenn gilt:

(1) $I \not\models \mathcal{B}$,

(2) $J = \bigcup_{b \in \mathcal{B}, I \xrightarrow{b,h} I'} I'$.

Ein **Core-CHASE-Schritt** wird mit $I \xrightarrow{\mathcal{B}\downarrow} K$ bezeichnet, falls $I \xrightarrow{\mathcal{B}} J$ gilt und $K = \mathrm{core}(J)$ ist.

Definition 2.30 (Core-CHASE)**.** Der **Core-CHASE** ist analog zum Standard-CHASE als eine Folge von Core-CHASE-Schritten definiert und wird bei Anwendung auf eine Instanz I mit Abhängigkeiten $\mathcal{B}$ mit $chase_{\mathrm{core}}(I, \mathcal{B})$ bezeichnet.

Definition 2.31 (Oblivious CHASE)**.** Der **naive Oblivious CHASE** ist analog zum Standard-CHASE als eine Folge von CHASE-Schritten definiert mit dem Unterschied, dass vor der Ausführung des CHASE-Schrittes nicht geprüft wird, ob der Kopf einer anzuwendenden Abhängigkeit bereits erfüllt wird. Das heißt, die Prüfung auf aktive Trigger wird ausgesetzt.
Der **Skolem-Oblivious CHASE** ist wie der naive Oblivious CHASE definiert mit dem Unterschied, dass bei Ausführung einer tgd keine neu markierten Nullwerte eingeführt werden, sondern eine sogenannte **Skolemfunktion** angewendet auf die Variablen in **y** neue markierte Nullwerte eindeutig bestimmt, sodass eine wiederholte Anwendung von bereits erfüllten tgd's und wiederholter Einführung von neuen markierten Nullwerten verhindert wird.

Beispiel 2.12. *Gegeben sei die Instanz*

$$\begin{aligned} I = {} & \mathit{STUDENTEN}(S_3, 3, \mathit{Müller}, \mathit{Max}, \mathit{Elektrotechnik}), \\ & \mathit{TEILNEHMER}(T_7, 002, 3) \end{aligned}$$

und die tgd

$$\begin{aligned} b = {} & \mathit{STUDENTEN}(\upsilon_{id_{stud}}, \upsilon_{matnr}, \upsilon_{nachn}, \upsilon_{vorn}, \upsilon_{stuga}) \\ & \wedge\ \mathit{TEILNEHMER}(\upsilon_{id_{teiln}}, \upsilon_{modnr}, \upsilon_{matnr}) \\ & \rightarrow \exists \varepsilon_{id_{noten}}, \varepsilon_{sem}, \varepsilon_{note} \colon \mathit{NOTEN}(\varepsilon_{id_{noten}}, \upsilon_{modnr}, \upsilon_{matnr}, \varepsilon_{sem}, \varepsilon_{note}). \end{aligned}$$

Anwendung des naiven Oblivious CHASE auf I mit b ergibt:

$$\begin{aligned} chase_{NO}(I, b) = {} & \mathit{STUDENTEN}(S_3, 3, \mathit{Müller}, \mathit{Max}, \mathit{Elektrotechnik}), \\ & \mathit{TEILNEHMER}(T_7, 002, 3), \\ & \mathit{NOTEN}(\eta_{ID_{noten_1}}, 002, 3, \eta_{sem_1}, \eta_{note_1}). \end{aligned}$$

Aufgrund der fehlenden Überprüfung auf aktive Trigger wird der naive Oblivious CHASE erneut ange-

wandt:

$$\begin{aligned} chase_{NO}(chase_{NO}(I,b),b) = \; & STUDENTEN(S_3, 3, \textit{Müller}, \textit{Max}, \textit{Elektrotechnik}), \\ & TEILNEHMER(T_7, 002, 3), \\ & NOTEN(\eta_{ID_{noten_1}}, 002, 3, \eta_{sem_1}, \eta_{note_1}), \\ & NOTEN(\eta_{ID_{noten_2}}, 002, 3, \eta_{sem_2}, \eta_{note_2}). \end{aligned}$$

Dies kann nun endlos wiederholt werden, weshalb der naive Oblivious CHASE in diesem Fall nicht terminiert.

Bemerkung. Der Oblivious CHASE wird in Algorithmus 1 umgesetzt, indem die Überprüfung auf aktive Trigger in Zeile 7 ausgelassen wird.

2.4. CHASE&BACKCHASE

Der CHASE&BACKCHASE-Algorithmus ist eine Anwendung des CHASE auf Anfragen, mit dem ein sogenannter *Universalplan* erstellt werden soll, welcher eine Mischung aller Anfragepläne ist, die durch die gegebenen Integritätsbedingungen zugelassen werden. Das Verfahren besteht aus zwei Phasen, wie der Name schon verlauten lässt. Vor seiner informalen Definition werden allerdings noch die Begriffe der Anfragen und Teilanfragen definiert. Sämtliche folgende Definitionen entstammen aus [DPT06].

Definition 2.32 (Anfrage)**.** Eine (**konjunktive**) **Anfrage** q über einem Schema R ist ein Ausdruck der Form

$$q(\mathbf{x}) : -\exists \mathbf{y} \colon \varphi(\mathbf{x}, \mathbf{y}),$$

wobei $\varphi(\mathbf{x}, \mathbf{y})$ eine Konjunktion von relationalen Atomen ist, sodass jede der Variablen in $\mathbf{x}$ und $\mathbf{y}$ in mindestens einem der Atome in $\varphi(\mathbf{x}, \mathbf{y})$ vorkommt.
Der Teil φ wird auch als **Rumpf** und q als **Kopf** der Anfrage bezeichnet.
Eine Anfrage q' ist in einer Anfrage q enthalten, falls das Ergebnis von q' im Ergebnis von q enthalten ist, wobei q' und q jeweils auf dieselbe Instanz I, die die Integritätsbedingungen $\mathcal{B}$ erfüllt, angewandt wurden. Dies ist der Fall, wenn es einen Homomorphismus h (**Containment-Mapping**) von q nach q' gibt. Bezeichnet wird dies mit $q' \subseteq_{\mathcal{B}} q$.
Zwei Anfragen q und q' angewandt auf eine Instanz I mit erfüllten Integritätsbedingungen $\mathcal{B}$ sind äquivalent ($q \equiv_{\mathcal{B}} q'$), falls $q' \subseteq_{\mathcal{B}} q$ und $q \subseteq_{\mathcal{B}} q'$ gilt.

Definition 2.33 (Teilanfrage)**.** Gegeben sei zwei Anfragen

$$\begin{aligned} q(\mathbf{x}) &: -\exists \mathbf{y} \colon \varphi(\mathbf{x}, \mathbf{y}), \\ q'(\mathbf{x}') &: -\exists \mathbf{y}' \colon \varphi'(\mathbf{x}', \mathbf{y}') \end{aligned}$$

über einem Schema R. Die Anfrage q' ist eine **Teilanfrage** von q, wobei $\varphi' \subseteq \varphi$, $\mathbf{x}' \subseteq \mathbf{x}$ und $\mathbf{y}' \subseteq \mathbf{y}$ gilt.

Definition 2.34 (CHASE&BACKCHASE)**.** Der CHASE&BACKCHASE besteht aus der CHASE-Phase und der BACKCHASE-Phase. In der CHASE-Phase wird durch die CHASE-Anwendung der CHASE-Parameter ($\star$) in das CHASE-Objekt ($\bigcirc$) eingearbeitet. Die BACKCHASE-Phase verändert das Ergebnis des CHASE ($\circledast$) durch weitere CHASE-artige Regeln, um dieses je nach Anwendung zu optimieren.

Die originale Anwendung des CHASE&BACKCHASE verwendet Anfragen als CHASE-Objekt und Integritätsbedingungen als CHASE-Parameter. Nach der CHASE-Phase wird die daraus entstandene Anfrage durch den BACKCHASE in einen optimierten Anfrageplan verwandelt. Dazu folgendes Beispiel:

Beispiel 2.13. *Seien S und T zwei Schemata, q eine Anfrage über S und $\mathcal{B}$ eine Menge von Integritätsbedingungen für $S \cup T$. Die Anfrage q soll nun in eine Anfrage q' über T umformuliert werden, sodass q' dieselbe Antwort wie q auf jede beliebige $S \cup T$ - Datenbankinstanz liefert, die $\mathcal{B}$ erfüllt. Das heißt, $q \equiv_{\mathcal{B}} q'$. Das CHASE&BACKCHASE-Verfahren läuft nun wie folgt ab:*

- ***CHASE-Phase**: Es wird $\mathcal{B}$ genutzt, um den CHASE auf q anzuwenden, bis kein CHASE-Schritt mehr möglich ist. Das daraus entstehende Resultat ist der **Universalplan** u.*
- ***BACKCHASE-Phase**: Es werden alle Teilanfragen q' von u auf Äquivalenz mit q unter $\mathcal{B}$ überprüft und diejenige als Ergebnis des Verfahrens ausgegeben, die die geringste Anzahl an Konjunktionen im Rumpf aufweist. Es genügt dabei, zu überprüfen, ob $q' \subseteq_{\mathcal{B}} q$ [DPT06].*

Eine mögliche Anwendung von CHASE&BACKCHASE ist die Umformulierung einer Anfrage in eine Anfrage über ausschließlich Sichten.

- ***CHASE-Phase**: Seien eine Anfrage Q, eine Menge von Integritätsbedingungen $\mathcal{B}$ und eine Menge von Sichtenbedingungen $\mathcal{V}$ gegeben. Dann ist eine Anfrage U ein **Universalplan**, falls diese durch eine auf $\mathcal{V}$ eingeschränkte Anfrage $Q' = \text{chase}(Q, \mathcal{B} \cup \mathcal{V})$ entstanden ist.*
- ***BACKCHASE-Phase**: Die Teilanfragen des Universalplans U werden bezüglich $\mathcal{B}$ und $\mathcal{V}$ auf Äquivalenz mit Q geprüft. Alle zu Q äquivalenten minimalen Teilanfragen werden dabei als Ergebnis ausgegeben (d. h. Anfragen, die selbst keine zu Q äquivalenten Teilanfragen enthalten).*

Eine Veranschaulichung dieser Variante befindet sich in Unterabschnitt 3.1.2.

Außerdem findet CHASE&BACKCHASE im Provenance-Management Verwendung, um die Existenz von sogenannten CHASE-Inversen zu überprüfen, welche Rückabbildungen von geCHASEten Instanzen zu ihren ursprünglichen Instanzen vor der CHASE-Anwendung sind.
Nachdem nun alle Begriffe und Konzepte für das Verständnis des CHASE besprochen wurden, folgt nun im nächsten Kapitel ein kurzer Überblick über einige Anwendungsszenarien des CHASE. Im Anschluss daran werden einige bestehende Tools betrachtet, die den CHASE auf einzelne dieser Anwendungen spezialisiert einsetzen. Am Ende des Kapitels wird dann entschieden, inwiefern die Tools für das eigene nutzbar gemacht werden können.

3. Aktueller Stand der Forschung und Technik

Dieses Kapitel ist in zwei Abschnitte unterteilt, in denen sich zum Einen mit dem aktuellen Stand der Forschung und zum Anderen mit dem aktuellen Stand der Technik befasst wird. Es werden dabei verschiedene Anwendungsgebiete und Tools des CHASE betrachtet. Dazu soll die Übersicht des CHASE aus Abschnitt 1.1 als Stütze dienen und hier noch einmal dargestellt werden:

	$\star$	$\bigcirc$	Ergebnis	Ziel	Tool
0.	Abhängigkeiten	DB-Schema	DB-Schema mit Integritätsbedingungen	optimierter DB-Entwurf	
I.	Abhängigkeiten	Anfragen	Anfragen	Semantische Optimierung	PDQ
II.	Sichten	Anfragen	Anfragen auf Sichten	AQuV	ProvCB
II'.	Operationen	Anfragen	Anfragen auf Operationen	AQuO	
III.	s-t tgd's, egd's, tgd's	Quell-DB	Ziel-DB	Datenaustausch, Datenintegration	Llunatic, ChaseFUN
IV.	tgd's, egd's	DB	modifizierte DB	Cleaning	Llunatic, ChaseFUN
V.	tgd's, egd's	unvollständige DB	Anfrageergebnis	sichere Antworten	
VI.	s-t tgd's, egd's, tgd's	DB	Anfrageergebnis	invertierbare Auswertung	

Tabelle 3.1. Überblick von CHASE-Anwendungen und zugehörigen Parametern

3.1. Forschung und Theorie

In diesem Abschnitt soll der aktuelle Stand der Forschung des CHASE-Algorithmus betrachtet werden. Dazu werden Arbeiten und Ergebnisse insbesondere der letzten Jahre vorgestellt, zu denen sowohl mathematische Erkenntnisse als auch neue Techniken gehören.
Der CHASE-Algorithmus wurde 1979 von zwei Forschungsteams entworfen und in [ABU79, MMS79] vorgestellt. Seitdem hat sich rund um den CHASE ein eigenes Forschungsgebiet entwickelt. Es haben sich mehrere Anwendungsszenarien herausgestellt, auf die der CHASE eingesetzt werden kann. Dies sind unter Anderem Datenaustausch und -integration, Provenance-Management, Anfrageoptimierung unter gegebenen Integritätsbedingungen und die Einarbeitung von Sichten in Anfragen, wobei letzteres auch als "Answering Queries using Views" (AQuV) bekannt ist. Dadurch gilt der CHASE heute als ein universales Werkzeug, das in vielen Szenarien der Datenbankforschung zum Einsatz kommt.
Im Folgenden sollen einige der Anwendungsmöglichkeiten des CHASE näher betrachtet werden, die für die Umsetzung des CHASE-Tools vorwiegend verwendet werden.

3.1.1. Semantische Optimierung

Semantische Optimierung von Anfragen ist das Umformulieren dieser Anfragen unter Einarbeitung von bestimmten gegebenen Integritätsbedingungen und ist dem Bereich der Anfragetransformationen zuzuordnen. In [DPT99] wurden Anfragen, die auf logischen Schemata formuliert wurden, in Anfragen auf physischen Schemata transformiert, wobei die Integritätsbedingungen des logischen Schemas in die Anfrage mit eingearbeitet wurden. Es wird durch den CHASE ein *universeller Anfrageplan* erstellt, auf dem dann ein BACKCHASE durchgeführt wird, um gewisse Strukturen der originalen Anfrage zu entfernen. Dabei wird ebenfalls auf Erfüllung der Integritätsbedingungen geprüft. Ein universeller Anfrageplan ist

dabei eine geCHASEte Anfrage, die alle Zugriffsstrukturen und -pfade enthält, die durch die Integritätsbedingungen zugelassen sind. Dieses Konzept wird sehr vereinfacht in Abbildung 3.1 veranschaulicht.

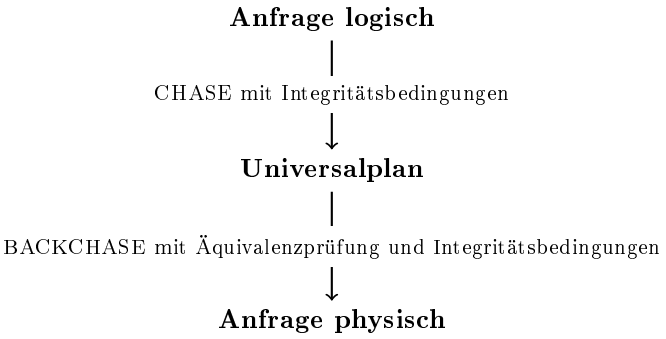


Abbildung 3.1. Anfragetransformation

Alternativ lässt sich eine Anfrage auch als ein Tableau darstellen, wobei jede Zeile des Tableaus ein Relationenschema darstellt, das in der Anfrage mitverwendet wird. Unter gegebenen Integritätsbedingungen wird dann versucht, die Anzahl der Zeilen des Tableaus zu minimieren, da jedes Einbeziehen einer Zeile einen Verbund in der Anfrageausführung darstellt.

Beispiel 3.1. *Gegeben sei eine Relation $r(\{A, B, C\})$ mit FD $B \to C$.*

A	B	C		A	B	C
a_1	a_2	b_1	$\xrightarrow{\text{CHASE}}$	a_1	a_2	a_3
b_2	a_2	a_3		b_2	a_2	a_3

Ziel ist es, die Anfrage $\pi_{A,B}(r) \bowtie \pi_{B,C}(r)$ zu optimieren.
Die Abbildung des geCHASEten Tableaus (rechts) auf Zeile $\langle a_1, a_2, a_3 \rangle$ mit $b_1 \to a_1$, $a_1 \to a_1$, $a_2 \to a_2$ und $a_3 \to a_3$ zeigt, dass der natürliche Verbund $\pi_{A,B}(r) \bowtie \pi_{B,C}(r)$ eingespart werden kann, indem direkt r genutzt wird.

3.1.2. Answering Queries using Views

Answering Queries using Views (AQuV) ist das Beantworten von Anfragen auf Sichten, indem zusätzlich zu den üblichen Integritätsbedingungen noch Bedingungen für die entsprechenden Sichten angegeben werden, unter denen gestellte Anfragen umformuliert werden sollen. So bleiben nach der Umformulierung der Anfrage nur noch die Sichten übrig und alle anderen Strukturen werden entfernt. Veranschaulicht wird dies im folgenden Beispiel, welches auf einem entnommenen Beispiel aus [DH13] basiert:

Beispiel 3.2. *Gegeben sei die Anfrage $q(x) \leftarrow R(x, w, y), S(y, z)$ auf den Relationenschemata R und S mit Sichten*

$$V_R(x, y) \leftarrow R(x, w, y)$$
$$V_S(y, z) \leftarrow S(y, z)$$
$$V_{RS}(x, z) \leftarrow R(x, w, y), S(y, z).$$

Die Sichten werden durch CHASE und BACKCHASE als tgd's $\mathcal{V}$ ausgedrückt:

$$c_{V_R}\colon R(x,w,y) \to V_R(x,y)$$
$$b_{V_R}\colon V_R(x,y) \to \exists w\colon R(x,w,y)$$

$$c_{V_S}\colon S(y,z) \to V_S(y,z)$$
$$b_{V_S}\colon V_S(y,z) \to S(y,z)$$

$$c_{V_{RS}}\colon R(x,w,y) \wedge S(y,z) \to V_{RS}(x,z)$$
$$b_{V_{RS}}\colon V_{RS}(x,z) \to \exists w,y\colon R(x,w,y) \wedge S(y,z)$$

Anwendung des CHASE auf q und Berechnung des universellen Plans u davon ergeben

$$chase_{\mathcal{V}}(q) = R(x,w,y), S(y,z), V_R(x,y), V_S(y,z), V_{RS}(x,z) \text{ und}$$
$$u(x) = V_R(x,y), V_S(y,z), V_{RS}(x,z).$$

In der BACKCHASE-Phase werden anschließend alle Teilanfragen

$$\{\{\}, \{V_R(x,y)\}, \{V_S(y,z)\}, \{V_{RS}(x,z)\}, \{V_R(x,y), V_S(y,z)\},$$
$$\{V_R(x,y), V_{RS}(x,z\}, \{V_S(y,z), V_{RS}(x,z)\}, \{V_R(x,y), V_S(y,z), V_{RS}(x,z)\}\}$$

von $u(x)$ auf Äquivalenz mit q geprüft, was letztlich

$R_1(x) \leftarrow V_R(x,y), V_S(y,z)$ *und*
$R_2(x) \leftarrow V_{RS}(x,z)$ *ergibt.*

Dazu wird der CHASE wieder auf die Teilanfragen mit den Regeln b_{V_R}, b_{V_S} und $b_{V_{RS}}$ angewandt und ein Homomorphismus von q nach q', dem Ergebnis dieser CHASE-Anwendung, gesucht.

Eine sehr vereinfachte Darstellung dessen zeigt Abbildung 3.2. Hier wird deutlich, dass es sich im Prinzip um dieselbe Problematik wie zuvor in der semantischen Optimierung handelt (siehe Abbildung 3.1). Ausgehend von einer Anfrage q, die an ein Schema S gestellt wurde, soll diese mit dem CHASE&BACKCHASE in eine Anfrage über dem Schema T umformuliert werden und dabei dasselbe Ergebnis wie q liefern.

Deutsch und Hull machen sich in [DH13] den CHASE&BACKCHASE-Algorithmus zunutze, wobei hierbei Provenance-Informationen genutzt werden, um die Anzahl an Teilanfragen in der BACKCHASE-Phase zu reduzieren. Bekannt ist dieses Verfahren als *why-Provenance*. Für jedes Prädikat im CHASE werden dabei die Informationen mitgeführt, aus welchen Teilanfragen das Prädikat jeweils entstanden ist. Um die resultierende Anfrage auf Äquivalenz mit der originalen Anfrage zu prüfen, wird nach einem Homomorphismus von der originalen Anfrage auf die Ergebnisanfrage mit den eingearbeiteten Integritätsbedingungen gesucht.

3.1.3. Datenaustausch und -integration

Der CHASE findet ebenso beim Datenaustausch Anwendung. Bei der Formulierung von Kompositionen und Inversen von Schemaabbildungen in der Evolution von Schemata in [FKPT11] wurden neue

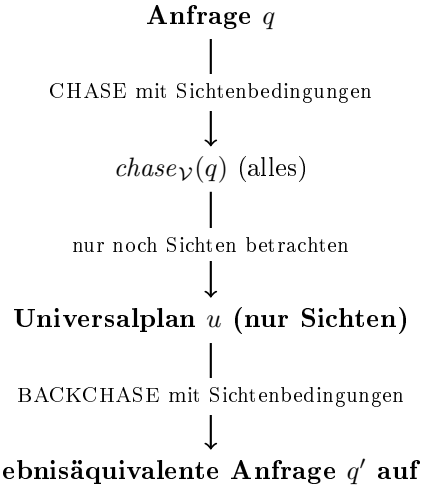


Abbildung 3.2. Anfragetransformation auf Sichten

Techniken zur Bildung von CHASE-Inversen untersucht. Dort wurden die *exakte CHASE-Inverse*, die *allgemeine CHASE-Inverse* und die *relaxte CHASE-Inverse* eingeführt. Diese stellen durch den CHASE entstandene Arten von Inversen von Schemaabbildungen dar, die je nach Art ein ursprüngliches Schema exakt oder nur teilweise wieder rekonstruieren können. So können schwächere Inverse wie die relaxte CHASE-Inverse genutzt werden, wenn die Rekonstruktion des originalen Schemas durch beispielsweise eine exakte CHASE-Inverse nicht möglich ist.
Deutsch, Nash und Remmel führten in [DNR08] den Core-CHASE und eine Erweiterung davon ein. Sie beschäftigten sich mitunter mit dem Problem, ob es Fälle gibt, in denen vorausgesagt werden kann, dass der CHASE terminiert. Dazu versuchten sie, neue Modelle und Formulierungen einzuführen, unter denen umfangreichere Klassen von Anfragen und Bedingungen behandelt werden können wie beispielsweise welche mit Negation, Ungleichheit oder Disjunktion. Mit einem allgemeinen Modellbegriff wurden dort bereits verschiedene CHASE-Anwendungen wie Query Containment, Bedingungsimplikation oder Datenaustausch zusammengefasst.

3.1.4. Provenance-Management

Provenance-Management beschäftigt sich mit der Herkunft von Daten bei Ergebnissen von Auswertungen, Berechnungen, Messungen und Anfragen. Diese Daten werden dann entweder direkt angegeben oder durch Anfragen beschrieben. Im Bereich des Provenance-Managements gibt es eine Masterarbeit von Tanja Auge an der Universität Rostock, in der mithilfe des CHASE&BACKCHASE Provenance-Anfragen in Big-Data-Analytics-Umgebungen umgesetzt werden können [Aug17].
Außerdem wurden in [DH13] Provenance-Informationen genutzt (hier insbesondere die minimale Zeugenbasis), um die exponentielle Menge an Teilanfragen, auf die der BACKCHASE angewandt werden würde, zu reduzieren. Dies sorgt zwar für eine längere Laufzeit während der CHASE-Phase, jedoch auch für eine beachtliche Laufzeitreduzierung in der BACKCHASE-Phase.

3.2. Umgesetzte CHASE-Tools

Dieser Abschnitt stellt ein paar Tools vor, die den CHASE in jeweils einer Variante umsetzen. Bei der Auswahl der Tools wurde hierbei abgewägt, inwiefern deren Anwendungsszenarien für das in dieser Arbeit zu entwickelnde Tool relevant sind, da dieses bestimmte Problemstellungen vorwiegend abdecken soll.

Die Verfügbarkeit und Dokumentation des Quellcodes der Tools spielt außerdem auch eine Rolle, um bestimmte Techniken zu verstehen und zu verwenden, die für die Implementierung der Tools verwendet wurden, und um eine Einarbeitung in den Quellcode zu vereinfachen. Gegebenenfalls dienen die Tools direkt als Gerüst für die eigene Implementierung.

3.2.1. PDQ

Für die semantische Optimierung von Anfragen wurde das Tool PDQ entwickelt [BLT14], welches Selektions-Projektions-Verbund-Anfragen und Schemata bestehend aus Beschreibungen von Relationen und Integritätsbedingungen als Eingabe nimmt und aus ihnen Anfragepläne erstellt, die mittels Kostenfunktionen bewertet werden. PDQ steht für "Proof-Driven Query Planning" und optimiert Anfragen durch den Beweis ihrer Beantwortbarkeit. Jeder Beweis liefert einen Anfrageplan, der durch eine Kostenfunktion mit anderen gefundenen Anfrageplänen bewertet und verglichen wird. PDQ wurde in der Programmiersprache Java geschrieben und verwendet XML als Ein- und Ausgabe.
Das Tool wurde im Rahmen einer Projektveranstaltung des Lehrstuhls DBIS der Universität Rostock getestet. Es wurden sechs Probleme untersucht bestehend aus Relationenschemata und Integritätsbedingungen, zu denen zu einer gestellten Anfrage ein optimaler Anfrageplan ermittelt werden soll. Von den sechs Problemen sind jeweils drei als einfach und kompliziert eingestuft. PDQ schaffte es bei fünf der sechs Tests, einen optimalen Anfrageplan zu bestimmen. Einer der Tests wird in Abbildung 3.3 gezeigt mit Abbildung 3.4a und Abbildung 3.4b als Eingaben und Abbildung 3.5 als Ausgabe. Abbildung 3.4a stellt die Anfrage aus Abbildung 3.3 dar. Der Verbund ist dabei implizit durch das Angeben von zwei Schemata mit einem gemeinsamen Attribut `z` enthalten. Im unteren Teil von Abbildung 3.5 unter dem `type`-Attribut `PROJECT` ist zu erkennen, dass, wie in Abbildung 3.3 erwartet, die optimierte Anfrage, die Projektion, auf $r(R_1)$ ausgeführt und der Verbund mit $r(R_2)$ ausgelassen wird.
Aufgrund der verzweigten Komplexität des Quellcodes wird PDQ jedoch nicht für die Entwicklung des eigenen CHASE-Tools verwendet.

Schema: $R_1(A_1, A_2), R_2(A_2, A_3)$

TGD: $(\forall a_1, a_2 : R_1(a_1, a_2) \rightarrow \exists a_3 : R_2(a_2, a_3))$

Anfrage: $\pi_{a_1}(r(R_1) \bowtie r(R_2))$

Datalog: $R'(a_1) \leftarrow R_1(a_1, b_1), R_2(b_1, b_2)$

Ziel: optimierte Anfrage $\pi_{a_1}(r(R_1))$

Abbildung 3.3. Projektbeispiel

```
<?xml version="1.0" encoding="UTF-8"?>
<query type="conjunctive">
    <body>
        <atom name="t1">
            <variable name="y" />
            <variable name="z" />
        </atom>
        <atom name="t2">
            <variable name="z" />
            <variable name="w" />
        </atom>
    </body>
    <head name="Q">
        <variable name="y" />
    </head>
</query>
```

(a) im Projekt von PDQ genutztes Query.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<schema>
    <relations>
        <relation name="t1">
            <attribute name="a1" type="java.lang.Integer" />
            <attribute name="a2" type="java.lang.Integer" />
            <access-method name="m1.a1" type="FREE" cost="5" />
        </relation>

        <relation name="t2">
            <attribute name="a3" type="java.lang.Integer" />
            <attribute name="a4" type="java.lang.Integer" />
            <access-method name="m2.a3" type="FREE" cost="5" />
        </relation>
    </relations>
    <dependencies>
        <dependency>
            <body>
                <atom name="t1">
                    <variable name="y" />
                    <variable name="z" />
                </atom>
            </body>
            <head>
                <atom name="t2">
                    <variable name="z" />
                    <variable name="w" />
                </atom>
            </head>
        </dependency>
    </dependencies>
</schema>
```

(b) im Projekt von PDQ genutztes Schema.xml

Abbildung 3.4. PDQ Eingaben

```
<?xml version="1.0" encoding="UTF-8"?>
<plan type="linear" cost="5.0" control-flow="TOP_DOWN">
<command name="T1">
    <operator type="PROJECT">
        <outputs>
            <attribute name="c1" type="java.lang.Integer"/>
        </outputs>
        <project>
            <attribute name="c1" type="java.lang.Integer"/>
        </project>
        <child>
            <operator type="PROJECT">
                <outputs>
                    <attribute name="c1" type="java.lang.Integer"/>
                    <attribute name="c2" type="java.lang.Integer"/>
                </outputs>
                <project>
                    <attribute name="r1.a1" type="java.lang.Integer"/>
                    <attribute name="r1.a2" type="java.lang.Integer"/>
                </project>
                <child>
                    <operator type="ACCESS" relation="r1" access-method="m1.a1">
                        <outputs>
                            <attribute name="r1.a1" type="java.lang.Integer"/>
                            <attribute name="r1.a2" type="java.lang.Integer"/>
                        </outputs>
                    </operator>
                </child>
            </operator>
        </child>
    </operator>
</command>
</plan>
```

Abbildung 3.5. Beispielausgabe in PDQ

3.2.2. ProvCB

Mit dem Tool ProvCB [ICDK14] ist es möglich, Sichten in Anfragen durch Anfragetransformation einzuarbeiten. Dazu kommt hier das CHASE&BACKCHASE-Verfahren zum Einsatz, mit welchem die geCHASEten Anfragen so reduziert werden können, dass nur solche Anfragen übrig bleiben, die nur noch die eingearbeiteten Sichten statt der originalen Relationen enthalten. ProvCB verwendet hierbei einen *provenance-aware-CHASE*, der Provenance-Informationen verarbeitet, um hinzugefügte Verbunde zu den Teilanfragen des universellen Plans zurückzuverfolgen, die für die jeweiligen Verbunde verantwortlich waren. Auf diese Weise kann ProvCB die minimalen Umformulierungen direkt vom Ergebnis einer einzelnen CHASE-Anwendung auf den universellen Plan ablesen, ohne die exponentiell vielen Teilanfragen zu CHASEn, die durch das CHASE&BACKCHASE-Verfahren entstehen würden.
Aufgrund des organisierten und kommentierten Quellcode wird ProvCB als Orientierung für die Implementierung des CHASE-Tools dieser Arbeit verwendet, indem zum Teil die Klassenhierarchie und einige Eigenschaften mancher Klassen übernommen werden. ProvCB's Paketorganisation ist in Abbildung 3.6 zu sehen.

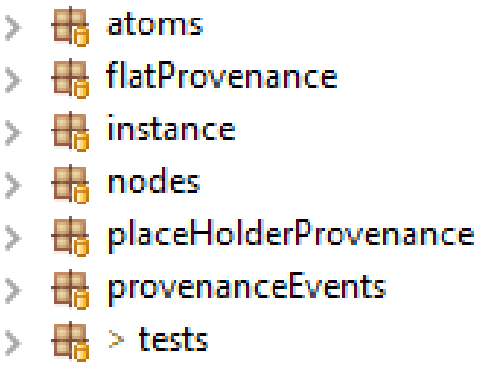


Abbildung 3.6. ProvCB Packetorganisation

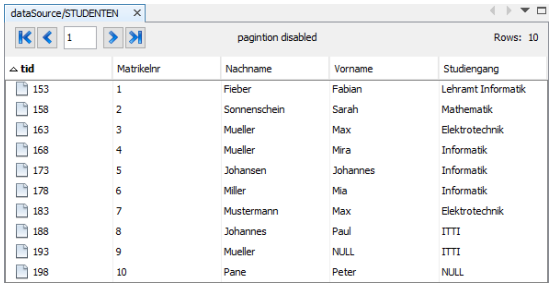
3.2.3. Llunatic

Llunatic [GMPS14] ist in der Lage, Zieldatenbanken aus gegebenen Quelldatenbanken und Abhängigkeiten in Form von s-t tgd's und egd's zu berechnen. Das Tool ist laut den Autoren das erste, welches

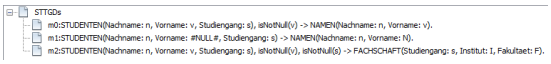
Datenintegrations- und Datensäuberungsszenarien vereinheitlicht lösen kann.
Ebenso wie PDQ wurde auch Llunatic in einer Projektveranstaltung getestet. Dabei wurden einige der mitgelieferten Anwendungsszenarien in Form von XML-Dateien und ein eigenes Beispiel mit einer Studentendatenbank getestet. Dieses Beispiel ist in Abbildung 3.7 zu sehen und entspricht dabei der STUDENTEN-Tabelle aus Tabelle 1.2 mit zwei zusätzlichen Einträgen mit Nullwerten beim Vornamen und Studiengang und den ID's 9 und 10. Die anderen Abbildungen stellen die Durchführung des Beispiels im GUI von Llunatic dar. Abbildung 3.8a und Abbildung 3.8b sind die Eingaben der Beispielanwendung dargestellt in Llunatic. Das durch die st tgd's in Abbildung 3.8b entstandene Schema und die aus Abbildung 3.8a entstandene Relation sind in Abbildung 3.9a und Abbildung 3.9b zu sehen.
Aufgrund des undokumentierten Quellcodes von Llunatic, welcher sich über mehr als einem Megabyte und mehreren hundert Dateien erstreckt, ist eine direkte Weiterentwicklung und Nutzung als Gerüst für die eigene Implementierung ausgeschlossen.

```
<?xml version="1.0" encoding="UTF-8"?>
<scenario>
  <source>
    <type>GENERATE</type>
    <generate>
<![CDATA[
SCHEMA:
STUDENTEN(Matrikelnr, Nachname, Vorname, Studiengang)
INSTANCE:
STUDENTEN(Matrikelnr: 1, Nachname: "Fieber", Vorname: "Fabian", Studiengang: "Lehramt Informatik")
STUDENTEN(Matrikelnr: 2, Nachname: "Sonnenschein", Vorname: "Sarah", Studiengang: "Mathematik")
STUDENTEN(Matrikelnr: 3, Nachname: "Mueller", Vorname: "Max", Studiengang: "Elektrotechnik")
STUDENTEN(Matrikelnr: 4, Nachname: "Mueller", Vorname: "Mira", Studiengang: "Informatik")
STUDENTEN(Matrikelnr: 5, Nachname: "Johansen", Vorname: "Johannes", Studiengang: "Informatik")
STUDENTEN(Matrikelnr: 6, Nachname: "Miller", Vorname: "Mia", Studiengang: "Informatik")
STUDENTEN(Matrikelnr: 7, Nachname: "Mustermann", Vorname: "Max", Studiengang: "Elektrotechnik")
STUDENTEN(Matrikelnr: 8, Nachname: "Johannes", Vorname: "Paul", Studiengang: "ITTI")
STUDENTEN(Matrikelnr: 9, Nachname: "Mueller", Vorname: #NULL#, Studiengang: "ITTI")
STUDENTEN(Matrikelnr:10, Nachname: "Pane", Vorname: "Peter", Studiengang: #NULL#)
]]>
    </generate>
  </source>
  <target>
    <type>GENERATE</type>
    <generate>
<![CDATA[
SCHEMA:
NAMEN(Nachname, Vorname)
FACHSCHAFT(Studiengang, Institut, Fakultaet)
]]>
    </generate>
  </target>
  <dependencies>
<![CDATA[
STTGDs:
STUDENTEN(Nachname: $n, Vorname: $v, Studiengang: $s), {isNotNull($v)} -> NAMEN(Nachname: $n, Vorname: $v).
STUDENTEN(Nachname: $n, Vorname: #NULL#, Studiengang: $s) -> NAMEN(Nachname: $n, Vorname: $N).
STUDENTEN(Nachname: $n, Vorname: $v, Studiengang: $s), {isNotNull($v)}, {isNotNull($s)}
-> FACHSCHAFT(Studiengang: $s, Institut: $I, Fakultaet: $F).
]]>
  </dependencies>
  <partialOrder/>
</scenario>
```

Abbildung 3.7. Beispieleingabe in Llunatic



(a) im Projekt von Llunatic genutzte Studenten

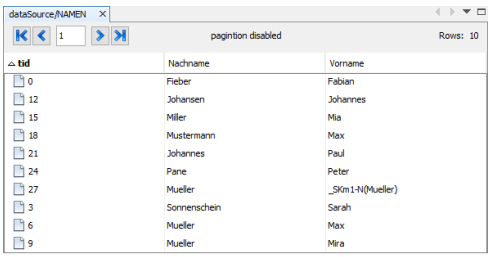


(b) im Projekt von Llunatic genutzte s-t tgd's

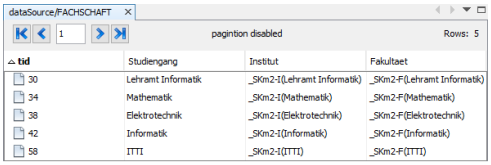
Abbildung 3.8. In Llunatic verbildlichte Eingabe

3.2.4. ChaseFUN

ChaseFUN [BIL17] kann wie Llunatic Datenaustausch und -integration durchführen und Zieldatenbanken aus Quelldatenbanken und Abhängigkeiten berechnen. Das Tool wurde in Java geschrieben und bietet Features für eine detailliertere Einsicht in den Datenaustauschprozess während der CHASE-Ausführung. Intern verwendet ChaseFUN Saturation Sets und Konfliktgraphen. Saturation Sets sind dabei Gruppierungen von Zuweisungen, die irgendwann durch FD's bzw. egd's miteinander interagieren würden.



(a) im Projekt von Llunatic ausgegebene Namen



(b) im Projekt von Llunatic ausgegebene Fachschaft

Abbildung 3.9. In Llunatic verbildlichte Ausgabe

Beim CHASEn eines solchen Saturation Sets werden alternierend ein tgd-Schritt und eine Reihe von egd-Schritten durchgeführt. Die Knoten des Konfliktgraphen repräsentieren s-t tgd's, die jeweils Zuweisungen haben, die sich im selben Saturation Set befinden, falls die s-t tgd's durch eine Kante im Konfliktgraphen miteinander verbunden sind.
In Abbildung 3.10 und Abbildung 3.11 ist das Beispiel aus [BIL17] abgebildet. In Abbildung 3.10 befinden sich die Quell- und Zielinstanz und die Abhängigkeiten in Form von s-t tgd's. Abbildung 3.11 (i) zeigt die Zuweisungen der Variablen in den s-t tgd's durch die korrespondierenden Werte aus der Quellinstanz. Den existenzquantifizierten Variablen werden dabei neue markierte Nullwerte zugewiesen, die einen globalen inkrementierenden Index teilen. Die Saturation Sets sind nun Mengen dieser Zuweisungen, wobei bestimmte Zuweisungen identische Werte haben. Das Saturation Set S_1 in Abbildung 3.11 (ii) beispielsweise enthält die Zuweisungen, in denen "Leonardo Di Caprio" als Namen vorkommen.
Leider ist der Quellcode von ChaseFUN nicht öffentlich zugänglich, weshalb es nicht möglich ist, dieses Tool für die eigene Implementierung in irgendeiner Form zu verwenden.

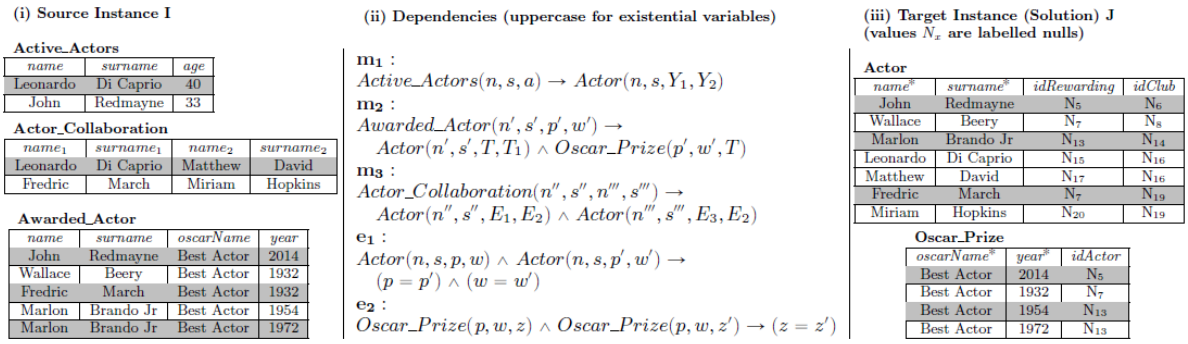


Abbildung 3.10. ChaseFUN Quell-, Zielinstanz und Abhängigkeiten

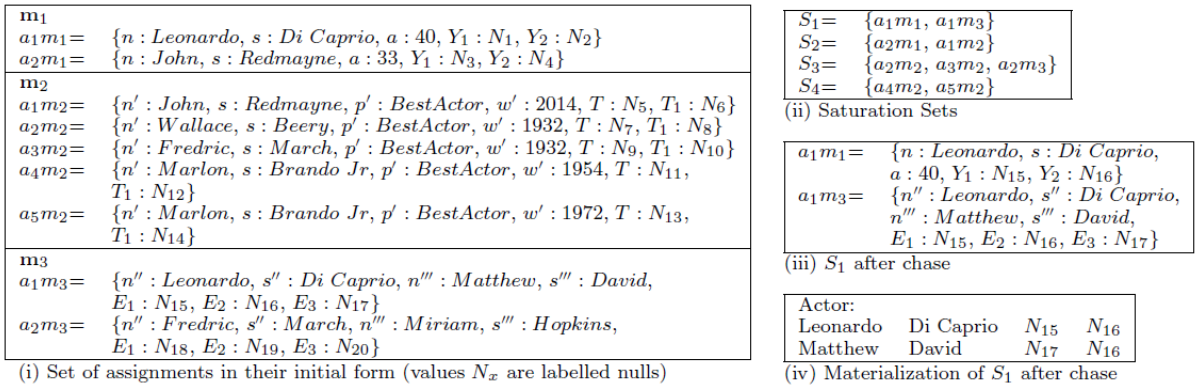


Abbildung 3.11. ChaseFUN Zuweisungen und Saturation Sets

3.2.5. Zusammenfassung und Entscheidung

Die in diesem Kapitel vorgestellten Tools sind jeweils auf eine bestimmte CHASE-Anwendung spezialisiert und lassen sich aufgrund der ebenfalls auf die Anwendungen spezialisierten Projektstrukturen und des größtenteils undokumentierten Quellcodes dementsprechend nur schwierig bis gar nicht auf ein allgemeineres Konzept ändern oder gar erweitern. Ebenfalls ist die vollständige Einarbeitung zum Verständnis des Codes und der Projektstruktur zu aufwändig, um eine mögliche Übernahme zur Weiterarbeit an den Tools in Betracht zu ziehen. In manchen Fällen ist der Quellcode noch nicht einmal ohne Weiteres verfügbar. Eine kurze Übersicht mit den betrachteten Tools und der begründeten Entscheidung bezüglich ihrer Verwendung befindet sich in Tabelle 3.2.
Die Tools sind immer auf ein oder zwei Anwendungen beschränkt, was nochmals Tabelle 3.3 zeigt. Das Ziel dieser Arbeit ist aber die Vorbereitung eines allgemeinen Tools für die vier farblich markierten Anwendungen in Tabelle 3.3. Aufgrund der genannten Umstände zur Weiternutzung der bestehenden Tools, wird das Tool für diese Arbeit von Grund auf neu implementiert. Die besprochenen theoretischen Grundlagen aus Kapitel 2 sollen als Richtlinie der Projektstruktur dienen. Die Vorstellung des Tools und eine genaue Einsicht in die konkrete Umsetzung der Theorie erfolgt im anschließenden Kapitel.

Tool	⋆	◯	Entscheidung	Grund
PDQ	Abhängigkeiten	Anfragen	keine Weiterentwicklung	Quellcodeeinarbeitung/-erweiterung zu aufwändig
ProvCB	Sichten	Anfragen	keine Weiterentwicklung	Quellcodeeinarbeitung/-erweiterung zu aufwändig
Llunatic	s-t tgd's, egd's	Quelldatenbank	keine Weiterentwicklung	Quellcodeeinarbeitung/-erweiterung zu aufwändig
ChaseFUN	s-t tgd's, egd's	Quelldatenbank	keine Weiterentwicklung	Quellcode nicht verfügbar

Tabelle 3.2. Überblick der betrachteten Tools

	⋆	◯	Ergebnis	Ziel	Tool
0.	Abhängigkeiten	DB-Schema	DB-Schema mit Integritätsbedingungen	optimierter DB-Entwurf	
I.	Abhängigkeiten	Anfragen	Anfragen	Semantische Optimierung	PDQ
II.	Sichten	Anfragen	Anfragen auf Sichten	AQuV	ProvCB
II'.	Operationen	Anfragen	Anfragen auf Operationen	AQuO	
III.	s-t tgd's, egd's, tgd's	Quell-DB	Ziel-DB	Datenaustausch, Datenintegration	Llunatic, ChaseFUN
IV.	tgd's, egd's	DB	modifizierte DB	Cleaning	Llunatic, ChaseFUN
V.	tgd's, egd's	unvollständige DB	Anfrageergebnis	sichere Antworten	
VI.	s-t tgd's, egd's, tgd's	DB	Anfrageergebnis	invertierbare Auswertung	

Tabelle 3.3. Überblick von CHASE-Anwendungen und zugehörigen Parametern

4. Eigene Konzeptentwicklung und Implementierung

In diesem Kapitel werden das Konzept für die Implementierung des CHASE-Tools und die Implementierung selbst vorgestellt. Zur Implementierung wurden die Programmiersprache Java Version 8 und die Entwicklungsumgebung Eclipse verwendet.
Die theoretische Ausarbeitung dieser Arbeit soll als Vorbild für das Konzept des CHASE-Tools dienen. Der Grundgedanke, die verschiedenen Anwendungsfälle des CHASE in einem einzigen, universalen Tool anwendbar zu machen, kommt daher, dass die CHASE-Objekte und -Parameter im Wesentlichen untereinander austauschbar sind, ohne dass sich die Ausführung des CHASE großartig verändert. Während in Anfragen Variablen durch andere Variablen oder Konstanten ausgetauscht werden können, werden in Datenbankinstanzen Nullwerte durch andere Nullwerte oder Konstanten ersetzt. Diese Hierarchie von Termen, die bereits durch die Definition des CHASE-Schrittes herausgebildet und in Abbildung 2.1 verbildlicht wurde, führt dazu, dass sich die verschiedenen Anwendungsfälle auf eine Ebene bringen lassen können. Jedoch ist die Implementierungsarbeit lediglich auf den Anwendungsfall mit Datenbankinstanzen als Objekt und tgd's und egd's als Parameter aufgrund von zeitlichen Restriktionen der Masterarbeit beschränkt.
Der Kern des Tools ist der zuvor vorgestellte CHASE-Algorithmus. Die Eingabe des Algorithmus' ist eine Instanz zusammen mit einer Menge an Integritätsbedingungen, die jeweils tgd's oder egd's sein können. Die Ausgabe ist eine zur Eingabe veränderte Instanz, die die angegebenen Integritätsbedingungen erfüllt. In bestimmten Fällen kann der Algorithmus auch bewusst fehlschlagen, in welchem Fall eine leere Instanz ausgegeben wird, um `NullPointerException`s in einem eventuellen übergeordneten Programm zu vermeiden. Zusätzlich macht eine implementierte Textausgabe auf den Fehlschlag und auf die leere Instanz als Rückgabe aufmerksam. Dazu werden außerdem die beiden für den Fehlschlag verantwortlichen Terme in der Textausgabe mitgenannt.
Der leitende Gedanke bei der Entwicklung des Tools war gewesen, dass der Großteil aus der Theorie möglichst eins-zu-eins im Tool umgesetzt wird. Viele Konzepte der Definitionen aus Kapitel 2 wie Terme, Atome, Instanzen oder Integritätsbedingungen wurden daher jeweils als Datenstrukturen, insbesondere als Klassen, beschrieben. Bei den Quellcodeausschnitten zu den entsprechenden Klassen werden bei den Methoden nur die Methodenbezeichnungen mit ihren Parametern angegeben, um für Übersichtlichkeit zu sorgen. Die Funktionen, Parameter und Rückgabewerte der Methoden werden in den einzelnen Abschnitten anhand von Programmbeispielen am Quellcode genauer erklärt. Auf die Konstruktoren, Getter und Setter, die hier prinzipiell für jedes Klassenattribut implementiert wurden, wird jedoch ganz verzichtet. Eine Ausnahme bildet hier `getTermValue` von der Klasse `Term`. Auch überschriebene Standardmethoden wie `toString`, `hashCode` und `equals` werden hier nicht berücksichtigt.
Zu jeder Klasse gibt es außerdem ein Beispielprogramm, das die Erstellung und Handhabung der Objekte der jeweiligen Klassen aufzeigt und deren Methoden demonstriert. Der Quellcode dieser Beispielprogramme wird hier zur einfacheren Nachvollziehbarkeit mitangegeben.
Die am häufigsten genutzte Datenstruktur ist `HashSet`, was aufgrund der Anlehnung an eine Menge geeignet ist, um vieles aus der Theorie in der Implementierung abzubilden. Mit `HashSet`s gibt es allerdings keinen indexbasierten Zugriff auf die Elemente oder eine bestimmte Reihenfolge dieser. Für Fälle, in denen die Reihenfolge der Elemente doch eine Rolle spielt wie beispielsweise bei Tupeln und deren

Zuordnung zu bestimmten Attributen, werden stattdessen `ArrayLists` verwendet.
Abbildung 4.1 stellt eine Übersicht des Projektes mit den verwendeten Klassen und ihren Beziehungen zueinander als UML-Diagramm dar. Anhand der vielen Beziehungen zu anderen Klassen ist zu sehen, dass die `Term`-Klasse elementar für die `Atom`- und `TermMapping`-Klassen ist. Die Atome wiederum bilden die Basis für die Klassen `Instance` und die der Integritätsbedingungen `IntegrityConstraint`, `Tgd` und `Egd`.
Die folgenden Anschnitte des Kapitels beschreiben jeweils die einzelnen Klassen, die für die Umsetzung des CHASE benötigt werden. Dazu wird beschrieben, aus welchen Variablen die Klassen zusammengesetzt sind und welche Methoden ihnen zur Verfügung stehen. Auf die Methoden wird ebenfalls einzeln eingegangen, indem deren Parameter und Rückgabewerte erklärt werden und wie diese zustande kommen. Das wird am Ende eines jeden Abschnitts dann nochmal mit einem Programmbeispiel demonstriert.

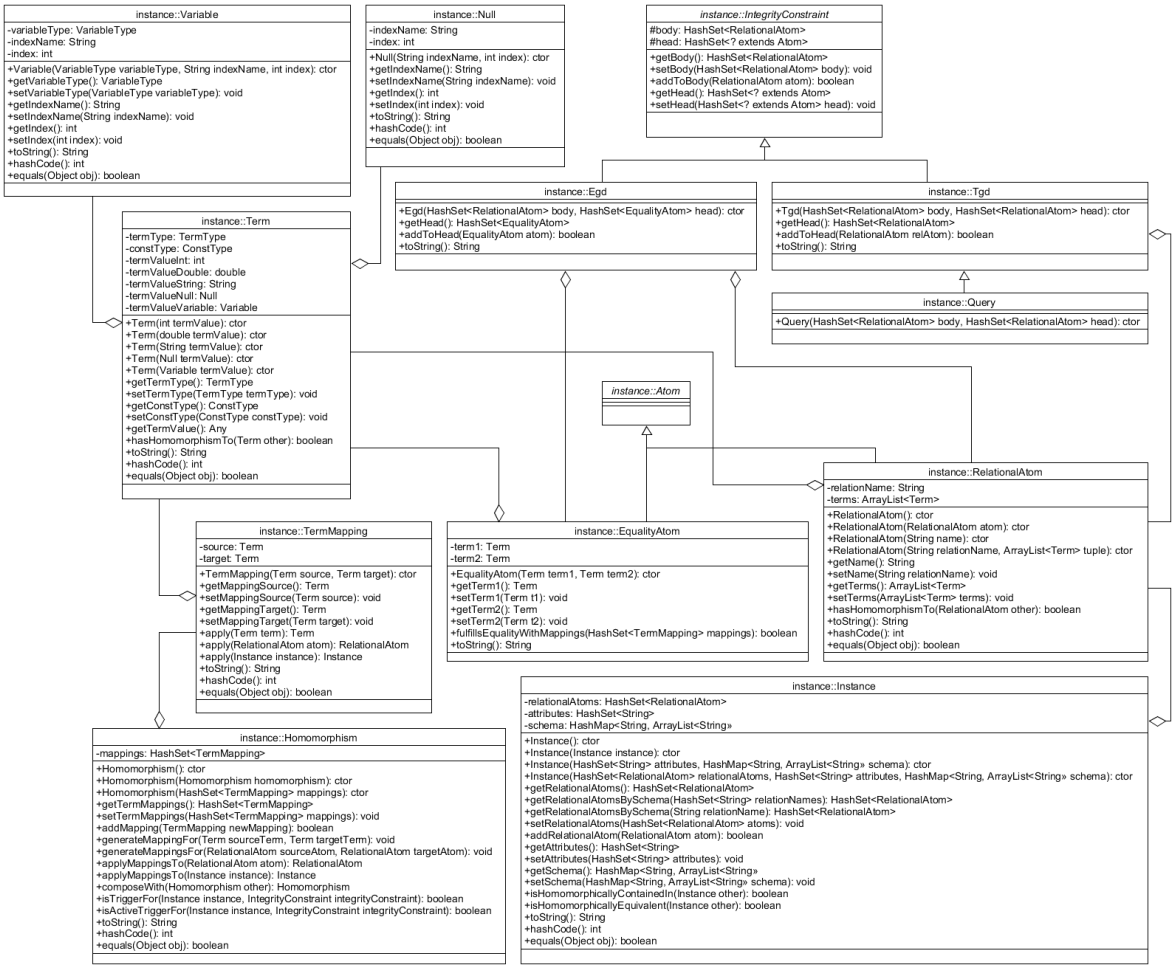


Abbildung 4.1. UML-Diagramm des CHASE-Tools

4.1. Terme

Terme stellen in dem CHASE-Tool eine elementare Datenstruktur dar und werden als Klasse `Term` repräsentiert. Sie können ebenso wie in der Theorie eine Variable, ein Nullwert oder eine Konstante sein. Für die Konstanten werden einfache Datentypen verwendet. Da Konstanten jedoch sowohl Zeichenketten als auch Zahlen oder andere Werte sein können und während der Laufzeit nicht bekannt ist, welchen Datentyp eine Konstante hat, ist der hier verwendete Ansatz dynamische Typisierung für die Ausgabefunktion

des Wertes der Konstanten. Die zur Zeit unterstützten Datentypen sind String, Double und Integer. Für jeden Datentypen gibt es eine Feldvariable der Klasse, die den entsprechenden Wert speichert. Ein Term kann immer nur einen Wert von einem Typen enthalten, weshalb immer nur eine dieser Feldvariablen belegt wird.
Für die Extrahierung des Termwertes wird die Methode `getTermValue` genutzt. Der Rückgabetyp ist dabei `<Any>`, was die dynamische Typisierung simuliert. In der tatsächlichen Anwendung der Funktion muss der Rückgabewert dann noch korrekt zum richtigen Datentypen gecastet werden, um mit dem Wert weiter arbeiten zu können. Dies kann allerdings zu Fehlern führen, falls der erwartete Rückgabewert nicht den entsprechenden Datentypen hat. Daher wird mithilfe einer Enumeration `ConstType` der Typ des Konstantenterms ermittelt, wodurch ein korrektes Casting ermöglicht wird.
Nullwerte und Variablen werden als Strings repräsentiert. Zur Unterscheidung von Nullwerten, Variablen und konstanten Stringwerten werden bei den Nullwerten und Variablen identifizierende Zeichenketten am Anfang der Strings verwendet. Diese sind `#V` für gegebene Variablen, `#E` für existenzquantifizierte Variablen in Integritätsbedingungen und `#N` für Nullwerte in Instanzen. Außerdem ist bei Nullwerten und Variablen wichtig, dass sie demselben Bezeichnungsschema folgen wie zuvor in Abschnitt 2.3 eingeführt. Hier im Tool wird dies mit `#N_indexName_index` für beispielsweise Nullwerte gehandhabt. Der Teil `indexName` steht hier wieder repräsentativ für das Attribut, für das der Nullwert steht, und `index` für die Nummerierung der Nullwerte für `indexName`. Analog ist dies auch für Variablen der Fall. Dies ist hauptsächlich wichtig für die Eingabe von solchen Termen und für die Ausgabe und Lesbarkeit. Intern werden jedoch für Variablen und Nullwerte extra Klassen verwendet, um die Bezeichnung und den Index des Wertes speichern und leicht wieder extrahieren zu können. Die Ausgabe mit der `toString`-Methode erfolgt dann wieder im gleichen Format wie die Eingabe.
Zur Unterscheidung von Variablen, Nullwerten und Konstanten wird eine Enumeration `TermType` verwendet, die bei der Erstellung eines Terms bestimmt, um was für einen Term es sich handelt. Im Fall einer Konstanten sorgt die zusätzliche bereits erwähnte Enumeration `ConstType` für die Unterscheidung des Datentyps des Terms. Für eine eventuelle Erweiterung des Tools um die Unterstützung von mehr Datentypen müssten diese Enumerations entsprechend erweitert werden.
Die Enumeration `TermType` besitzt außerdem ein Ranking, welches die Hierarchie von Konstanten, Nullwerten und Variablen bei Termabbildungen widerspiegelt. Konstanten besitzen dabei den Wert 0, Nullwerte den Wert 1 und Variablen den Wert 2. Bei Termabbildungen wird darauf geachtet, dass Terme mit einem höheren Wert durch Terme mit dem niedrigeren Wert ersetzt werden. Diese Hierarchie entstammt aus Abbildung 2.1 und [BKM+17] und zeigt in Algorithmus 1 in Zeile 17 Wirkung. Mit der Methode `isSmallerThan` von `TermType` kann die Wertigkeit von `TermType`s untereinander verglichen werden. Es wird `true` ausgegeben, wenn der interne Wert des ausführenden `TermType`s kleiner ist als der Wert des anderen.
Mit der Methode `hasHomomorphismTo` kann geprüft werden, ob ein Term einen Homomorphismus zu einem anderen Term besitzt. Das Ergebnis ist `true`, falls der Term, der die Methode ausführt,

- eine existenzquantifizierte Variable ist,
- eine gegebene Variable und der andere Term dieselbe Variable ist,
- ein Nullwert und der andere Term keine Variable ist und
- beide Terme dieselbe Konstante sind.

```
1  public class Term
2  {
3    private TermType termType;
4    private ConstType constType;
5
```

```
6    private int termValueInt;
7    private double termValueDouble;
8    private String termValueString;
9    private Null termValueNull;
10   private Variable termValueVariable;
11
12   public <Any> getTermValue();
13   public boolean hasHomomorphismTo(Term);
14 }
```

Listing 4.1 Klasse `Term` mit Feldvariablen und Methoden

Beispiel 4.1. *Das folgende Beispielprogramm dient zur Veranschaulichung der Erstellung von `Term`-Objekten. Erstellt werden diese in den Zeilen 1 - 5 mit den verschiedenen Term- und Konstantentypen. Gezeigt wird auch, dass der Datentyp des Rückgabewertes von `getTermValue` (Zeilen 7 und 8) der richtigen Variable zugewiesen werden muss, um Programmabstürze durch Datentyp-inkompatible Zuweisungen zu verhindern (Zeilen 8 und 15). Der `TermType` und `ConstType` werden ebenfalls zur Veranschaulichung ausgegeben (Zeilen 17 - 19). Außerdem wird bei zufällig bestimmten Termen mit `hasHomomorphismTo` überprüft, ob es einen Homomorphismus gibt (Zeilen 21 - 24).*

```
1  Term termInt = new Term(42);
2  Term termDouble = new Term(3.14);
3  Term termString = new Term("Konstante");
4  Term termNull = new Term("#N_vorname_1");
5  Term termVariable = new Term("#V_vorname_1");
6
7  int intValue = termInt.getTermValue();     // ok
8  //String stringValue = termInt.getTermValue();  // inkompatibel, aber keine
       Kompilierfehler
9
10 System.out.println(termInt);     // 42
11 System.out.println(termString);    // Konstante
12 System.out.println(termVariableV);  // #V_vorname_1
13
14 System.out.println(intValue);    // 42
15 //System.out.println(stringValue);  // Exception bei Ausführung
16
17 System.out.println(termDouble.getTermType()); // Const
18 System.out.println(termDouble.getConstType());  // Double
19 System.out.println(termNull.getTermType());    // Null
20
21 System.out.println(termInt.hasHomomorphismTo(termNull));     // false
22 System.out.println(termNull.hasHomomorphismTo(termString));    // true
23 System.out.println(termVariableV.hasHomomorphismTo(termVariableE)); // false
24 System.out.println(termVariableE.hasHomomorphismTo(termVariableV)); // true
```

Listing 4.2 Beispielprogramm für `Term`

4.2. Variablen und Nullwerte

Die `Variable`- und `Null`-Klasse repräsentieren Variablen in Integritätsbedingungen und Nullwerte in Instanzen. Beide Klassen besitzen die Feldvariablen `indexName` als `String` und `index` als Integer. Wie in der `Term`-Klasse bereits erwähnt, stehen der `indexName` für das Attribut, für das der Nullwert oder die Variable steht, und `index` für die Nummerierung der entsprechenden Nullwerte oder Variablen. Die `Variable`-Klasse besitzt noch als zusätzliche Feldvariable `variableType`, welche die Enumeration `VariableType` als Typ besitzt. Mit ihr wird bestimmt, ob es sich bei einer Variablen um eine gegebene oder existenzquantifizierte Variable handelt.
Die beiden Klassen stehen für sich und bieten keine Einschränkung bezüglich ihrer Nutzung. Es wird also nicht verhindert, dass beispielsweise Nullwerte in Integritätsbedingungen und Variablen in Instanzen verwendet werden können.

```
1 public class Null
2 {
3   private String indexName;
4   private int index;
5 }
```

Listing 4.3 Klasse `Null` mit Feldvariablen

```
1 public class Variable
2 {
3   private VariableType variableType;
4   private String indexName;
5   private int index;
6 }
```

Listing 4.4 Klasse `Variable` mit Feldvariablen

Beispiel 4.2. *Das Beispielprogramm zeigt die Erstellung von Variablen und Nullwerten (Zeilen 1 - 8). Anschließend werden diese als Text ausgegeben (Zeilen 10 - 15). Zudem wird gezeigt, dass sich Terme auch erstellen lassen, indem statt der festgesetzt formatierten Strings ein `Variable`- oder `Null`-Objekt übergeben werden kann (Zeilen 17 und 18).*

```
1  Variable varV1 = new Variable(VariableType.V, "vorname", 1);
2  Variable varV2 = new Variable(VariableType.V, "vorname", 2);
3
4  Variable varE1 = new Variable(VariableType.E, "nachname", 1);
5  Variable varE2 = new Variable(VariableType.E, "nachname", 2);
6
7  Null null1 = new Null("vorname", 1);
8  Null null2 = new Null("vorname", 2);
9
10 System.out.println(varV1);   // #V_vorname_1
11 System.out.println(varE2);   // #E_nachname_2
12 System.out.println(null1);   // #N_nachname_1
13 System.out.println(varE1.getVariableType());   // E
14 System.out.println(varV2.getIndexName());// vorname
15 System.out.println(null2.getIndex()); // 2
16
17 Term termVar = new Term(varV1);
```

```
18 Term termNull = new Term(null1);
19
20 System.out.println(termVar);   // #V_vorname_1
21 System.out.println(termNull); // #N_nachname_1
```

Listing 4.5 Beispielprogramm für `Variable` und `Null`

4.3. Atome

Atome sind ein weiterer Aspekt, der beim CHASE eine wichtige Rolle spielt. Unterschieden wird im Tool wie in der Theorie auch zwischen relationalen (`RelationalAtom`) und Gleichheitsatomen (`EqualityAtom`). Beide sind der Grundbaustein für Instanzen, Anfragen und Integritätsbedingungen. Relationale Atome selbst hingegen bestehen aus Termen und der Bezeichnung einer Relation. Entsprechend umgesetzt sind sie auch im CHASE-Tool mit einer `ArrayList` aus Termen und einer Stringvariablen, die die Relationenbezeichnung speichert. Gleichheitsatome enthalten zwei Terme, zwischen denen die Gleichheitsbeziehung ausgedrückt werden soll. Eine abstrakte Klasse `Atom` dient als Elternklasse für die beiden genannten Atom-Klassen. Die Klasse selbst besitzt weder Feldvariablen noch Methoden und dient lediglich dazu, in bestimmten Fällen Objekte von `RelationalAtom` und `EqualityAtom` zu nutzen, ohne zu wissen, um was für ein Atom es sich genau handelt.
Mit der Methode `hasHomomorphismTo` kann überprüft werden, ob es einen Homomorphismus von einem relationalen Atom zu einem anderen relationalen Atom gibt. Die Methode ruft dabei stellenweise die für Terme implementierte Methode `hasHomomorphismTo` auf und gibt `true` aus, falls es für jeden einzelnen Term des einen relationalen Atoms einen Homomorphismus zu den korrespondierenden Termen des anderen relationalen Atoms gibt. Der Wert `false` ist andernfalls das Ergebnis.
Für Gleichheitsatome prüft `fulfillsEqualityWithMappings` mit einer Menge von Termabbildungen, ob das Gleichheitsatom unter diesen Termabbildungen erfüllt ist. Dazu werden die beiden Terme des Gleichheitsatoms auf die Terme abgebildet, die in den Termabbildungen aufgeführt sind, und anschließend auf Gleichheit überprüft. Das Ergebnis ist `true`, falls die Gleichheit erfüllt ist, und `false` andernfalls.

```
1 public class RelationalAtom extends Atom
2 {
3   private String relationName;
4   private ArrayList<Term> terms;
5
6   public boolean hasHomomorphismTo(RelationalAtom);
7 }
```

Listing 4.6 Klasse `RelationalAtom` mit Feldvariablen und Methoden

```
1 public class EqualityAtom extends Atom
2 {
3   private Term term1;
4   private Term term2;
5
6   public boolean fulfillsEqualityWithMappings(HashSet<TermMapping>);
7 }
```

Listing 4.7 Klasse `EqualityAtom` mit Feldvariablen und Methoden

Beispiel 4.3. *Das Beispielprogramm zeigt die Erstellung von relationalen Atomen. Es werden zuerst* `ArrayLists` *von Termen benötigt, die den jeweiligen Atomen zugeordnet werden (Zeilen 6 - 18). Anschließend wird ein zusätzliches Atom mit Variablen erstellt (Zeilen 24 - 31). Mit diesem und den beiden zuvor erstellten Atomen wird dann die Ausführung der Methode* `hasHomomorphismTo` *demonstriert (Zeilen 35 - 38).*
Ein Beispielprogramm für `EqualityAtom` *folgt im Anschluss von Abschnitt 4.6, weil Termabbildungen für die Demonstration benötigt weden.*

```
1  // Definition von ArrayLists von Termen
2  ArrayList<Term> termsStudent1 = new ArrayList<Term>();
3  ArrayList<Term> termsStudent2 = new ArrayList<Term>();
4
5  // ArrayList 1 wird mit Termen befüllt
6  termsStudent1.add(new Term(1));
7  termsStudent1.add(new Term("Fieber"));
8  termsStudent1.add(new Term("Fabian"));
9  termsStudent1.add(new Term("Lehramt␣Informatik"));
10
11 // ArrayList 2 wird mit Termen befüllt
12 termsStudent2.add(new Term(2));
13 termsStudent2.add(new Term("Sonnenschein"));
14 termsStudent2.add(new Term("Sarah"));
15 termsStudent2.add(new Term("Mathematik"));
16
17 RelationalAtom atomStudent1 = new RelationalAtom("STUDENTEN", termsStudent1);
18 RelationalAtom atomStudent2 = new RelationalAtom("STUDENTEN", termsStudent2);
19
20 System.out.println(atomStudent1);// STUDENTEN(1, Fieber, Fabian, Lehramt
       Informatik)
21 System.out.println(atomStudent2);// STUDENTEN(2, Sonnenschein, Sarah, Mathematik
       )
22
23 // relationales Atom mit Variablen
24 ArrayList<Term> termsVariables = new ArrayList<Term>();
25
26 termsVariables.add(new Term("#V_matrikelnummer_1"));
27 termsVariables.add(new Term("#V_nachname_1"));
28 termsVariables.add(new Term("#V_vorname_1"));
29 termsVariables.add(new Term("#V_studiengang_1"));
30
31 RelationalAtom atomVariables = new RelationalAtom("STUDENTEN", termsVariables);
32
33 System.out.println(atomVariables);// STUDENTEN(#V_matrikelnummer_1, #
       V_nachname_1, #V_vorname_1, #V_studiengang_1)
34
35 System.out.println(atomStudent1.hasHomomorphismTo(atomStudent2)); // false
36 System.out.println(atomVariables.hasHomomorphismTo(atomStudent1));  // true
37 System.out.println(atomVariables.hasHomomorphismTo(atomStudent2));  // true
38 System.out.println(atomStudent1.hasHomomorphismTo(atomVariables));  // false
```

Listing 4.8 Beispielprogramm für `RelationalAtom`

4.4. Instanzen

Die Implementierung von Instanzen unterscheidet sich wie die vorigen Konzepte ebenfalls nicht von der entsprechenden theoretischen Definition. Da Instanzen eine Menge von Tupeln aus verschiedenen Relationen sind, wird hier in der `Instance`-Klasse ein `HashSet` aus relationalen Atomen verwendet, welche die Instanz bilden. Es werden außerdem ein `HashSet` aus Strings für die in der Instanz nutzbaren Attribute und eine `HashMap` für die zugelassenen Relationenschemata verwendet. Die `HashMap` nutzt den Relationennamen, einen String, als Schlüssel und eine `ArrayList` aus Strings als zugehörigen Wert. Diese `ArrayList`s enthalten die Attribute, die zum jeweiligen als Schlüssel verwendeten Relationenschema gehören.
Um nur Atome mit bestimmten Relationennamen zu extrahieren, kann die Methode `getRelationalAtomsBySchema` mit den Relationennamen als Parameter genutzt werden. Wie bei Atomen kann mithilfe der Methode `isHomomorphicallyContainedIn` überprüft werden, ob ein Homomorphismus von einer Instanz zu einer anderen Instanz existiert. Die Methode gibt `true` aus, falls zu jedem Atom der ausführenden Instanz ein Homomorphismus zu einem Atom der anderen Instanz existiert. Andernfalls ist das Ergebnis `false`.
Ob zwei Instanzen äquivalent sind, prüft die Methode `isHomomorphicallyEquivalent`. Mit ihr wird `isHomomorphicallyContainedIn` in beide Richtungen ausgeführt. Das heißt, das Ergebnis ist `true`, falls `isHomomorphicallyContainedIn` ausgehend von jeder der beiden zu überprüfenden Instanzen angewendet auf die jeweils andere ebenfalls `true` ist, und `false` andernfalls.

```
1  public class Instance
2  {
3    private HashSet<RelationalAtom> relationalAtoms;
4    private HashSet<String> attributes;
5    private HashMap<String, ArrayList<String>> schema;
6
7    public HashSet<RelationalAtom>
8        getRelationalAtomsBySchema(HashSet<String>);
9    public boolean isHomomorphicallyContainedIn(Instance);
10   public boolean isHomomorphicallyEquivalent(Instance);
11 }
```

Listing 4.9 Klasse `Instance` mit Feldvariablen und Methoden

Beispiel 4.4. *Das Beispielprogramm zeigt die Erstellung einer Instanz mit jeweils einem `HashSet` von relationalen Atomen (Zeilen 4 - 8) und Attributen (Zeilen 11 - 20) und einer `HashMap` für das Schema (Zeilen 23 - 39). Die Beispielinstanz enthält drei Atome, die in Zeile 46 einmal ausgegeben werden. Nach der Textausgabe der Instanz wird eine zweite Instanz erstellt, die sich zur ersten Instanz im ersten STUDENTEN-Atom unterscheidet, indem dieses Nullwerte enthält (Zeilen 52 - 67). Dann wird mit `isHomomorphicallyContainedIn` und `isHomomorphicallyEquivalent` angezeigt, ob es einen Homomorphismus von der einen zur anderen Instanz gibt bzw. ob beide Instanzen äquivalent sind (Zeilen 75 - 79).*

```
1  /* ... dieselben STUDENTEN-Definitionen wie im Beispiel zu RelationalAtom ... */
2  /* ... und Definition eines NOTEN-Atoms */
3  // Definition der relationalen Atome für die Instanz
4  HashSet<RelationalAtom> atoms = new HashSet<RelationalAtom>();
5
6  atoms.add(atomStudent1);
7  atoms.add(atomStudent2);
8  atoms.add(atomNote1);
```

```
9
10 // Definition von Attributen der Instanz
11 HashSet<String> attributes = new HashSet<String>();
12
13 attributes.add("matrikelnummer");
14 attributes.add("name");
15 attributes.add("vorname");
16 attributes.add("studiengang");
17
18 attributes.add("modulnummer");
19 attributes.add("semester");
20 attributes.add("note");
21
22 // Definition des Schemas der Instanz
23 HashMap<String, ArrayList<String>> dbSchema = new HashMap<String, ArrayList<
       String>>();
24
25 ArrayList<String> studentenSchema = new ArrayList<String>();
26 ArrayList<String> notenSchema = new ArrayList<String>();
27
28 studentenSchema.add("matrikelnummer");
29 studentenSchema.add("name");
30 studentenSchema.add("vorname");
31 studentenSchema.add("studiengang");
32
33 notenSchema.add("modulnummer");
34 notenSchema.add("matrikelnummer");
35 notenSchema.add("semester");
36 notenSchema.add("note");
37
38 dbSchema.put("STUDENTEN", studentenSchema);
39 dbSchema.put("NOTEN", notenSchema);
40
41 Instance instance = new Instance(atoms, attributes, dbSchema);
42
43 // STUDENTEN(1, Fieber, Fabian, Lehramt Informatik),
44 // STUDENTEN(2, Sonnenschein, Sarah, Mathematik),
45 // NOTEN(1, 1, SS 16, 2.0)
46 System.out.println(instance);
47
48 // Definition eines Studenten mit Nullwerten im Vor- und Nachnamen (statt "
       Fieber" und "Fabian")
49 ArrayList<Term> termsStudentNull = new ArrayList<Term>();
50
51 // ArrayList 1 wird mit Termen befüllt
52 termsStudentNull.add(new Term(1));
53 termsStudentNull.add(new Term("#N_nachname_1"));
54 termsStudentNull.add(new Term("#N_nachname_1"));
55 termsStudentNull.add(new Term("Lehramt␣Informatik"));
56
57 RelationalAtom atomStudentNull = new RelationalAtom("STUDENTEN",
       termsStudentNull);
58
```

```
59 HashSet<RelationalAtom> atomsNull = new HashSet<RelationalAtom>();
60
61 atomsNull.add(atomStudentNull);
62 atomsNull.add(atomStudent2);
63 atomsNull.add(atomNote1);
64
65 // Erstellung zweiter Instanz wie erste Instanz,
66 // allerdings mit Nullwerten im ersten Tupel
67 Instance instanceNulls = new Instance(atomsNull, attributes, dbSchema);
68
69 // STUDENTEN(1, #N_nachname_1, #N_nachname_1, Lehramt Informatik),
70 // STUDENTEN(2, Sonnenschein, Sarah, Mathematik),
71 // NOTEN(1, 1, SS 16, 2.0)
72 System.out.println(instanceNulls);
73
74 // false
75 System.out.println(instance.isHomomorphicallyContainedIn(instanceNulls));
76 // true
77 System.out.println(instanceNulls.isHomomorphicallyContainedIn(instance));
78 // false
79 System.out.println(instance.isHomomorphicallyEquivalent(instanceNulls));
```

Listing 4.10 Beispielprogramm für `Instance`

4.5. Integritätsbedingungen

Tgd's und egd's werden durch die Klassen `Tgd` und `Egd` repräsentiert, die von einer abstrakten Klasse `IntegrityConstraint` erben. Sowohl tgd's als auch egd's haben `body` und `head` als Feldvariablen, welche jeweils ein `HashSet` aus Atomen sind. Diese Variablen entsprechen dabei den Definitionen von "Rumpf" und "Kopf" der Integritätsbedingungen, weshalb alle Atome von `body` und `head` relationale Atome sind. Eine Ausnahme bildet hier `head` der Klasse `Egd`, welcher ein `HashSet` von Gleichheitsatomen ist. Sowohl `head` als auch `body` werden von `IntegrityConstraint` vererbt, womit die Klassen `Tgd` und `Egd` selbst keine eigenen Feldvariablen besitzen. Lediglich die Getter und Setter der beiden Klassen unterscheiden sich durch das Zurückgeben und Speichern von relationalen Atomen bzw. Gleichheitsatomen.
Aufgrund der Verschiedenheit des Inhalts von `head` bei tgd's (relationale Atome) und egd's (Gleichheitsatome) ist `head` in `IntegrityConstraint` als ein `HashSet` mit dem Parameter `? extends Atom` definiert, was bedeutet, dass dies ein `HashSet<RelationalAtom>` oder `HashSet<EqualityAtom>` sein kann. Entsprechend ist dies in den Unterklassen `Tgd` und `Egd` so umgesetzt worden.

```
1 public abstract class IntegrityConstraint
2 {
3   protected HashSet<RelationalAtom> body;
4   protected HashSet<? extends Atom> head;
5 }
```

Listing 4.11 Klasse `IntegrityConstraint` mit Feldvariablen

Beispiel 4.5. *Dieses Beispielprogramm zeigt die Erstellung und Textausgabe einer tgd (Zeilen 2 - 40) und einer egd (Zeilen 43 - 77). Der Rumpf der tgd besteht aus zwei Atomen (Zeilen 2 - 18) und der Kopf aus einem Atom (Zeilen 21 - 31). Die egd besteht ebenfalls aus zwei Atomen im Rumpf (Zeilen 43 - 55),*

wobei eines davon auch in der tgd verwendet wird (Zeilen 6 - 9 und 14). Der Kopf der egd besteht aus drei Gleichheitsatomen (Zeilen 58 - 66).

```
1  // Rumpf der tgd
2  ArrayList<Term> constraintBodyStudent1 = new ArrayList<Term>();
3  ArrayList<Term> constraintBodyTeilnehmer = new ArrayList<Term>();
4  HashSet<RelationalAtom> tgdBody = new HashSet<RelationalAtom>();
5
6  constraintBodyStudent1.add(new Term("#V_matrikelnummer_1"));
7  constraintBodyStudent1.add(new Term("#V_nachname_1"));
8  constraintBodyStudent1.add(new Term("#V_vorname_1"));
9  constraintBodyStudent1.add(new Term("#V_studiengang_1"));
10
11 constraintBodyTeilnehmer.add(new Term("#V_modulnummer_1"));
12 constraintBodyTeilnehmer.add(new Term("#V_matrikelnummer_1"));
13
14 RelationalAtom tgdBodyAtom1 = new RelationalAtom("STUDENTEN",
       constraintBodyStudent1);
15 RelationalAtom tgdBodyAtom2 = new RelationalAtom("TEILNEHMER",
       constraintBodyTeilnehmer);
16
17 tgdBody.add(tgdBodyAtom1);
18 tgdBody.add(tgdBodyAtom2);
19
20 // Kopf der tgd
21 ArrayList<Term> tgdHeadNoten = new ArrayList<Term>();
22 HashSet<RelationalAtom> tgdHead = new HashSet<RelationalAtom>();
23
24 tgdHeadNoten.add(new Term("#V_modulnummer_1"));
25 tgdHeadNoten.add(new Term("#V_matrikelnummer_1"));
26 tgdHeadNoten.add(new Term("#E_semester_1"));
27 tgdHeadNoten.add(new Term("#E_note_1"));
28
29 RelationalAtom tgdHeadAtom1 = new RelationalAtom("NOTEN", tgdHeadNoten);
30
31 tgdHead.add(tgdHeadAtom1);
32
33 // Definition der tgd
34 Tgd tgd = new Tgd(tgdBody, tgdHead);
35
36 // STUDENTEN(#V_matrikelnummer_1, #V_nachname_1, #V_vorname_1, #V_studiengang_1)
       ,
37 // TEILNEHMER(#V_modulnummer_1, #V_matrikelnummer_1)
38 // ->
39 // NOTEN(#V_modulnummer_1, #V_matrikelnummer_1, #E_semester_1, #E_note_1)
40 System.out.println(tgd);
41
42 // Rumpf der egd
43 ArrayList<Term> constraintBodyStudent2 = new ArrayList<Term>();
44 HashSet<RelationalAtom> egdBody = new HashSet<RelationalAtom>();
45
46 constraintBodyStudent2.add(new Term("#V_matrikelnummer_1"));
47 constraintBodyStudent2.add(new Term("#V_nachname_2"));
```

```
48 constraintBodyStudent2.add(new Term("#V_vorname_2"));
49 constraintBodyStudent2.add(new Term("#V_studiengang_2"));
50
51 RelationalAtom egdBodyAtom1 = new RelationalAtom("STUDENTEN",
       constraintBodyStudent2);
52
53 // Mitnutzung des STUDENTEN-Atoms der tgd
54 egdBody.add(tgdBodyAtom1);
55 egdBody.add(egdBodyAtom1);
56
57 // Kopf der egd
58 EqualityAtom egdHeadAtom1 = new EqualityAtom(new Term("#V_nachname_1"), new Term
       ("#V_nachname_2"));
59 EqualityAtom egdHeadAtom2 = new EqualityAtom(new Term("#V_vorname_1"), new Term(
       "#V_vorname_2"));
60 EqualityAtom egdHeadAtom3 = new EqualityAtom(new Term("#V_studiengang_1"), new
       Term("#V_studiengang_2"));
61
62 HashSet<EqualityAtom> egdHead = new HashSet<EqualityAtom>();
63
64 egdHead.add(egdHeadAtom1);
65 egdHead.add(egdHeadAtom2);
66 egdHead.add(egdHeadAtom3);
67
68 // Definition der egd
69 Egd egd = new Egd(egdBody, egdHead);
70
71 // STUDENTEN(#V_matrikelnummer_1, #V_nachname_1, #V_vorname_1, #V_studiengang_1)
       ,
72 // STUDENTEN(#V_matrikelnummer_1, #V_nachname_2, #V_vorname_2, #V_studiengang_2)
73 // ->
74 // #V_nachname_1 = #V_nachname_2,
75 // #V_vorname_1 = #V_vorname_2,
76 // #V_studiengang_1 = #V_studiengang_2
77 System.out.println(egd);
```

Listing 4.12 Beispielprogramm für `IntegrityConstraint`

4.6. Termabbildungen

Die `TermMapping`-Klasse repräsentiert Termabbildungen. Zwei Feldvariablen `source` und `target`, welche jeweils vom Typ `Term` sind, stellen die Quelle und das Ziel der Termabbildung dar. Eine Termabbildung bildet einen Term auf einen anderen Term ab. Insbesondere werden hier solche Termabbildungen behandelt, die von einer Variablen oder einem Nullwert auf einen anderen Term abbilden. Termabbildungen von einer Konstanten auf einen anderen Term können zwar als `TermMapping` existieren, werden jedoch in der Methode `apply` nicht ausgeführt. Beispiele für Termabbildungen sind $x \to y$, $\eta_{\text{name}_1} \to \text{Max}$ und $\upsilon_{\text{fach}_7} \to \upsilon_{\text{fach}_2}$.
Mittels der Methode `apply` kann die Termabbildung auf einen Term, ein relationales Atom oder eine Instanz angewandt werden, wodurch die durch die Abbildung betroffenen Terme jeweils durch das Ziel

der Abbildung ersetzt werden. Dabei werden die als Argument übergebenen Terme, Atome und Instanzen nicht verändert oder zerstört, sondern es werden jeweils neue Objekte erzeugt, auf die die Abbildung ausgehend von den ursprünglichen Objekten angewandt wird.

```
1 public class TermMapping
2 {
3   private Term source;
4   private Term target;
5
6   public Term apply(Term);
7   public RelationalAtom apply(RelationalAtom);
8   public Instance apply(Instance);
9 }
```

Listing 4.13 Klasse `TermMapping` mit Feldvariablen und Methoden

Beispiel 4.6. *Dieses Beispielprogramm zeigt die Erstellung von Termabbildungen (Zeilen 1 und 2) und deren Anwendung durch* `apply` *jeweils auf einen Term (Zeilen 9 und 10), ein relationales Atom (Zeile 11) und eine Instanz (Zeilen 12 und 13). Außerdem wird durch das Anwenden von Termabbildungen die Erstellung von* `EqualityAtom`*-Objekten und deren Ausführung der Methode* `fulfillsEqualityWithMappings` *gezeigt. Dies wird am Ende des Programms mit demselben Gleichheitsatom aber mit verschiedenen Mengen von Termabbildungen gezeigt, bei denen verschiedene Ergebnisse ermittelt werden (Zeilen 31 - 37).*

```
1  TermMapping mapping1 = new TermMapping(new Term("#N_nachname_1"), new Term("
       Fieber"));
2  TermMapping mapping2 = new TermMapping(new Term("#N_vorname_1"), new Term("
       Fabian"));
3
4  System.out.println(mapping1); // #N_nachname_1 -> Fieber
5  System.out.println(mapping2); // #N_vorname_1 -> Fabian
6
7  /* ... Definition einer Instanz mit einem STUDENTEN-Atom mit Nullwerten beim Vor
       - und Nachnamen (wie im Beispielprogramm zu Instanzen) ... */
8
9  System.out.println(mapping1.apply(new Term("#N_nachname_1"))); // Fieber
10 System.out.println(mapping1.apply(new Term("#N_vorname_1"))); // #N_vorname_1
11 System.out.println(mapping1.apply(atomStudent)); // STUDENTEN(1, Fieber, #
       N_vorname_1, Lehramt Informatik)
12 System.out.println(mapping2.apply(instance)); // STUDENTEN(1, #N_nachname_1,
       Fabian, Lehramt Informatik)
13 System.out.println(mapping1.apply(mapping2.apply(instance))); // STUDENTEN(1,
       Fieber, Fabian, Lehramt Informatik)
14
15 // Beispiel zu EqualityAtom
16 TermMapping mapping1 = new TermMapping(new Term("#V_vorname_1"), new Term("Sarah
       "));
17 TermMapping mapping2 = new TermMapping(new Term("#V_vorname_2"), new Term("
       Fabian"));
18 TermMapping mapping3 = new TermMapping(new Term("#V_vorname_2"), new Term("Sarah
       "));
19
20 HashSet<TermMapping> mappings = new HashSet<TermMapping>();
```

```
21
22 mappings.add(mapping1);
23 mappings.add(mapping2);
24
25 System.out.println(mapping1); // #V_vorname_1 -> Fieber
26 System.out.println(mapping2); // #V_vorname_2 -> Fabian
27
28 EqualityAtom eAtom = new EqualityAtom(new Term("#V_vorname_1"), new Term("#
       V_vorname_2"));
29
30 System.out.println(eAtom);   // #V_vorname_1 = #V_vorname_2
31 System.out.println(eAtom.fulfillsEqualityWithMappings(mappings)); // false
32
33 // mapping2 aus der Mapping-Liste entfernen und mapping3 stattdessen hinzufügen
34 mappings.remove(mapping2);
35 mappings.add(mapping3);
36
37 System.out.println(eAtom.fulfillsEqualityWithMappings(mappings)); // true
```

Listing 4.14 Beispielprogramm für `TermMapping` und `EqualityAtom`

4.7. Homomorphismus

Homomorphismen haben auch ihre eigene Klasse `Homomorphism`. Ein Homomorphismus besitzt hier ein `HashSet` aus `TermMapping`s als Feldvariable. Dies ist dem nachempfunden, dass Homomorphismen Mengen von Termabbildungen sind.
Die Methode `generateMappingsFor` generiert für zwei gegebene Terme ein `TermMapping`, falls dies zwischen dem Quellterm und dem Zielterm möglich ist. Ein Mapping wird generiert, falls es einen Homomorphismus vom Quellterm zum Zielterm gibt, was mit der Term-Methode `hasHomomorphismTo` geprüft wird. Für zwei relationale Atome kann die Methode ebenfalls angewandt werden. Dabei wird eine Menge von Termabbildungen vom Quellatom zum Zielatom generiert und diese der Feldvariable `mappings` hinzugefügt.
Mit `applyMappingsTo` können die in einem Homomorphismus enthaltenen Mappings auf ein relationales Atom angewandt werden. Das dabei angegebene Atom, auf dem die Abbildungen angewandt werden sollen, wird dadurch nicht verändert, sondern es wird ein neues Atom als Kopie erstellt, auf welches dann die Abbildungen ausgeführt und welches anschließend als Ergebnis zurückgegeben wird.
Die Methode `composeWith` komponiert den ausführenden Homomorphismus mit dem als Parameter übergebenen. Für zwei Homomorphismen `h` und `p` entspricht der Methodenaufruf `h.composeWith(p)` dem mathematischen Ausdruck der Komposition von Abbildungen $(\mathtt{h} \circ \mathtt{p})(x) := \mathtt{p}(\mathtt{h}(x))$. Der Rückgabewert ist ein Homomorphismus, der die Komposition der beiden Homomorphismen darstellt.
Den beiden Methoden `isTriggerFor` und `isActiveTriggerFor` werden jeweils eine Instanz und eine Integritätsbedingung als Parameter übergeben. Für diese werden dann geprüft, ob es sich bei dem Homomorphismus, der diese Methoden ausführt, um einen Trigger bzw. um einen aktiven Trigger handelt. Beim Rückgabewert dieser beiden Methoden handelt es sich daher um ein `boolean`.
Die Methode `isTriggerFor` prüft, ob der Rumpf der Integritätsbedingung durch die Instanz erfüllt wird, indem für jedes relationale Atom aus dem Rumpf der Integritätsbedingung versucht wird, ein Homomorphismus zu einem Atom der Instanz zu finden, wobei die vorhandenen Termabbildungen des Homomorphismus auf die Atome der Integritätsbedingung vorher angewandt werden. Dadurch wird ausgeschlossen, dass für jedes Atom der Integritätsbedingung unabhängig von den anderen nach einem Homomorphismus

gesucht wird. So werden Variablen, die in verschiedenen Atomen vorkommen, gleich abgebildet. Lässt sich zu einem Atom der Integritätsbedingung kein Homomorphismus zur Instanz finden, ist der Rückgabewert `false`. Der Wert `true` wird hingegen zurückgegeben, falls sich zu jedem Atom der Bedingung ein Homomorphismus zur Instanz finden lässt.
Zu beachten ist bei `isActiveTriggerFor`, dass der Test auf einen aktiven Trigger normalerweise erfordert, zuerst zu testen, ob es sich bei einem Homomorphismus überhaupt um einen Trigger durch `isTriggerFor` handelt. Dieser Test wird hier allerdings ausgelassen, um Performance beim CHASE-Algorithmus zu sparen. Da für den Schritt in Zeile 6 in Algorithmus 1 bereits Trigger im vornherein durchiteriert werden, muss für Zeile 7 nicht nochmal überprüft werden, ob es sich bei "dem aktiven Trigger um einen Trigger handelt". Genauere Details dazu folgen allerdings in Abschnitt 4.8.

```
1  public class Homomorphism
2  {
3    private HashSet<TermMapping> mappings;
4
5    public boolean generateMappingFor(Term, Term);
6    public boolean generateMappingsFor(RelationalAtom, RelationalAtom);
7    public RelationalAtom applyMappingsTo(RelationalAtom);
8    public RelationalAtom applyMappingsTo(Instance);
9    public Homomorphism composeWith(Homomorphism);
10
11   public boolean isTriggerFor(Instance, IntegrityConstraint);
12   public boolean isTriggerActiveFor(Instance, IntegrityConstraint);
13 }
```

Listing 4.15 Klasse `Homomorphism` mit Feldvariablen und Methoden

Beispiel 4.7. *Dieses Beispielprogramm zeigt die Nutzung von Homomorphismen als eine Menge (`HashSet`) von Termabbildungen. Dazu wird eine Instanz und eine tgd aus vorigen Beispielen verwendet, die anfangs noch einmal ausgegeben werden (Zeilen 7 und 13). Zuerst wird ein Homomorphismus erstellt und dieser mit bestimmten Termabbildungen befüllt (Zeilen 16 - 22).*
Die Anwendungen der Methoden `isTriggerFor` und `is AcitveTriggerFor` auf die Instanz und die tgd geben jeweils `true` als Ergebnis aus (Zeilen 31 und 32). Nachdem das TEILNEHMER-Atom aus der Instanz entfernt wurde (Zeile 34), wodurch der Homomorphismus kein Trigger mehr für die Instanz und die tgd darstellt, gibt eine erneute Anwendung diesmal `false` als Ergebnis von `isTriggerFor` aus (Zeile 36). Dass `isActiveTriggerFor` hier dennoch `true` in Zeile 37 ausgibt, liegt daran, dass die Methode nicht noch einmal mittels `isTriggerFor` überprüft, ob der Homomorphismus ein Trigger ist, um Performance im CHASE-Algorithmus zu sparen.
Schließlich wird die Komposition von Termabbildungen demonstriert, indem zwei weitere Homomorphismen dafür erstellt werden, die mit vorgefertigten Termabbildungen belegt werden (Zeilen 39 - 48). Diese werden anschließend komponiert und das Ergebnis ebenfalls ausgegeben (Zeilen 55 und 57).

```
1  /* ... Instanz-, tgd-Erstellung wie in vorigen Beispielen ... */
2
3  // TEILNEHMER(1, 2),
4  // STUDENTEN(2, Sonnenschein, Sarah, Mathematik),
5  // STUDENTEN(1, Fieber, Fabian, Lehramt Informatik),
6  // NOTEN(1, 1, SS 16, 2.0)
7  System.out.println(instance);
8
9  // STUDENTEN(#V_matrikelnummer_1, #V_nachname_1, #V_vorname_1, #V_studiengang_1)
```

```
10 // TEILNEHMER(#V_modulnummer_1, #V_matrikelnummer_1)
11 // ->
12 // NOTEN(#V_modulnummer_1, #V_matrikelnummer_1, #E_semester_1, #E_note_1)
13 System.out.println(tgd);
14
15 // Definition eines Homomorphismus'
16 Homomorphism homomorphism = new Homomorphism();
17
18 h.addMapping(new TermMapping(new Term("#V_matrikelnummer_1"), new Term(2)));
19 h.addMapping(new TermMapping(new Term("#V_nachname_1"), new Term("Sonnenschein")
       ));
20 h.addMapping(new TermMapping(new Term("#V_vorname_1"), new Term("Sarah")));
21 h.addMapping(new TermMapping(new Term("#V_studiengang_1"), new Term("Mathematik"
       )));
22 h.addMapping(new TermMapping(new Term("#V_modulnummer_1"), new Term(1)));
23
24 // #V_nachname_1 -> Sonnenschein,
25 // #V_studiengang_1 -> Mathematik,
26 // #V_matrikelnummer_1 -> 2,
27 // #V_modulnummer_1 -> 1,
28 // #V_vorname_1 -> Sarah
29 System.out.println(h.getTermMappings());
30
31 System.out.println(h.isTriggerFor(instance, tgd));      // true
32 System.out.println(h.isActiveTriggerFor(instance, tgd));  // true
33
34 instance.getRelationalAtoms().remove(atomTeilnehmer1);
35
36 System.out.println(h.isTriggerFor(instance, tgd));      // false
37 System.out.println(h.isActiveTriggerFor(instance, tgd));  // true
38
39 HashSet<TermMapping> mappings1 = new HashSet<TermMapping>();
40 HashSet<TermMapping> mappings2 = new HashSet<TermMapping>();
41
42 mappings1.add(new TermMapping(new Term("#V_var_1"), new Term("#V_var_2")));
43 mappings1.add(new TermMapping(new Term("#V_var_3"), new Term("#V_var_4")));
44 mappings2.add(new TermMapping(new Term("#V_var_2"), new Term("#V_var_3")));
45
46 // Definition zweier Homomorphismen zur Veranschaulichung der Komposition von
       Termabbildungen
47 Homomorphism h1 = new Homomorphism(mappings1);
48 Homomorphism h2 = new Homomorphism(mappings2);
49
50 // [#V_var_1 -> #V_var_2, #V_var_3 -> #V_var_4]
51 System.out.println(mappings1);
52 // [#V_var_2 -> #V_var_3]
53 System.out.println(mappings2);
54 // {#V_var_2 -> #V_var_3, #V_var_3 -> #V_var_4, #V_var_1 -> #V_var_3}
55 System.out.println(h1.composeWith(h2));
56 // {#V_var_3 -> #V_var_4, #V_var_2 -> #V_var_4, #V_var_1 -> #V_var_4}
57 System.out.println((h1.composeWith(h2)).composeWith(h1));
```

Listing 4.16 Beispielprogramm für `Homomorphism`

4.8. CHASE-Algorithmus

Der CHASE-Algorithmus der Kern des Tools. Für den CHASE und seine verschiedenen Anwendungsfälle, von denen im Rahmen der Arbeit lediglich die Anwendung des CHASE auf Instanzen umgesetzt wurde, gibt es die Klasse `Chase`, unter der die verschiedenen Anwendungsmöglichkeiten als einzelne Methoden zu finden sind. Der Quellcode des CHASE-Algorithmus' ist in Anhang B zu finden. Die hier beschriebenen Schritte werden anhand der Zeilennummern des Quellcodes erklärt. Die Variablennamen entsprechen denen aus Algorithmus 1. Um den Pseudocode des CHASE-Algorithmus' mit der implementierten Version durch häufige Zeilennummernreferenzen zu vergleichen, wird dieser hier nochmal in Algorithmus 2 angezeigt. Um bei den Zeilenangaben nicht zu verwirren, wird bei den Verweisen immer die sich darauf beziehende Referenz mitangegeben. In der Implementierung werden in einer gesonderten Hilfsmethode

Algorithmus 2 : Standard-CHASE

Input : Instanz I, Menge von tgd's und egd's $\mathcal{B}$
Output : Instanz I mit reingeCHASEten Integritätsbedingungen $\mathcal{B}$
oder FAIL bei nichterfüllter egd

```
1  Function chase(I, B)
2      I' := I
3      while I' ≠ {} do
4          N := {}, μ := {}
5          foreach b ∈ B mit Rumpf p1(x) do
6              foreach Trigger h für b in I mit h(p1(x)) ∩ I' ≠ {} do
7                  if h ist aktiver Trigger für b in μ(N ∪ I) then
8                      if b = ∀x p1(x) → ∃y p2(x, y) ist tgd then
9                          h' := h ∪ {y → v} mit v neue Nullwerte
10                         N := N ∪ h'(p2(x, y))
11                     end
12                     else if b = ∀x p1(x) → xi = xj ist egd then
13                         if (h(xi) ≠ h(xj)) ∧ ({h(xi), h(xj)} ⊆ Const) then
14                             FAIL
15                         end
16                         else
17                             ν := {max(h(xi), h(xj)) ↦ min(h(xi), h(xj))}
18                             μ := μ ∘ (ν ∘ μ)
19                         end
20                     end
21                 end
22             end
23         end
24         I' := μ(N ∪ I) − I
25         I := μ(I) ∪ I'
26     end
27     return I
28 end
```

`generateTriggers` alle Homomorphismen generiert, die für eine gerade ausgewählte Integritätsbedingung und die im Parameter übergebene Instanz einen Trigger darstellen. Die Methode `generateTriggers` tut dies mittels Rekursion. Der zugehörige Quellcode ist ebenfalls in Anhang B in zu finden. Durch diese Triggerliste kann nun wie in Zeile 6 von Algorithmus 2 iteriert werden. Um in Zeile 7 in Algorithmus 2 dann Performance einzusparen, wird in der Methode `isActiveTriggerFor` der `Homomorphism`-Klasse nicht noch einmal erst mittels `isTriggerFor` überprüft, ob es sich bei einem Homomorphismus überhaupt um einen Trigger handelt, da dies auch schon bei der Generierung des Homomorphismus' beachtet wurde. Es wird in `isActiveTriggerFor` also nur überprüft, ob der Kopf der Integritätsbedingung erfüllt wird.

Die Eingaben des Algorithmus' sind eine Instanz `I` und eine Menge von Integritätsbedingungen `B`, hier repräsentiert durch ein `Set<IntegrityConstraint>`. Eine `HashMap<String, Integer>` `nullCounter` (Zeilen 6 - 12 in Anhang B) zählt für Attribute die jeweils erstellten Nullwerte durch tgd's. Damit dies funktioniert, müssen die Bezeichnungen der in der Instanz vorhandenen Attribute den entsprechenden `indexNamen` der Nullwerte und Variablen entsprechen. So müssen beispielsweise bei Nutzung des Attributs `vorname` die Variablen ebenfalls `#V_vorname` mit der jeweiligen Indexnummer benannt sein. Es darf hier keine andere Bezeichnung oder Abkürzung des Attributs verwendet werden.
In einer `while`-Schleife wird überprüft, ob die anfangs mit der Eingabe `I` initialisierte temporäre Instanz `I_NewFacts` nicht leer ist, was der Abfrage entspricht, ob neue Fakten hinzugekommen sind (Zeile 19 in Anhang B). Zu Beginn einer solchen `while`-Iteration wird wie in Algorithmus 2 die temporäre Instanz `N` erstellt (Zeile 22 in Anhang B). Außerdem folgt in Zeile 25 in Anhang B die Definition einer zusätzlichen temporären Instanz `I_Union_N`, die die Vereinigung der Instanzen `I` und `N` darstellen soll. Grund für die Nutzung einer solchen temporären Instanz ist der Umstand, dass Mengen, die die Methode `addAll` zur Vereinigung von Mengen nutzen, bei der Anwendung verändert werden. Es wird also kein neues Mengenobjekt erstellt und als Ergebnis zurückgegeben. Um den Ausdruck $N \cup I$ zu repräsentieren, bei dem sowohl I als auch N nach der Operation unverändert bleiben, muss hier also eine dritte, temporäre Instanz verwendet werden, die sowohl alle Atome aus `I` als auch aus `N` enthält.
Eine `foreach`-Schleife in Zeile 30 in Anhang B) iteriert dann über die Integritätsbedingungen `B` gefolgt von der Triggergenerierung durch `generateTriggers` in Zeile 35 in Anhang B. Zusätzlich wird in jeder Iteration die Vereinigung `I_Union_N` der Eingabeinstanz `I` und den neu generierten Atomen durch tgd's `N` aktualisiert (Zeile 33 in Anhang B). Anschließend wird mit einer zweiten `foreach`-Schleife in Zeile 37 in Anhang B über die generierten Trigger iteriert. Dies entspricht der Zeilen 5 und einen Teil von Zeile 6 in Algorithmus 2. Eine vollständige Iteration mit einer Integritätsbedingung bildet einen CHASE-Schritt, welcher in Definition 2.26 erläutert wurde.
Mittels eines booleschen Flags `bIntersectsI_NewFactsFlag` (Zeile 41 in Anhang B) wird getestet, ob ein Atom aus dem Rumpf der aktuellen Integritätsbedingung äquivalent zu einem Atom aus der Instanz mit den neuen Fakten `I_NewFacts` ist. Sobald eine solche Äquivalenz festgestellt wurde, wird das Flag auf `true` gesetzt (Zeilen 61 - 63 in Anhang B) und jede weitere Überprüfung abgebrochen, da nur eine Übereinstimmung notwendig ist (Zeilen 64 und 71 in Anhang B). Das hat den Zweck, dass die Ausführung des CHASE mit dem aktuellen Trigger nur dann fortgesetzt wird, falls mindestens ein Atom des Rumpfes der Integritätsbedingung durch die neuen Fakten der letzten Iteration erfüllt wird. Diese Überprüfung von Zeile 41 bis 73 in Anhang B entspricht dabei der Zeile 6 in Algorithmus 2. Wenn das Flag `false` ist, wird demzufolge der Algorithmus nicht weiter ausgeführt dank der Überprüfung dessen in Zeile 75 in Anhang B.
Als nächstes folgt die Überprüfung, ob der aktuelle Trigger für die Eingabeinstanz und neuen Fakten `I_NewFacts` aktiv ist (Zeile 77 in Anhang B, Zeile 7 in Algorithmus 2). Dann wird die aktuelle Integritätsbedingung `b` mit `instanceof` unterschieden in eine tgd (Zeile 80 in Anhang B) und eine egd (Zeile 116 in Anhang B).
Für die Generierung eines neuen Atoms im Falle einer tgd wird in den Atomen des Kopfes der tgd nach existenzquantifizierten Variablen gesucht (Zeilen 86 - 92 in Anhang B) und diese mithilfe des anfangs eingeführten `nullCounters` durch neu generierte Nullwerte ersetzt (Zeilen 95 - 104 in Anhang B). Dafür wird ein temporärer Homomorphismus als Kopie des aktuellen Triggers in Zeile 82 in Anhang B erstellt, welcher mit der Abbildung der existenzquantifizierten Variablen durch die neuen Nullwerte in Zeile 106 in Anhang B erweitert wird. Schließlich werden diese Abbildungen auf den Kopf der tgd angewandt und die neuen Atome mit den Nullwerten der Variablen `N` zugewiesen (Zeile 111 in Anhang B), die sämtliche solcher generierten Atome während einer Iteration von Zeile 28 in Anhang B speichert. Diese Schritte entsprechen den Zeilen 8 und 9 in Algorithmus 2.
Für die egd's werden die Gleichheitsatome im Kopf auf Gleichheit überprüft, indem die Termabbildungen des aktuellen Triggers auf die einzelnen Terme des jeweiligen Gleichheitsatoms angewandt werden (Zeilen

121 - 128 in Anhang B). Jetzt muss lediglich geprüft werden, ob beide abgebildeten Terme Konstanten sind (Zeile 132 in Anhang B). Eine Überprüfung auf exakte Gleichheit, das heißt, auf die tatsächlichen Werte der Terme, ist hier nicht notwendig, da durch die Überprüfung auf einen aktiven Trigger in Zeile 77 in Anhang B bereits einhergeht, dass die beiden Terme ungleich sein müssen. Bei Ungleichheit wird in Zeile 134 in Anhang B dann eine Textmeldung ausgegeben, die beschreibt, dass der CHASE fehlgeschlagen ist und eine leere Instanz als Ergebnis ausgegeben wird (Zeile 135 in Anhang B). Während die Theorie einfach nur spezifiziert, dass ein "FAIL" oder eine Meldung bei einem fehlgeschlagenen CHASE ausgegeben wird, ist hier wegen dem Rückgabetypen `Instance` der `chase`-Funktion eine Rückgabe eines `Instance`-Objektes notwendig. Als Lösung wurde hier eine neue leere Instanz gewählt, die im Gegensatz zu einem eventuellen `null`-Rückgabewert keine `NullPointerException` verursacht, falls ein übergeordnetes Programm, das diese `chase`-Funktion verwendet, auf die Ergebnisinstanz zugreifen möchte. Es steht an dieser Stelle allerdings nicht fest, ob diese Lösung designtechnisch die beste ist.
Im anderen Fall, dass nicht beide Terme Konstanten sind (Zeile 138), wird eine Abbildung vom "größeren" Term auf den "kleineren" erstellt (Zeile 140 in Anhang B). Diese Art der Hierarchie entspricht der aus Abbildung 2.1. Umgesetzt ist dies mit den zwei Hilfsfunktionen `getSmallerTerm` und `getBiggerTerm`. Diese Abbildung `omega` wird nun mit den Abbildungen aus dem in Zeile 28 in Anhang B erstellten Homomorphismus `mu` komponiert (Zeile 144 in Anhang B). Die Bezeichnungen sind an μ und ω in Algorithmus 2 angelehnt und der Vorgang entspricht den dortigen Zeilen 12 und 13.
Nachdem alle Integritätsbedingungen durchiteriert sind, werden in den Zeilen 154 - 166 in Anhang B die Instanzen aktualisiert, was den Zeilen 14 und 15 in Algorithmus 2 entspricht. Danach folgen eventuell weitere Iterationen der `while`-Schleife in Zeile 19 in Anhang B. Bei erfolgreicher Durchführung der Funktion wird am Ende die Instanz `I` als Ergebnis ausgegeben.

Beispiel 4.8. *Zur Demonstration soll folgendes Beispielprogramm den Ablauf des CHASE-Algorithmus' zeigen. Dazu wurden eine Instanz und eine tgd erstellt, wie sie auch in Beispiel 2.10 verwendet wurden. Die Instanz und die tgd werden nochmal zu Beginn als Text ausgegeben (Zeilen 5 und 9). Danach folgt die Durchführung und Textausgabe des CHASE in Form der `chase`-Funktion (Zeile 15). Um die einzelnen Ausführungsschritte zu erklären, werden im Folgenden ausschließlich Zeilennummernreferenzen aus Anhang B verwendet.*
Die `chase`-Funktion erhält also nach Aufruf die erstellte Instanz (nachfolgend `I`) und eine Menge (`HashSet`) von Integritätsbedingungen (nachfolgend `B`), in der sich lediglich die erstellte tgd befindet. Zuerst wird in der Funktion die `HasMap nullCounter` erstellt, die den Wert des maximalen Indexes der erstellten Nullwerte speichert (Zeile 6). Die initialen Werte sind `{studiengang=0, note=0, vorname=0, name=0, matrikelnummer=0, modulnummer=0, semester=0}`. Als nächstes wird `I` als initiale neue Fakten der temporären Instanz `I_NewFacts` definiert, indem `I` kopiert und `I_NewFacts` zugewiesen wird (Zeile 39). Nun beginnt die `while`-Schleife, die solange wiederholt wird, bis `I_NewFacts` nach einer Iteration leer geworden ist (Zeile 41). Es folgen die Initialisierungen der temporären Instanzen `N` und `I_Union_N` (`I` vereinigt mit neuen Atomen durch tgd's) und des Homomorphismus `mu`, welcher allerdings in diesem Beispielprogramm immer leer sein wird, da keine egd's ausgewertet werden. Jetzt wird die erstellte tgd `b` aus `B` ausgewählt und Trigger für `I` und `b` generiert (Zeile 35). Die Trigger sehen dabei wie folgt aus: `{#V_vorname_1 -> Max, #V_matrikelnummer_1 -> 3, #V_studiengang_1 -> Elektrotechnik, #V_modulnummer_1 -> 2, #V_nachname_1 -> Mueller}, {#V_modulnummer_1 -> 7, #V_vorname_1 -> Max, #V_matrikelnummer_1 -> 3, #V_studiengang_1 -> Elektrotechnik, #V_nachname_1 -> Mueller}`. Davon wird dann einer am Beginn der nächsten `foreach`-Schleife in Zeile 37 ausgewählt (in dem Fall der erstere, durch den die Modulnummer auf "2" abgebildet wird).
Im nächsten Schritt wird mithilfe des booleschen Flags geprüft, ob die neuen Fakten dieser Iteration `I_NewFacts` sich mit mindestens einem Atom aus dem Rumpf von `b` überschneiden, was auch der Fall ist (Zeile 75). Der gewählte Trigger wird anschließend daraufhin geprüft, ob er für `I_Union_N` und `b` ein aktiver Trigger ist (Zeile 77). Da dies der Fall ist, wird `b` nun zwischen tgd und egd unterschieden,

wovon der Fall der tgd eintritt. Jetzt werden die Termabbildungen des aktuellen Triggers auf den Kopf der tgd `b` *angewandt, sodass alle* `#V`*-Werte durch deren abgebildeten Werte ersetzt werden. Anstelle der existenzquantifizierten Variablen* `#E` *sollen nun Nullwerte treten, die neu generiert werden (Zeilen 95 - 104). Dazu werden die Werte aus* `nullCounter` *genutzt, um Nullwerte mit neuen, noch nicht zuvor genutzten Indizes zu erstellen. Die Werte von* `nullCounter` *sind nun dieselben wie vorher bis auf* `{note=1, semester=1}`*. Die Ersetzung der* `#E`*-Werte erfolgt dann mit einer weiteren Termabbildung auf die neuen Nullwerte, die auf den Kopf der tgd angewandt wird und somit zur Ersetzung führt (Zeilen 104 - 106). Die daraus entstandenen Atome werden* `N` *zugewiesen. (Zeile 111).*
Somit wurde der erste Trigger ausgeführt und die `foreach`*-Schleife aus Zeile 37 wählt den nächsten Trigger aus, welcher sich zum ersten lediglich in der Abbildung* `#V_modulnummer_1 -> 7` *unterscheidet. Daher sind die ausgeführten nächsten Schritte dieselben wie in der eben durchgeführten Iteration. Nach dieser zweiten Iteration hat* `nullCounter` *die weiter veränderten Werte* `{note=2, semester=2}` *und* `N` *nun die neuen Atome* `{NOTEN(7, 3, #N_semester_2, #N_note_2), NOTEN(2, 3, #N_semester_1, #N_note_1)}`*.*
Da sowohl keine weiteren Trigger als auch Integritätsbedingungen vorhanden sind, passieren die nächsten Schritte ab Zeile 154. Hier wird `I_Union_N` *aktualisiert, indem die Atome aus* `N` *hinzugefügt werden. Die neuen Fakten bekommen nun die Atome aus* `I_Union_N` *zugeschrieben, wovon alle Atome aus* `I` *entfernt werden, sodass nur noch die Atome aus* `N` *übrig bleiben. Falls egd's ausgewertet wurden, hätten vorher die dadurch gebildeten Termabbildungen Anwendung gefunden, um gegebenenfalls Nullwerte zu verändern. Abschließend wird* `I` *um die neuen Fakten erweitert, sodass zur ursprünglichen Eingabeinstanz die beiden* `NOTEN`*-Atome hinzugekommen sind.*
Da `I_NewFacts` *nicht leer ist, folgt eine weitere Iteration der* `while`*-Schleife in Zeile 19. Die darauffolgenden Schritte und auch die generierten Trigger wären dieselben wie in der ersten Iteration. Diesmal allerdings triggern keine neuen Fakten aus* `I_NewFacts` *die tgd, weshalb das boolesche Flag in Zeile 75 durch den Wert* `false` *eine weitere Ausführung der darauffolgenden Schritte verhindert. Dasselbe passiert direkt danach mit dem zweiten Trigger. Nun sind keine weiteren Trigger und Integritätsbedingungen mehr vorhanden. In den letzten Schritten der* `chase`*-Funktion werden lediglich noch die Instanzen wieder ab Zeile 154 aktualisiert, was allerdings keine Auswirkungen hat, da sich nichts geändert hat. Da keine neuen Fakten mehr hinzugekommen sind, wird keine* `while`*-Iteration mehr ausgeführt und die Funktion terminiert mit der Rückgabe von* `I` *als Ergebnis in Zeile 171.*

```
1  /* ... Erstellung einer Instanz und tgd wie in anderen Beispielen ... */
2  // TEILNEHMER(7, 3),
3  // TEILNEHMER(2, 3),
4  // STUDENTEN(3, Mueller, Max, Elektrotechnik)
5  System.out.println(instance);
6  // STUDENTEN(#V_matrikelnummer_1, #V_nachname_1, #V_vorname_1, #
       V_studiengang_1),
7  // TEILNEHMER(#V_modulnummer_1, #V_matrikelnummer_1)
8  // NOTEN(#V_modulnummer_1, #V_matrikelnummer_1, #E_semester_1, #E_note_1)
9  System.out.println(constraints);
10 // TEILNEHMER(7, 3),
11 // TEILNEHMER(2, 3),
12 // STUDENTEN(3, Mueller, Max, Elektrotechnik),
13 // NOTEN(7, 3, #N_semester_1, #N_note_1),
14 // NOTEN(2, 3, #N_semester_2, #N_note_2)
15 System.out.println(Chase.chase(instance, constraints));
```

Listing 4.17 Beispielprogramm für `Chase`

Insgesamt konnte der CHASE, wie er in [BKM+17] dargestellt wurde, erfolgreich und nahezu eins-zu-eins implementiert werden. Mit der Implementierung können Objekte der Klasse `Instance` als CHASE-Objekt und Integritätsbedingungen in Form von tgd's und egd's in einer Menge als CHASE-Parameter der `chase`-Funktion übergeben werden, wodurch die Integritätsbedingungen in die Instanz eingearbeitet werden, sodass sie von der Instanz am Ende der Ausführung erfüllt werden. Weitere Varianten des CHASE wie der Oblivious CHASE oder Core-CHASE deckt die Implementierung jedoch nicht ab. Auch die Nutzung anderer CHASE-Objekte und -Parameter wird nicht vom Tool unterstützt. Eine genauere Auflistung der umgesetzten und nicht umgesetzten Konzepte und Features und ein Ausblick auf die Möglichkeiten zum Ausbau des Tools schließen diese Masterarbeit im nächsten Kapitel ab.

5. Fazit und Ausblick

Der in [BKM+17] aufgeführte CHASE-Algorithmus konnte in ein Tool implementiert werden. Viele grundlegende theoretische Konzepte zum CHASE wurden verwirklicht, womit eine Basis geschaffen wurde, die für eine weitere Entwicklung und Umsetzung von noch nicht abgedeckten Anwendungsszenarien des CHASE verwendet werden kann. Aufgrund der Vielschichtigkeit und des benötigten umfangreichen theoretischen Verständnisses des CHASE, das sich nicht in der begrenzten Arbeitszeit der Masterarbeit rechtzeitig aufbereiten ließ, ist es letztendlich nicht gelungen, das anfangs beschriebene Ziel der Umsetzung des CHASE-Algorithmus' in bestimmten Anwendungsbereichen zu erreichen. Es wurde ein Tool geschaffen, welches nur den Standard-CHASE umsetzt, welcher in [BKM+17] ausführlich beschrieben wurde. Das Gerüst hingegen, welches das prototypisch umgesetzte CHASE-Tool liefert, kann zum weiteren Ausbau der Funktionen verwendet werden, mit denen der CHASE allmählich universal einsetzbar gemacht werden kann.
Von der in Abschnitt 1.2 geschilderten Aufgabenstellung konnte mit dem Standard-CHASE aus [BKM+17] lediglich Anwendung III. zum Teil in Form des Datenaustausches umgesetzt werden. Aufgrund des Standard-CHASE und seiner Art und Weise, Instanzen so zu verändern, dass sie gegebene Integritätsbedingungen erfüllen, wurde mit seiner Implementierung auch zum Teil IV. mitumgesetzt, was allerdings nicht Teil der Aufgabenstellung war. Die Implementierung wurde in Abschnitt 4.8 ausführlich beschrieben und ist in Anhang B zu finden. Als Konzepte zur Umsetzung des CHASE wurden verwirklicht:

- Terme (siehe Definition 2.17 und Abschnitt 4.1)
- Variablen für Integritätsbedingungen und Anfragen (siehe Bemerkungen zu Variablen und Nullwerten unter Definition 2.23 und Abschnitt 4.2)
- Nullwerte für Instanzen (siehe wie bei Variablen)
- relationale Atome und Gleichheitsatome (siehe Definition 2.19 und Abschnitt 4.3)
- Instanzen (siehe Definition 2.21 und Abschnitt 4.4)
- Integritätsbedingungen (siehe Definitionen 2.22, 2.23 und Abschnitt 4.5)
- Homomorphismen mittels Termabbildungen, mit denen auch die Termhierarchie zur Ersetzung von bestimmten Termen auf andere durch die `apply`-Methoden umgesetzt wurde (siehe Definitionen 2.24 und 2.25, Abbildung 2.1, Abschnitt 4.6 und Abschnitt 4.7)

Anfragen wurden lediglich als Klasse `Query` definiert, die von der Klasse `Tgd` erbt, was auch in Abbildung 4.1 zu sehen ist. Allerdings lag der Fokus bei der Umsetzung des CHASE mit Instanzen, weshalb es der Klasse an Funktionalitäten fehlt und deshalb ein eigener Abschnitt in Kapitel 4 nicht sinnvoll gewesen wäre. Nicht umgesetzt werden konnten aufgrund von zeitlichen Restriktionen:

- Anfragen und damit verbundene CHASE-Anwendungsfälle (I. und II. in Tabelle 1.1)
- Aufbereitung der Konzepte des Provenance-Managements und somit deren Umsetzung für VI.

- Kopplung der Positionen von Termen in relationalen Atomen von Instanzen mit zugehörigen Attributen, wodurch das in der `Instance`-Klasse definierte `schema` (siehe Listing 4.9) nicht bestimmt, wie die Terme in relationalen Atomen in einer Instanz geordnet und an Attribute gebunden sind
  - dadurch auch keine Umsetzung von s-t tgd's und damit verbundenen CHASE-Anwendungsfällen (hier insbesondere VI. und der Datenintegrationsanteil von III.)
- mittels Schnittstellen Anbindung an ein Kern-SQL und relationalen Tabellen bei der Ausführung des CHASE
- das BACKCHASE-Verfahren

Das Ziel, verschiedene CHASE-Anwendungen in einem Tool zu vereinen, indem die CHASE-Objekte und CHASE-Parameter der Anwendungen flexibilisiert werden, wurde somit nicht erreicht. Für eine Umsetzung dieses Ziels müssen die genannten offen stehenden Anliegen noch konzeptionell für die konkrete Umsetzung erarbeitet und implementiert werden. Weitere Anliegen, die nicht für die Zielumsetzung notwendig sind, jedoch die Qualität des Tools steigern könnten, wären beispielsweise:

- dynamische Eingabe von Daten für die Ausführung des Tools; bisher nur hart-codierte Anwendungsdaten wie das Studentenbeispiel aus Abschnitt 1.3 und Neukompilierung des Tools bei sich ändernder "Eingabe"
  - eventuell Eingabe auch durch externe Dateien beziehen und diese im Tool parsen
- Ausgabe von Instanzen, Integritätsbedingungen und Homomorphismen sortieren, da diese aufgrund von verwendeten `HashSet`s mittels Hashwerten unübersichtlich sortiert sind
- Einbeziehung weiterer Optimierungstechniken wie beispielsweise Provenance-Informationen (siehe Unterabschnitt 3.1.4 und [DH13]) und Normalisierung von tgd's ([BKM+17])
- gegebenenfalls ein anderer Ansatz bezüglich der Umsetzung der `Term`-Klasse, sodass die in Abbildung 2.1 abgebildete Termhierarchie stärker durch die `Term`-Klasse an sich zur Geltung kommt und so auch keine Enumerationen `TermType` und `ConstType` mehr verwendet werden müssen
  - bei Änderung müssten dann auch an anderen Stellen des Tools Änderungen vorgenommen werden, wo die Terme von der aktuellen Strukturumsetzung Gebrauch machen
  - dasselbe gegebenenfalls auch bei Variablen bezüglich `VariableType` zur Unterscheidung von gegebenen und existenzquantifizierten (bzw. ausgezeichneten und nichtausgezeichneten) Variablen
- Umsetzung des Naive Oblivious CHASE (etwa durch ein zusätzliches Flag im implementierten CHASE-Algorithmus, der angibt, ob die Prüfung auf aktive Trigger durchgeführt werden soll), Skolem-Oblivious CHASE und Core-CHASE

Literaturverzeichnis

[ABU79] Aho, Alfred V. ; Beeri, Catriel ; Ullman, Jeffrey D.: The Theory of Joins in Relational Databases. In: *ACM Trans. Database Syst.* 4 (1979), September, Nr. 3, 297–314. http://dx.doi.org/10.1145/320083.320091. – DOI 10.1145/320083.320091. – ISSN 0362–5915

[Aug17] Auge, Tanja: *Umsetzung von Provenance-Anfragen in Big-Data-Analytics-Umgebungen.* Masterarbeit, Universität Rostock, 2017

[BIL17] Bonifati, Angela ; Ileana, Ioana ; Linardi, Michele: ChaseFUN: a Data Exchange Engine for Functional Dependencies at Scale. In: *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.*, 2017, 534–537

[BKM+17] Benedikt, Michael ; Konstantinidis, George ; Mecca, Giansalvatore ; Motik, Boris ; Papotti, Paolo ; Santoro, Donatello ; Tsamoura, Efthymia: Benchmarking the Chase. In: *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems.* New York, NY, USA : ACM, 2017 (PODS '17). – ISBN 978–1–4503–4198–1, 37–52

[BLT14] Benedikt, Michael ; Leblay, Julien ; Tsamoura, Efthymia: PDQ: Proof-driven Query Answering over Web-based Data. In: *Proc. VLDB Endow.* 7 (2014), August, Nr. 13, 1553–1556. http://dx.doi.org/10.14778/2733004.2733028. – DOI 10.14778/2733004.2733028. – ISSN 2150–8097

[DH13] In: Deutsch, Alin ; Hull, Richard: *Provenance-Directed Chase&Backchase.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2013. – ISBN 978–3–642–41660–6, 227–236

[DNR08] Deutsch, Alin ; Nash, Alan ; Remmel, Jeff: The Chase Revisited. In: *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems.* New York, NY, USA : ACM, 2008 (PODS '08). – ISBN 978–1–60558–152–1, 149–158

[DPT99] Deutsch, Alin ; Popa, Lucian ; Tannen, Val: Physical Data Independence, Constraints, and Optimization with Universal Plans. In: *Proceedings of the 25th International Conference on Very Large Data Bases.* San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1999 (VLDB '99). – ISBN 1–55860–615–7, 459–470

[DPT06] Deutsch, Alin ; Popa, Lucian ; Tannen, Val: Query Reformulation with Constraints. In: *SIGMOD Rec.* 35 (2006), März, Nr. 1, 65–73. http://dx.doi.org/10.1145/1121995.1122010. – DOI 10.1145/1121995.1122010. – ISSN 0163–5808

[FKMP05] Fagin, Ronald ; Kolaitis, Phokion G. ; Miller, Renée J. ; Popa, Lucian: Data exchange: semantics and query answering. In: *Theoretical Computer Science* 336 (2005), Nr. 1, 89 - 124. http://dx.doi.org/https://doi.org/10.1016/j.tcs.2004.10.033. – DOI https://doi.org/10.1016/j.tcs.2004.10.033. – ISSN 0304–3975. – Database Theory

[FKPT11] In: Fagin, Ronald ; Kolaitis, Phokion G. ; Popa, Lucian ; Tan, Wang-Chiew: *Schema Mapping Evolution Through Composition and Inversion*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2011. – ISBN 978–3–642–16518–4, 191–222

[GMPS14] Geerts, Floris ; Mecca, Giansalvatore ; Papotti, Paolo ; Santoro, Donatello: That's All Folks!: Llunatic Goes Open Source. In: *Proc. VLDB Endow.* 7 (2014), August, Nr. 13, 1565–1568. http://dx.doi.org/10.14778/2733004.2733031. – DOI 10.14778/2733004.2733031. – ISSN 2150–8097

[GMS12] Greco, Sergio ; Molinaro, Cristian ; Spezzano, Francesca: *Incomplete Data and Data Dependencies in Relational Databases*. Morgan & Claypool Publishers, 2012. – ISBN 1608459268, 9781608459261

[Heu17] Heuer, Andreas: *"Theorie Relationaler Datenbanken"*. Vorlesung Sommersemester, Universität Rostock, 2017

[ICDK14] Ileana, Ioana ; Cautis, Bogdan ; Deutsch, Alin ; Katsis, Yannis: Complete Yet Practical Search for Minimal Query Reformulations Under Constraints. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2014 (SIGMOD '14). – ISBN 978–1–4503–2376–5, 1015–1026

[MMS79] Maier, David ; Mendelzon, Alberto O. ; Sagiv, Yehoshua: Testing Implications of Data Dependencies. In: *ACM Trans. Database Syst.* 4 (1979), Dezember, Nr. 4, 455–469. http://dx.doi.org/10.1145/320107.320115. – DOI 10.1145/320107.320115. – ISSN 0362–5915

Anfragenverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Anhang A. Beispieldatensatz

$\mathbf{ID_{Studenten}}$	**Matrikelnummer**	**Name**	**Vorname**	**Studiengang**
S_1	1	Fieber	Fabian	Lehramt Informatik
S_2	2	Sonnenschein	Sarah	Mathematik
S_3	3	Müller	Max	Elektrotechnik
S_4	4	Müller	Mira	Informatik
S_5	5	Johansen	Johannes	Informatik
S_6	6	Miller	Mia	Informatik
S_7	7	Mustermann	Max	Elektrotechnik
S_8	8	Joahnnes	Paul	ITTI

Tabelle A.1. Beispielrelation von STUDENTEN

$\mathbf{ID_{Noten}}$	**Modulnummer**	**Matrikelnummer**	**Semester**	**Note**
N_1	001	1	SS 16	2.0
N_2	001	2	SS 16	1.7
N_3	001	4	SS 16	1.7
N_4	001	5	SS 16	3.0
N_5	002	1	WS 15/16	3.7
N_6	002	2	WS 14/15	1.3
N_7	002	3	WS 14/15	2.3
N_8	002	4	WS 15/16	1.0
N_9	002	5	WS 15/16	1.3
N_{10}	002	6	WS 15/16	2.0
N_{11}	002	7	WS 15/16	3.3
N_{12}	003	1	WS 16/17	1.0
N_{13}	004	3	WS 16/17	1.3
N_{14}	004	2	WS 16/17	3.0
N_{15}	005	4	SS 17	2.7
N_{16}	005	7	SS 17	1.7
N_{17}	006	4	SS 17	2.7
N_{18}	006	5	SS 17	4.0
N_{19}	007	1	SS 16	2.3
N_{20}	007	3	SS 16	1.7
N_{21}	009	1	SS 16	3.0
N_{22}	009	5	SS 15	5.0
N_{23}	009	5	SS 16	2.7

Tabelle A.2. Beispielrelation von NOTEN

$\mathbf{ID_{Module}}$	**Modulnummer**	**Titel**	**Vertiefung**
M_1	001	DBIII	Informationssysteme
M_2	002	Mathematik 1	xxx
M_3	003	MeKa	Smart Computing
M_4	004	Individuellesa Wissensmanagement	Informationssysteme
M_5	005	DWBI	Wirtschaftsinformatik
M_6	006	Kognitive Systeme	Smart Computing
M_7	007	TRDB	Informationssysteme
M_8	008	Systembiologie	Modelle und Algorithmen
M_9	009	NEidI	xxx

Tabelle A.3. Beispielrelation von MODULE

$\mathbf{ID_{Dozenten}}$	**Modulnummer**	**Dozent**
$D_{1.1}$	001	Professor A
$D_{1.2}$	001	Dozent A
D_2	002	Professor B
D_3	003	Professor C
D_4	004	Professor D
D_5	005	Dozent B
D_6	006	Professor D
D_7	007	Professor A
D_8	008	Professor E
D_9	009	Professor A

Tabelle A.4. Beispielrelation von DOZENTEN

$\mathbf{ID_{Teilnehmer}}$	**Modulnummer**	**Matrikelnummer**
T_1	001	1
T_2	001	2
T_3	001	4
T_4	001	5
T_5	002	1
T_6	002	2
T_7	002	3
T_8	002	4
T_9	002	5
T_{10}	002	6
T_{11}	002	7
T_{12}	003	1
T_{13}	004	3
T_{14}	004	5
T_{15}	004	7
T_{16}	005	4
T_{17}	005	7
T_{18}	006	4
T_{19}	006	5
T_{20}	007	1
T_{21}	007	3
T_{22}	007	5
T_{23}	008	3
T_{24}	009	1
T_{25}	009	4
T_{26}	009	5

Tabelle A.5. Beispielrelation von TEILNEHMER

Anhang B. Quellcode

```
1  public static Instance chase(Instance I, Set<IntegrityConstraint> B)
2  {
3    /* START OF pre setup */
4    // create a HashMap for counting the indexes of new generated Null values
5    // it has the attributes of instance I as key and the counter starting at
         0 as value
6    HashMap<String, Integer> nullCounter = new HashMap<>();
7
8    // populate the HashMap with 0 for every attribute of instance I
9    for (String attribute : I.getAttributes())
10   {
11     nullCounter.put(attribute, 0);
12   }
13   /* END OF pre setup*/
14
15   /* temporary instances and mappings creation */
16   // will contain all new facts after one iteration of the while loop
17   Instance I_NewFacts = new Instance(I);
18
19   while (!I_NewFacts.getRelationalAtoms().isEmpty())
20   {
21     // will contain all relational atoms generated by tgd's
22     Instance N = new Instance(I.getAttributes(), I.getSchema());
23
24     // the union of I and N
25     Instance I_Union_N = new Instance(I);
26
27     // will contain all mappings generated by egd's
28     Homomorphism mu = new Homomorphism();
29
30     for (IntegrityConstraint b : B)
31     {
32       // in every iteration of b update the union set of I and N for new
           generated atoms in N by tgd's
33       I_Union_N.getRelationalAtoms().addAll(N.getRelationalAtoms());
34
35       HashSet<Homomorphism> triggers = generateTriggers(I, b);
36
37       for (Homomorphism h : triggers)
38       {
```

```
39          // flag for the intersection of b.getBody() and I_NewFacts not
                being empty
40          // true, if at least one of the RelationalAtoms of b.getBody() has
                a homomorphism to I_NewFacts with trigger h
41          boolean bIntersectsI_NewFactsFlag = false;
42
43          // for every constraintAtom in b.getBody() ...
44          for (RelationalAtom bBodyAtom : b.getBody())
45          {
46            // ... apply the mappings of h to constraintAtom ...
47            // temporary atom to not alter constraintAtom
48            RelationalAtom mappedBodyAtom = new RelationalAtom(bBodyAtom);
49
50            // apply all mappings of h to the temporary atom
51            for (TermMapping mapping : h.getTermMappings())
52            {
53              mappedBodyAtom = mapping.apply(mappedBodyAtom);
54            }
55
56            // ... and for every instanceAtom in I_NewFacts ...
57            for (RelationalAtom instanceAtom : I_NewFacts.getRelationalAtoms
                  ())
58            {
59              // ... test if the mapped constraintAtom equals the instance
                    atom
60              // flag becomes true and we can break out of the loop because
                    we need only one satisfied instance atom in total
61              if (mappedBodyAtom.equals(instanceAtom))
62              {
63                bIntersectsI_NewFactsFlag = true;
64                break;
65              }
66            }
67
68            // break out of the outer loop because we don't need further
                  iterations if the flag is true
69            if (bIntersectsI_NewFactsFlag)
70            {
71              break;
72            }
73          }
74
75          if (bIntersectsI_NewFactsFlag)
76          {
77            if (h.isActiveTriggerFor(mu.applyMappingsTo(I_Union_N), b))
78            {
79              // b is a tgd
80              if (b instanceof Tgd)
81              {
```

```
82                Homomorphism h_temp = new Homomorphism(h);
83
84                for (RelationalAtom bHeadAtom : (HashSet<RelationalAtom>) b.
                      getHead())
85                {
86                  for (Term term : bHeadAtom.getTerms())
87                  {
88                    if (term.getTermType() == TermType.Variable)
89                    {
90                      Variable oldVariable = (Variable) term.getTermValue();
91
92                      if (oldVariable.getVariableType() == VariableType.E)
93                      {
94                        // replace in nullCounter the value for an attribute
                              by its by 1 incemented value (counter goes up for
                              that attribute)
95                        nullCounter.replace(oldVariable.getIndexName(),
                              nullCounter.get(oldVariable.getIndexName()) + 1);
96
97                        // new Null created with the indexName of the
                              Variable and the new counter number for the index
98                        Null newNull = new Null(oldVariable.getIndexName(),
                              nullCounter.get(oldVariable.getIndexName()));
99
100                       // wrapping oldVariable and newNull in Term's for the
                              mapping
101                       Term sourceTerm = new Term(oldVariable);
102                       Term targetTerm = new Term(newNull);
103
104                       TermMapping newMapping = new TermMapping(sourceTerm,
                              targetTerm);
105
106                       h_temp.addMapping(newMapping);
107                     }
108                   }
109                 }
110
111                 N.addRelationalAtom(h_temp.applyMappingsTo(bHeadAtom));
112               }
113             }
114
115             // b is an egd
116             else
117             {
118               for (EqualityAtom bHeadAtom : (HashSet<EqualityAtom>) b.
                      getHead())
119               {
120                 // temporary terms that are the result of applying the
                        mappings of h to the equality atoms of the egd
```

```
121                 Term eqTerm1 = bHeadAtom.getTerm1();
122                 Term eqTerm2 = bHeadAtom.getTerm2();
123
124                 for (TermMapping mapping : h.getTermMappings())
125                 {
126                   eqTerm1 = mapping.apply(eqTerm1);
127                   eqTerm2 = mapping.apply(eqTerm2);
128                 }
129
130                 // if both terms of the equality atom are Constants the
                        CHASE fails
131                 // testing for unequality isn't necessary here because of
                        the test 'h.isActiveFor(I_Union_N)' before (the terms
                        are considered as unequal already)
132                 if (eqTerm1.getTermType() == TermType.Const && eqTerm2.
                        getTermType() == TermType.Const)
133                 {
134                   System.out.println("The CHASE failed because of a
                          violated egd. An empty instance will be returned. The
                          two terms in question are " + eqTerm1.toString() + "
                          and " + eqTerm2.toString() + ".");
135                   return new Instance();
136                 }
137
138                 else
139                 {
140                   TermMapping maxToMinMapping = new TermMapping(
                          getBiggerTerm(eqTerm1, eqTerm2), getSmallerTerm(
                          eqTerm1, eqTerm2));
141                   Homomorphism omega = new Homomorphism();
142                   omega.addMapping(maxToMinMapping);
143
144                   mu = mu.composeWith(omega).composeWith(mu);
145                 }
146               }
147             } // end of if b instanceof else
148           } // end of if h acitve trigger
149         } // end of if bIntersectsI_NewFactsFlag
150       } // end of foreach homomorphism h
151     } // end of foreach integrity constriant b
152
153     // update the union of I and N
154     I_Union_N.getRelationalAtoms().addAll(N.getRelationalAtoms());
155
156     // apply the mappings of the egd's to I_Union_N
157     I_NewFacts = mu.applyMappingsTo(I_Union_N);
158
159     // get rid of the unchanged, old facts from I
160     I_NewFacts.getRelationalAtoms().removeAll(I.getRelationalAtoms());
```

```
161
162     // apply the mappings of the egd's to I
163     I = mu.applyMappingsTo(I);
164
165     // update I with the new facts
166     I.getRelationalAtoms().addAll(I_NewFacts.getRelationalAtoms());
167
168     // repeat until no more new facts are generated
169   } // end of while I_NewFacts != empty
170
171   return I;
172 }
```

Listing B.1 Implementierter CHASE-Algorithmus

```
1  /**
2   * Generates all homomorphisms that are triggers for the given instance and
        integrity constraint.
3   *
4   * @param instance the instance to that the integrity constraints are
        mapped to by the triggers
5   * @param b the integrity constraint that are mapped to the instance by the
        triggers
6   * @return all possible triggers for {@code instance} and {@code b}
7   */
8  private static HashSet<Homomorphism> generateTriggers(Instance instance,
       IntegrityConstraint b)
9  {
10   // the return object
11   HashSet<Homomorphism> triggers = new HashSet<Homomorphism>();
12
13   HashSet<String> constraintSchemas = new HashSet<String>();
14
15   // get every different relation name from the constraint body
16   for (RelationalAtom constraintAtom : b.getBody())
17   {
18     constraintSchemas.add(constraintAtom.getName());
19   }
20
21   // get the instance atoms filtered by the schemas appearing in the
        constraint body
22   HashSet<RelationalAtom> instanceAtoms = instance.
        getRelationalAtomsBySchema(constraintSchemas);
23
24   // recursive trigger generation
25   generateTriggersRec(instanceAtoms, b.getBody(), new Homomorphism(),
        triggers);
26
27   return triggers;
```

```
28 }
29
30 /**
31  * This is the recursive function part of {@link #generateTriggers(Instance
        , IntegrityConstraint) generateTriggers}. Every
32  * recursive call removes one atom from the constraint body and generates
        mappings for that chosen atom to the object atoms.
33  * For every object atom it is tested if a generation of mappings from the
        chosen constraint atom is possible via
34  * {@link instance.Homomorphism#applyMappingsTo(RelationalAtom)} and {@link
        instance.RelationalAtom#hasHomomorphismTo(RelationalAtom)}.
35  * If that's the case, the current homomorphism gets copied for the next
        recursion step and the mappings are generated and
36  * added to that copy. The next recursion call will have the same object
        atoms, the current constraint atom removed from the
37  * constraint body, the copied homomorphism instance and the trigger list,
        that gets changed by adding the homomorphism at
38  * the beginning of a recursive call when all constraint atoms are removed,
        i. e. mappings have been generated from every
39  * constraint atom to one corresponding set of object atoms.
40  *
41  * @param objectAtoms the target relational atoms for that triggers have to
        be generated with given {@code constraintAtoms}
42  * @param constraintAtoms the source relational atoms for that triggers
        have to be generated with given {@code objectAtoms}
43  * @param h the current trigger to be
44  * @param triggers the trigger set that will contain all the generated
        triggers
45  */
46 private static void generateTriggersRec(HashSet<RelationalAtom> objectAtoms
    , HashSet<RelationalAtom> constraintAtoms, Homomorphism h, HashSet<
    Homomorphism> triggers)
47 {
48   // recursion termination
49   // all constraint atoms have been removed from the constraint body
50   if (constraintAtoms.isEmpty())
51   {
52     triggers.add(h);
53
54     return;
55   }
56
57   // get one atom of the constraint body
58   RelationalAtom constraintAtom = constraintAtoms.iterator().next();
59
60   // iterate over all instance atoms
61   for (RelationalAtom objectAtom : objectAtoms)
62   {
63     // for the current constraint atom try to find a homomorphism to the
```

```
        current object atom while appliying the mappings generated so far
        for the current trigger
64      if (h.applyMappingsTo(constraintAtom).hasHomomorphismTo(objectAtom))
65      {
66        // create a copy of the current trigger so that it doesn't get
              changed when the next constraint atom is chosen for mappings
              generation
67        Homomorphism hNew = new Homomorphism(h);
68
69        hNew.generateMappingsFor(constraintAtom, objectAtom);
70
71        // remove the current constraint atom from the constraint body
72        HashSet<RelationalAtom> bBodyTail = new HashSet<RelationalAtom>(
              constraintAtoms);
73        bBodyTail.remove(constraintAtom);
74
75        // recursive call with the same objectAtoms, constraint body with the
               current atom removed, the copied trigger and the trigger list
              from the beginning
76        generateTriggersRec(objectAtoms, bBodyTail, hNew, triggers);
77      }
78    }
79  }
```

Listing B.2 Hilfsfunktionen zur Generierung von Triggern

Anhang C. Aufbau des Datenträgers

Der beigelegte Datenträger enthält das entwickelte CHASE-Tool, die Masterarbeit selbst und die im Literaturverzeichnis angegebene Literatur. Die Verzeichnisse im Wurzelverzeichnis des Datenträgers sind dabei gleich benannt wie die nachfolgende Auflistung zur Beschreibung des Datenträgerinhalts.

Literaturquellen

Hier befinden sich sämtliche im Literaturverzeichnis angegebenen Literaturquellen. Alle Dokumente liegen dabei im PDF-Format vor und sind gleich benannt wie das entsprechende im Literaturverzeichnis angegebene Kürzel.
Hinweis: Die beigefügte Literatur dient der Nachvollziehbarkeit von Aussagen und darf nicht öffentlich bereitgestellt werden. Dies gilt speziell für Werke, welche durch Lizenzverträge der Universität Rostock durch die Universitätsbibliothek zur Verfügung gestellt wurden.

Masterarbeit

Enthält die TeX-Dateien, in denen die Arbeit geschrieben und von denen diese kompiliert wurde, die BibTeX-Datei, in denen die Literaturquellen angegeben sind, und die dabei genutzten Abbildungen, welche sich allesamt im Verzeichnis “Bilder” befinden. Ebenfalls enthalten ist eine PDF-Version der Arbeit.

Software

- Selbstentwickeltes CHASE-Tool im Verzeichnis “ChaseTool”
- ProvCB von I. Ileana, B. Cautis, A. Deutsch und Y. Katsis

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Masterarbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Rostock, den 10. September 2018 _____