Universität Rostock Traditio et Innovatio

Bachelorarbeit

Vereinheitlichung des CHASE auf Instanzen und Anfragen am Beispiel ChaTEAU

eingereicht von: Jakob Zimmer

eingereicht am: 02.03.2019

Gutachter: Prof. Dr. rer. nat. habil. Andreas Heuer
Dr.-Ing. Holger Meyer

Zusammenfassung

Der CHASE ist ein universeller Datenbankalgorithmus, welcher verschiedene Parameter wie z. B. Abhängigkeiten in ein Objekt einarbeiten kann, sodass als Ergebnis eine Kombination der beiden Komponenten entsteht. Im CHASE-Tool ChaTEAU wurde der CHASE-Algorithmus einzeln für das Einarbeiten von Integritätsbedingungen in zum einen Instanzen und zum anderen Anfragen als CHASE-Objekt implementiert. Diese unterschiedlichen Methoden für die verschiedenen Objekte spiegeln die Universalität des CHASEs nicht korrekt wider. Das Ziel der vorliegenden Bachelorarbeit ist es, die CHASE-Algorithmen auf Instanzen und Anfragen zu vereinheitlichen. Zu diesem Zweck wurde ein Konzept entwickelt, in welchem beide CHASE-Objekte als eine entartete Instanz dargestellt werden, in welche dann eine einzige angepasste CHASE-Methode Integritätsbedingungen einarbeitet. Dieses Konzept wurde dann in ChaTEAU implementiert, wobei außerdem noch weitere Anpassungen vorgenommen wurden, die die Funktionalität des Tools und die Korrektheit des Ergebnisses verbesserten. Ein umfassender Überblick über die zugrundeliegende Theorie des CHASE war für die korrekte Vereinheitlichung entscheidend, sodass diese Arbeit sowohl für diejenigen, die sich für den CHASE interessieren, als auch für diejenigen, die sich für ChaTEAU interessieren, relevant ist.

Abstract

The CHASE is a universal database algorithm, which can incorporate various parameters such as dependencies into an object, so that a combination of the two components is the result. In the CHASE tool ChaTEAU, the CHASE algorithm was implemented individually for incorporating integrity conditions in instances and queries as CHASE objects. These different methods for the different objects do not correctly reflect the universality of the CHASE. The goal of the present bachelor thesis is to unify the CHASE algorithms on instances and queries. For this purpose, a concept was developed in which both CHASE objects are represented as one degenerate instance, into which a single adapted CHASE method then incorporates integrity conditions. This concept was then implemented in ChaTEAU, with additional adaptations that improved the functionality of the tool and the correctness of the result. A comprehensive overview of the underlying theory of the CHASE was crucial for the correct unification, so that this work is appealing to both those interested in the CHASE and those interested in ChaTEAU.

Inhaltsverzeichnis

1. Einleitung

In diesem schnelllebigen Heute ist die Optimierung von täglich kurz oder lang andauernden Abläufen und Arbeitsschritten wahres Geld wert. In der Zeit, in der man warten müsste bis ein langsamer Algorithmus terminiert, eine zu komplex formulierte Anfrage ein Ergebnis liefert oder ein unnötig großer Datensatz geschrieben wird, kann man mit einer Optimierung die Produktivität steigern, Ressourcen sparen und auch Frust vermeiden. Daher stellt die Optimierung in der Informatik ein eigenes Themengebiet dar, in dem hochgradig geforscht wird. In der Datenbankforschung hat man das Glück, dass in [ABU79] und [MMS79] bereits 1979 ein mächtiger Universalalgorithmus beschrieben wurde, der Datenbankentwürfe, Anfragen auf Datenbanken, aber auch die gespeicherten Daten an sich optimieren kann – der *CHASE-Algorithmus*. Der *CHASE* (verkürzt genannt) kann sogar noch für einiges mehr genutzt werden. Allgemein formt er aus einer Menge von Abhängigkeiten und einem (in Abschnitt 1.1 genauer definierten) Objekt als Eingabe, ein Ergebnis, welches der Kombination beider Komponenten entspricht. So werden Abhängigkeiten in das Objekt eingearbeitet oder eben *eingechaset* (wie wir ab jetzt dazu sagen werden). Die "Universalität" des CHASEs rührt hier eben daher, dass man ihm eine Vielzahl an verschiedenen Objekten und zugehörige Abhängigkeiten als Eingabe übergeben kann.

Stellen wir uns beispielsweise als Objekt eine Datenbank mit den drei Relationen: STUDENTEN, TEILNEHMER und NOTEN vor (wie sie im Abschnitt 1.2 beschrieben sind). Eine Bedingung (Abhängigkeit) könnte dann lauten, dass alle Studenten aus STUDENTEN, die eine Note in NOTEN haben, auch als Teilnehmer des entsprechenden Moduls in TEILNEHMER auftauchen müssen. Mit dem CHASE kann man diese Bedingung in die *Instanz* der Datenbank einarbeiten. Man würde also wiederum eine Instanz erhalten, in der die TEILNEHMER-Relation um alle Studenten und entsprechende Module ergänzt worden ist, die z. B. noch nicht eingetragen wurden. Der CHASE könnte aber auch dafür genutzt werden als übergebenes Objekt die *Anfrage*

```
SELECT Matrikelnr, Modulnr
FROM STUDENTEN NATURAL JOIN NOTEN
```

zu optimieren. Durch unsere Bedingung wissen wir, dass alle Studenten und Module auch in der TEILNEHMER-Relation enthalten sein müssen. Mit diesem Wissen kann der CHASE die Anfrage zu

```
SELECT Matrikelnr, Modulnr
FROM TEILNEHMER
```

umformen und hat sie somit um einen überflüssigen `NATURAL JOIN` erleichtert. Zusammengefasst haben wir gerade mit ein und demselben Algorithmus verschiedene Objekte optimiert[1].

In dieser Arbeit wird genau für diese verschiedenen Objekte eine allgemeine Darstellung vorgestellt. Daraus ergibt sich auch das Gesamtziel. In dem von Martin Jurklies [Jur18] entwickelten und später von Fabian Renn [Ren19] erweiterten Tool ChaTEAU, in welchem der CHASE sehr theorienah auf Instanzen und Anfragen implementiert ist, fehlt noch der universale Charakter. Es werden bisher die verschiedenen Objekte in unterschiedlichen Methoden verarbeitet. Abhängigkeiten werden entweder mit `chase(Instance i)` in Instanzen oder mit `chase(Query q)` in Anfragen eingechaset.

[1] Der CHASE allein optimiert nicht, er bereitet die Optimierung nur dadurch vor, dass er alle notwendigen Informationen in das zu optimierende Objekt (hier die Anfrage) einbaut. Die Optimierung erfolgt in der BACKCHASE-Phase.

Im Folgenden wird ein Konzept (siehe Kapitel 4) vorgestellt, bei dem der Ansatz verfolgt wird, eine Methode `chase(CHASEobject o)` zu entwickeln, mit der Abhängigkeiten in ein beliebiges Objekt (erstmal Instanzen oder Anfragen) eingechaset werden können. Die Unterscheidung der Objekte wird nur an den unbedingt notwendigen Stellen in der Anwendung der Bedingungen innerhalb der sogenannten CHASE-Schritte (siehe Definitionen 2.12, 2.14) erfolgen.

1.1. Einführung CHASE

Wie einleitend erwähnt, kann dem CHASE eine Vielzahl an verschiedenen Parametern übergeben werden. Über diese und die zugehörigen Anwendungsfälle wird im folgenden Abschnitt eine kurze Übersicht gegeben. In Tabelle 1.1 wurden die Anwendungsgebiete des CHASEs nach Art der übergebenen Abhängigkeiten und Objekte definiert, wobei sich die ersten drei Attribute der Tabelle aus der allgemeinen Definition des CHASEs ergeben:

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

Der Parameter $\star$ symbolisiert eine Menge von Abhängigkeiten, die in ein Objekt $\bigcirc$, mit dem Ergebnis $\circledast$, eingechaset werden.

	Parameter $\star$	Objekt $\bigcirc$	Ergebnis $\circledast$	Ziel/Anwendungsgebiet
0.	Abhängigkeiten	DB-Schema[2]	DB-Schema mit Integritätsbedingungen	optimierter DB-Entwurf
I.	Abhängigkeiten	Anfragen	Anfragen	Semantische Optimierung
II.	Sichten	Anfragen	Anfragen auf Sichten	AQuV
II′.	Operationen	Anfragen	Anfragen auf Operationen	AQuO
III.	s-t TGDs, EGDs, TGDs	Quell-DB	Ziel-DB	Datenaustausch, Datenintegration
IV.	TGDs, EGDs	DB	modifizierte DB	Cleaning
V.	TGDs, EGDs	unvollständige DB	Anfrageergebnis	sichere Antworten
VI.	s-t TGDs, EGDs, TGDs	DB	Anfrageergebnis	invertierbare Auswertung

Tabelle 1.1.: Übersicht Anwendungsgebiete CHASE [AH19]

Der Fall *0* spiegelt die ursprüngliche Idee des CHASEs wieder – einen Datenbank-Entwurf mithilfe von Abhängigkeiten zu optimieren. Die Rede ist von funktionalen, Verbund-, und mehrwertigen Abhängigkeiten, die z. B. sicherstellen sollen, dass eine Zerlegung in Relationenschemata verlustfreie Verbunde garantiert [ABU79, MMS79].

Diese Abhängigkeiten können auch in Anfragen an eine Datenbank eingechaset werden. Das kann sinnvoll sein, wenn die Anfrage Verbunde enthält, die auf Grund von bestehenden Schlüsselbeziehungen irrelevant sind. Durch das Entfernen von überflüssigen Verbunden können Anfragen so um ein Vielfaches beschleunigt werden (Fall I). In Abschnitt 3.3 werden wir uns das Tool `PDQ` angucken, welches sich auf genau diesen Fall spezialisiert hat. In den Fällen II und II' sind es Sichten oder Operationen, die in die Anfragen eingechaset werden. Damit kann ermöglicht werden, Anfragen anhand von Sichten und Operatoren zu beantworten, ohne dass ein Zugriff auf den ganzen Datenbestand notwendig wird. Diese Verfahren werden *Answering Queries Using Views* bzw. *Answering Queries Using Operators* (AQuV bzw. AQuO) genannt. Beim AQuV werden beispielsweise in mobilen Anwendungen Speicherplatz und kostenintensive Zugriffe auf Remoteserver gespart, indem die Anfragen auf die lokal verfügbaren Sichten umgeschrieben werden [LMS95].

[2] PJ-Mapping

Neben dem CHASE auf Anfragen und Schemata, lässt sich der Algorithmus auch auf Datenbanken anwenden. Da werden vor allem verschiedene Bedingungen in Form von *TGDs*, *EGDs* und *s-t TGDs* (siehe Definitionen 2.6, 2.7) in Instanzen eingearbeitet. Das Ergebnis liegt dann, wie im Fall III, in Form von einer Zieldatenbank vor, welche Austauschdaten oder Integrationsdaten beinhaltet. Im Fall IV erhält man eine modifizierte Datenbank, die mithilfe der Bedingungen gecleant worden ist. Das *Cleaning* besteht vor allem darin, Duplikate zu eliminieren und Lücken in den Daten zu füllen. Es werden aber auch Inkonsistenzen in Datensätzen erkannt und in einigen Fällen sogar automatisch behoben. Die letzten beiden Fälle sind die Kern-Anwendungen vom Tool `Llunatic`, welches wir im Abschnitt 3.4 noch einmal genauer betrachten werden. Zuletzt kann das Ergebnis des CHASE aber auch ein Anfrageergebnis sein. So können im Fall V sichere Antworten an eine vorher unvollständige Datenbank garantiert werden, oder wie im Fall VI die Herkunft der Daten in der sogenannten invertierten Auswertung einer Datenbank geklärt werden (auch *Provenance Management* genannt – mehr dazu in [AH19]).

Diese Arbeit wird sich hauptsächlich auf die Fälle I, III und IV beziehen, wobei sich die Konzepte auch auf die anderen Anwendungsfälle übertragen lassen. Zur Veranschaulichung unserer weiteren Ausführungen geben wir im nächsten Abschnitt ein grundlegendes Beispiel an, welches sich durch die gesamte Ausarbeitung ziehen wird.

1.2. Allgemeines Beispiel

Die in den Beispielen der Grundlagen (Kapitel 2) und Implementierung (Kapitel 5) verwendeten Daten spiegeln eine typische universitäre Situation wider. In dieser gibt es Studenten, Teilnehmer von Kursen und Noten, die die Studenten in den Kursen erhalten können. Vereinfacht bestehen unsere Relationen aus den folgenden drei Schemata und den zugehörigen Instanzen.

- STUDENTEN = {Matrikelnr, Nachname, Vorname, Studiengang}
  - dom(Matrikelnr) := $\mathbb{N}$ = {1, 2, 3, ...}
  - dom(Nachname) := STRING = {Fieber, Sonnenschein, Mueller,...[3]}
  - dom(Vorname) := STRING = {Fabian, Sarah, Max,...[4]}
  - dom(Studiengang) := {Informatik, ITTI}
- TEILNEHMER = {Modulnr, Matrikelnr}
  - dom(Modulnr) := $\mathbb{N}$
  - dom(Matrikelnr) := $\mathbb{N}$
- NOTEN = {Modulnr, Matrikelnr, Semester, Note}
  - dom(Modulnr) := $\mathbb{N}$
  - dom(Matrikelnr) := $\mathbb{N}$
  - dom(Semester) := {WS 18/19, SS 19[5]}
  - dom(Note) := {1.0, 1.3, 1.7, 2.0, 2.3, 2.7, 3.0, 3.3, 3.7, 4.0, 5.0}

[3] Weitere Nachnamen vom Datentyp STRING
[4] Weitere Vornamen vom Datentyp STRING
[5] Eingeschränkt auf die relevanten Attributwerte

Matrikelnr	Nachname	Vorname	Studiengang
1	Fieber	Fabian	Informatik
2	Sonnenschein	Sarah	ITTI
3	Mueller	Max	Informatik

Tabelle 1.2.: STUDENTEN-Relation

Modulnr	Matrikelnr
1	1
1	3
2	3
3	1
3	2

Tabelle 1.3.: TEILNEHMER-Relation

Modulnummer	Matrikelnr	Semester	Note
1	1	WS 18/19	1.7
1	3	SS 19	3.0
2	3	SS 19	2.3
3	1	WS 18/19	1.0
3	2	WS 18/19	1.3

Tabelle 1.4.: NOTEN-Relation

Die Instanzen in den Tabellen 1.2, 1.3 und 1.4 sind nur als Beispieldatensatz zu sehen. Die genauen Attributwerte der Instanzen werden jeweils vor dem Gebrauch definiert. Dabei nutzen wir Ausdrücke der Form $R_i(w_1, ..., w_p)$ mit R_i als Relationennamen und w_i als Attribut- oder Nullwerte eines Tupels. Die Tabelle 1.2 würde als Instanz I somit wie folgt definiert werden:

$$\begin{aligned} I = \{&\text{STUDENTEN}(1, \text{Fieber}, \text{Fabian}, \text{Informatik}),\\ &\text{STUDENTEN}(2, \text{Sonnenschein}, \text{Sarah}, \text{ITTI}),\\ &\text{STUDENTEN}(3, \text{Mueller}, \text{Max}, \text{Informatik})\}. \end{aligned}$$

1.3. Aufbau der Arbeit

Um das Konzept dieser Arbeit nachvollziehbar aufzubereiten, werden im 2. Kapitel kurz die Grundlagen der relationenalgebraischen Konzepte erklärt. Auch der CHASE auf Anfragen und Instanzen sowie die jeweils erforderlichen Ersetzungsregeln werden bereits hier erläutert. Mithilfe von Beispielen (basierend auf dem Beispiel aus Abschnitt 1.2) wollen wir zeigen, dass der CHASE auf verschiedenen Objekten grundsätzlich gleich abläuft und somit auch programmiertechnisch verallgemeinert werden kann.

Das 3. Kapitel gibt einen kleinen Überblick über den Stand der Technik. Die Schwerpunkte werden hier auf der bisherigen Implementierung von ChaTEAU, dem Java-Toolkit `Graal`, dem Anfragen-Optimierungstool `PDQ` und dem universellen Mapping- und Cleaning-Tool `Llunatic` liegen.

Im Kapitel 4 wird dann das Konzept des allgemeinen CHASE-Objektes und der universalen CHASE-Methode von ChaTEAU vorgestellt. Mit dem Ziel, Instanzen und Anfragen im selben Algorithmus verarbeiten zu können, wird unter anderem gezeigt, wie sich Anfragen als Instanzen und Instanzen als Anfragen schreiben lassen sowie welche Vor- und Nachteile beide Varianten haben. Außerdem klären wir, an welchen Stellen im CHASE-Algorithmus eine Unterscheidung der beiden Objekte unbedingt notwendig bleibt.

Im darauffolgenden 5. Kapitel wird die Implementierung der universalen CHASE-Methode mit aktuell laufenden Beispielen aufgeführt. Dazu wird ebenfalls beschrieben, an welchen Stellen und warum die bestehende Implementierung zusätzlich zum Konzept verändert wurde, um die Funktionalität ChaTEAUs zu korrigieren und zu erweitern.

Das Kapitel 6 zum Fazit und Ausblick wartet neben einer Zusammenfassung dieser Arbeit auch noch mit einigen offenen Problemen und einem Ausblick auf, welcher den Ansatz für künftige Aufgaben bilden könnte. Im Anhang befinden sich, neben dem Literaturverzeichnissen, zusätzlich noch einige laufende Beispiele von ChaTEAU mit Ein- und Ausgabe zum Nachlesen.

2. Grundlagen

Der erste Abschnitt dieses Kapitels definiert kurz die beiden in dieser Arbeit relevanten CHASE-Objekte (Instanzen und Anfragen) sowie die zugehörigen Integritätsbedingungen (nach [HSS18, CM77, AH18, DHI12]). Im zweiten Abschnitt wird der allgemeine CHASE-Algorithmus jeweils für Instanzen und für Anfragen anschaulich erklärt (nach: [FKMP03]). Die Aufteilung rührt daher, dass für die verschiedenen CHASE-Objekte Unterschiede in den *Ersetzungsregeln* bei der Suche nach und der Anwendung von den sogenannten *Triggern* (Definition 2.9) besteht. Der auf Instanzen angewandte Algorithmus wurde zuerst in der Arbeit [AH19] vorgestellt. Für Anfragen wurde der Algorithmus in [Ren19] leicht angepasst. In dieser Arbeit werden beide Algorithmen sowie der Großteil der Definitionen an eine einheitliche Schreibweise angeglichen oder erweitert. Der CHASE auf Anfragen wird äquivalent zum CHASE auf Instanzen definiert. Definitionen, die entsprechend aus einer Quelle umgeformt wurden, sind mit dem Stichwort "nach" zitiert. Ein paar Worte zu den Unterschieden verschiedener CHASE-Varianten (siehe Abschnitt 2.2.3) und der *BACKCHASE-Phase* (siehe Abschnitt 2.2.4) sowie eine Einordnung der Grundlagen in den Kontext dieser Arbeit (siehe Abschnitt 2.3) bilden den Abschluss des Kapitels.

2.1. Relationale Konzepte

Bevor wir Instanzen und Anfragen formal definieren können, müssen noch ein paar weitere Grundbegriffe der Datenbanktheorie geklärt werden.

Definition 2.1. (Attribute und Wertebereiche, [HSS18]): Sei die nicht-leere endliche Menge $\mathcal{U}$ das *Universum* der Attribute. Ein $\mathcal{A} \in \mathcal{U}$ heißt *Attribut*. Sei $\mathcal{D} = \{D_1, ..., D_m\}$ die nicht-leere endliche Menge der *Wertebereiche* D_i mit $m \in \mathbb{N}$, so existiert die total definierte Funktion dom: $\mathcal{U} \longrightarrow \mathcal{D}$. Der Funktionswert von dom(A) ist der *Wertebereich* von A.

Definition 2.2. (Relationenschemata und Relationen, [HSS18]): Ein *Relationenschema* ist eine Menge $R \subseteq \mathcal{U}$. Für $R = \{A_1, ..., A_n\}$ mit $n \in \mathbb{N}$, ist die *Relation r über R* (geschrieben: *r(R)*) als eine endliche Menge von Abbildungen

$$t : R \longrightarrow \bigcup_{i=1}^{m} D_i$$

die *Tupel* genannt werden definiert, für die $t(a) \in dom(A)$ gilt. $t(A)$ schränkt dabei die Abbildung t auf $A \in R$ ein.

Definition 2.3. (Datenbankschema, [HSS18]): Ein *Datenbankschema* ist eine Menge von Relationenschemata $S := \{R_1, ..., R_p\}$ mit $p \in \mathbb{N}$.

Instanzen Eine Instanz repräsentiert die aktuell vorhandenen Daten einer Datenbank. Äquivalent zur Instanz einer Datenbank stellt die Instanz einer Relation die Daten zu einem bestimmten Relationenschema da.

Definition 2.4. (Instanz, [HSS18]): Für ein Datenbankschema S mit Relationenschemata $R_1, ..., R_p$, nennen wir die Menge von Relationen $d := \{r_1, ..., r_p\}$ *Instanz* der Datenbank mit $r_i(R_i)$ $\forall i \in \{1, ..., p\}$.

Für unser Allgemeines Beispiel wären die Datensätze der Tabelle 1.2 eine Instanz zum Relationenschema STUDENTEN = {Matrikelnr, Nachname, Vorname, Studiengang}. Die einzelnen Tupel der STUDENTEN-Relation entsprechen den drei Zeilen unter dem Relationenschema im Tabellenkopf. Wenn wir später von unserer Beispielinstanz sprechen werden, dann ist immer (soweit nicht anders angegeben) die Datenbankinstanz, bestehend aus den Datensätzen der Relationen in den Tabellen 1.2, 1.3 und 1.4, unter dem Datenbankschema S={STUDENTEN, TEILNEHMER, NOTEN} gemeint.

Als Nächstes werden wir Anfragen formal definieren. In der Einleitung haben wir zwar schon Beispielanfragen in der Datenbanksprache SQL formuliert, das entspricht aber nicht der Darstellung, welche wir im weiteren Verlauf der Arbeit verwenden werden.

Anfragen Mithilfe von Anfragen kann man über den Tabellen einer Datenbank Operationen ausführen, deren Ergebnis wieder eine Tabelle ist. Die dabei angewandten Operationen sind unter anderen die Projektion auf Attribute, die Selektion nach Konstanten, das Kreuzprodukt mehrerer Tabellen oder der Join von Tabellen in verschiedenen Varianten. Im Kontext dieser Arbeit werden wir uns auf die konjunktiven Anfragen beschränken.

Definition 2.5. (Konjunktive Anfragen, nach [CM77]): Eine *konjunktive Anfrage* (verkürzt *Anfrage* genannt), ist ein prädikatenlogischer Ausdruck erster Ordnung der Form

$$\exists y : \phi(x, y) \longrightarrow x_1, ..., x_m$$

wobei $\phi(x, y)$ eine Konjunktion von atomaren Ausdrücken der Form $R_i(a_1, ...a_p)$ ist und jedes a_j (mit $j \in \{1, ..., p\}$) entweder eine *ausgezeichnete Variable* aus x, eine *existenzquantifizierte Variable* aus y oder eine Konstante ist. Wir schreiben x und y als Abkürzung für die Vektoren $x :=< x_1, ..., x_m >$ und $y :=< y_1, ..., y_r >$. Der Bezeichner R_i steht für einen Relationennamen. Für die ausgezeichneten Variablen $x_1, ..., x_m$ (genannt: *Kopf der Anfrage*) gilt, dass sie in den Ausdrücken in $\phi(x, y)$ (genannt: *Rumpf der Anfrage*) enthalten sind. Andersherum definieren alle ausgezeichneten Variablen im Rumpf der Anfrage ihren Kopf.

In anderen Arbeiten wird anstelle von ausgezeichneten Variablen auch oft der Begriff "freie" Variablen verwendet. Die existenzquantifizierten Variablen werden auch "nicht ausgezeichnete" Variablen genannt. Der Kopf einer Anfrage spiegelt die Projektionsattribute der `SELECT`-Klausel einer SQL-Anfrage wider. Die Ausdrücke $\phi(x, y)$ im Rumpf der Anfrage sind mit den Relationen in der `FROM`-Klausel (wenn Konstanten in der Anfrage sind auch mit Selektionen aus der `WHERE`-Klausel) vergleichbar. Die Beispiel-Anfrage aus der Einleitung

```
SELECT Matrikelnr, Modulnr
FROM STUDENTEN NATURAL JOIN NOTEN
```

würde umgeformt in die neue Schreibweise wie folgt aussehen:

$$(\exists y_{Na}, y_{Vo}, y_{St}, y_{Se}, y_{No} : \text{STUDENTEN}(x_{Ma}, y_{Na}, y_{Vo}, y_{St}) \wedge \text{NOTEN}(x_{Mo}, x_{Ma}, y_{Se}, y_{No})) \longrightarrow x_{Ma}, x_{Mo}$$ [1]

Nach den Definitionen der CHASE-Objekte werden wir uns jetzt den für uns wichtigen CHASE-Parametern widmen. In den von uns betrachteten Fällen sind das *Integritätsbedingungen*.

[1] Attributbezeichner werden der Übersichtlichkeit halber immer auf 2 Character gekürzt, d.h. Ma = Matrikelnr, Na = Nachname, Vo = Vorname, St = Studiengang, Mo = Modulnr, Se = Semester, No = Note.

Integritätsbedingungen Allgemein betrachtet sind Integritätsbedingungen deskriptive Bedingungen an einen Datenbestand, die die möglichen Zustände einer Datenbank limitieren, um eine gewisse Konsistenz zu gewährleisten. Die bekanntesten Bedingungen sind unter anderem *Schlüssel*, die für die Tupel einer Relation eine identifizierende Attributmenge bilden, oder *funktionale Abhängigkeiten* (kurz: FD), bei denen eine bestimmte Attributmenge in einer Relation den Wert einer anderen Attributmenge bestimmt. Aber auch *Verbund-Abhängigkeiten* (kurz: JD), die garantieren, dass die Zerlegung einer Relation beim Verbund wieder die Originalrelation selbst ergibt, können als solche Bedingungen gezählt werden. In dieser Arbeit werden wir die Integritätsbedingungen mit einem allgemeineren Formalismus spezifizieren – den sogenannten *tuple-generating dependencies* und *equality-generating dependencies* (kurz: TGDs und EGDs).

Definition 2.6. (TGD, [DHI12, AH18]): *Tuple-generating dependencies* sind prädikatenlogische Ausdrücke erster Ordnung der Form

$$\forall x : (\phi(x) \longrightarrow \exists y : \psi(x, y)),$$

wobei $\phi(x)$ und $\psi(x, y)$ Konjunktionen von atomaren Ausdrücken der Form $R_i(a_1, ...a_p)$ sind und jedes a_j (mit $j \in \{1, ..., p\}$) für $\phi(x)$ eine allquantifizierte Variable aus x bzw. für $\psi(x, y)$ eine allquantifizierte Variable aus x oder eine existenzquantifizierte Variable aus y ist. Der Bezeichner R_i steht wie bei den Anfragen für einen Relationennamen. Jede Variable aus x kommt in $\phi(x)$ aber nicht notwendiger Weise in $\psi(x, y)$ vor. Eine TGD, bei der $\phi(x)$ ein Quellschema S und $\psi(x, y)$ ein Zielschema T repräsentieren, nennt man auch *source-to-target tuple-generating dependency* (kurz: s-t TGD).

Definition 2.7. (EGD, [DHI12, AH18]): *Equality-generating dependencies* sind prädikatenlogische Ausdrücke erster Ordnung der Form

$$\forall x : (\phi(x) \longrightarrow (x_1 = x_2), ..., (x_n = x_m)),$$

wobei $\phi(x)$ eine Konjunktion von atomaren Ausdrücken der Form $R_i(a_1, ...a_p)$ ist und jedes a_j (mit $j \in \{1, ..., p\}$) eine allquantifizierte Variable aus x ist. Der Bezeichner R_i steht wie bei den Anfragen und TGDs für einen Relationennamen. Alle Variablen in den Gleichheitsatomen auf der rechten Seite kommen auch in $\phi(x)$ vor.

Äquivalent zu den Anfragen werden wir $\phi(x)$ als den *Rumpf einer TGD* bzw. *EGD* bezeichen und $\psi(x, y)$ in TGDs bzw. die Gleichheitsatome der EGDs als den *Kopf* einer solchen. TGDs haben nur im Kopf existenzquantifizierte Variablen. Im Gegensatz dazu tauchen sie bei Anfragen nur im Rumpf auf und in EGDs gar nicht. Ein Anfragekopf ist außerdem nur eine Menge von ausgezeichneten Variablen, während TGDs und EGDs komplexere Strukturen im Kopf ausweisen. Der Einfachheit halber werden wir im Weiteren die Allquantifizierung von x vor den TGDs und EGDs weglassen.

Zur weiteren Veranschaulichung werden wir jetzt die Beispielbedingung aus der Einleitung als TGD formulieren. Diese forderte, dass alle Studenten aus STUDENTEN, die eine Note in NOTEN haben, auch als Teilnehmer des entsprechenden Moduls in TEILNEHMER auftauchen müssen. Aus den Relationen STUDENTEN und NOTEN wollen wir also Tupel für die TEILNEHMER-Relation generieren. So formuliert erkennt man, dass STUDENTEN und NOTEN im TGD-Rumpf auftauchen müssen und die TEILNEHMER in den TGD-Kopf gehören:

$$\text{STUDENTEN}(x_{Ma}, x_{Na}, x_{Vo}, x_{St}) \wedge \text{NOTEN}(x_{Mo}, x_{Ma}, x_{Se}, x_{No}) \longrightarrow \text{TEILNEHMER}(x_{Mo}, x_{Ma}).$$

Da in der TEILNEHMER-Relation keine Attribute vorkommen, die es nicht in der STUDENTEN- oder NOTEN-Relation gibt, stehen im TGD-Kopf keine existenzquantifizierten y-Variablen. Anders wäre es bei der TGD: Alle Studenten, die einen Eintrag in der TEILNEHMER-Relation haben, müssen auch in der STUDENTEN-Relation auftauchen:

$$\text{TEILNEHMER}(x_{Mo}, x_{Ma}) \longrightarrow \exists y_{Na}, y_{Vo}, y_{St} : \text{STUDENTEN}(x_{Ma}, y_{Na}, y_{Vo}, y_{St}).$$

Da die Attribute für Nachname, Vorname und Studiengang nicht im Schema der TEILNEHMER-Relation vorkommen (TEILNEHMER={Modulnr, Matrikelnr}), werden sie als existenzquantifizierte y-Variablen geschrieben.

Eine Beispiel-EGD könnte lauten: Wenn in der STUDENTEN-Relation Tupel für Studenten mit der gleichen Matrikelnummer vorkommen, dann müssen auch die Namen übereinstimmen:

$$\begin{aligned}&\text{STUDENTEN}(x_{Ma}, x_{Na1}, x_{Vo1}, x_{St1}) \wedge \text{STUDENTEN}(x_{Ma}, x_{Na2}, x_{Vo2}, x_{St2})\\ &\longrightarrow (x_{Na1} = x_{Na2}), (x_{Vo1} = x_{Vo2}).\end{aligned}$$

Die Bedingung der gleichen Matrikelnummern wird über die Gleichbenennung der Variablen für die Matrikelnummer (x_{Ma}) im EGD-Rumpf angegeben. Die daraus folgende Gleichheit des Namens steht in den Gleichheitsatomen im Kopf der EGD.

Wie genau der CHASE prüft, ob eine Bedingung angewandt werden muss und wie dann Instanzen und Anfragen durch eine TGD erweitert werden oder eine EGD auf ihnen ausgewertet wird, klären wir ausführlich im nächsten Abschnitt.

2.2. CHASE

Nach der Definition der relationalen Konzepte können wir jetzt den CHASE formal definieren. Da dem CHASE verschiedene Objekte und Parameter übergeben werden können, variiert der Algorithmus leicht bei den intern verwendeten Ersetzungsregeln. Anhand der Abschnitte werden wir aber erkennen, dass alle anderen Schritte des Algorithmus auf Instanzen und auf Anfragen gleich angewandt werden. Zuerst definieren wir die grundlegenden allgemeinen Begriffe wie *Homomorphismen* und *Trigger*. Danach widmen wir uns einzeln dem CHASE auf Instanzen und Anfragen zu.

Die Hauptaufgabe des CHASEs ist die Suche nach Abbildungen zwischen den Integritätsbedingungen in $\mathcal{B} := \{b_1, ..., b_m\}$ ($\star$ in Tabelle 1.1) und einem CHASE-Objekt $\mathcal{O}_i$ ($\bigcirc$ in Tabelle 1.1), den sogenannten *Homomorphismen*. Mithilfe dieser Homomorphismen kann der CHASE die einzelnen Integritätsbedingungen $b_i \in \mathcal{B}$ in $\mathcal{O}_i$ einchasen. Als Ergebnis erhalten wir ein modifiziertes CHASE-Objekt $\mathcal{O}_{i+1}$.

Zwischen $\mathcal{O}_i$ und $\mathcal{O}_{i+1}$ entsteht dann ebenfalls ein Homomorphismus, welcher einigen Ersetzungsregeln unterliegt. Diese Ersetzungsregeln unterscheiden sich jeweils für Instanzen und Anfragen.

Definition 2.8. (Homomorphismus, nach [GMS12]): Seien $\mathcal{O}_1$ und $\mathcal{O}_2$ zwei CHASE-Objekte. Ein *Homomorphismus* $h : \mathcal{O}_1 \longrightarrow \mathcal{O}_2$ ist eine Abbildung unter folgenden Regeln:

(a) Seien $\mathcal{O}_1$ und $\mathcal{O}_2$ Instanzen über denselbem Schema. Wir beziehen uns auf die Konstanten und Nullwerte in den Tupeln der Relationen.

(a1) Eine Konstante c kann nur auf sich selbst abgebildet werden: $h(c) = c$.

(a2) Ein Nullwert μ_i kann auf eine Konstante c, sich selbst oder einen anderen Nullwert μ_j abgebildet werden: $h(\mu_i) = [c|\mu_i|\mu_j]$.

Dabei gilt: für jedes Tupel $t = (a_1, ..., a_n)$ aus den Relationen in $\mathcal{O}_1$ existiert ein Tupel $h(t)$ in den Relationen aus $\mathcal{O}_2$, mit $h(t) = (h(a_1), ..., h(a_n))$.

(b) Seien $\mathcal{O}_1$ und $\mathcal{O}_2$ Anfragen mit den gleichen Bezeichnern für gleiche Relationen und Attribute.

(b1) Eine Konstante c kann nur auf sich selbst abgebildet werden: $h(c) = c$.

(b2) Eine ausgezeichnete Variable x_i kann nur auf eine Konstante c oder auf sich selbst abgebildet werden: $h(x_i) = [c|x_i]$.

(b3) Eine existenzquantifizierte Variable y_i kann auf eine Konstante c, auf eine ausgezeichnete Variable x_j, auf sich selbst oder auf eine andere existenzquantifizierte Variable y_j abgebildet werden: $h(y_i) = [c|x_j|y_i|y_j]$.

Dabei gilt für $x :=< x_1, ..., x_m >$ und $y :=< y_1, ..., y_r >$ aus $\phi(x, y)$ in $\mathcal{O}_1$, dass $\phi(h(x), h(y))$ in $\mathcal{O}_2$ mit $h(x) :=< h(x_1), ..., h(x_m) >$ und $h(y) :=< h(y_1), ..., h(y_r) >$ definiert ist.

Die CHASE-Objekte $\mathcal{O}_1$ und $\mathcal{O}_2$ gelten genau dann als *äquivalent* (geschrieben: $\mathcal{O}_1 \longleftrightarrow \mathcal{O}_2$), wenn es einen Homomorphismus von $\mathcal{O}_1$ nach $\mathcal{O}_2$ ($h_1 : \mathcal{O}_1 \longrightarrow \mathcal{O}_2$) und einen Homomorphismus von $\mathcal{O}_2$ nach $\mathcal{O}_1$ ($h_2 : \mathcal{O}_2 \longrightarrow \mathcal{O}_1$) gibt.

Seien zwei Beispielinstanzen I_1 und I_2 wie folgt gegeben:

$$I_1 = \{\text{STUDENTEN}(1,\ \mu_{Na}\ , \text{Fabian}, \text{Informatik})\},$$
$$I_2 = \{\text{STUDENTEN}(1, \text{Fieber}, \text{Fabian}, \text{Informatik})\}.$$

Instanz I_1 enthält statt eines konkreten Attributwertes einen Nullwert μ_{Na} für das Attribut Nachname. Dann kann man einen Homomorphismus ($h : I_1 \longrightarrow I_2$) definieren, der μ_{Na} auf die Konstante "Fieber" abbildet ($h(\mu_{Na}) =$ "Fieber") und die anderen Attributwerte auf sich selbst ($h(1) = 1,\ h($"Fabian"$) =$ "Fabian", $h($"Informatik"$) =$ "Informatik").

Für die Beispielanfragen

$$Q_1 = \exists y_{Na}, y_{Vo} : \text{STUDENTEN}(x_{Ma}, y_{Na}, y_{Vo}, x_{St}) \longrightarrow x_{Ma}, x_{St},$$
$$Q_2 = \exists y_{Vo} : \text{STUDENTEN}(x_{Ma}, \text{"Fieber"}, y_{Vo}, x_{St}) \longrightarrow x_{Ma}, x_{St}$$

existiert ein Homomorphismus ($h : Q_1 \longrightarrow Q_2$), der die existenzquantifizierte Variable y_{Na} auf die Konstante "Fieber" abbildet ($h(y_{Na}) =$ "Fieber") und die anderen Variablen auf sich selbst ($h(x_{Ma}) = x_{Ma},\ h(y_{Vo}) = y_{Vo},\ h(x_{St}) = x_{St}$).

Definition 2.9. (Trigger, nach [Jur18, Ren19]): Die Homomorphismen zwischen den Integritätsbedingungen in $\mathcal{B}$ ($b_i = [TGD|EGD]$) und einem CHASE-Objekt $\mathcal{O}$ werden *Trigger* genannt. Dabei wird der Rumpf von b_i auf $\mathcal{O}$ abgebildet.

(a) Sei $\mathcal{O}$ eine Instanz I.

- Der Rumpf von b_i wird auf die Tupel aus den Relationen in I abgebildet: $g_i : \phi(x) \longrightarrow I$, wobei die einzelnen Variablen x_i aus den atomaren Ausdrücken von $\phi(x)$ auf die einzelnen Attribut- und Nullwerte der Tupel abgebildet werden.

(b) Sei $\mathcal{O}$ eine Anfrage Q.

- Der Rumpf von b_i wird auf die atomaren Ausdrücke in $\phi(x, y)$ aus Q abgebildet: $g_i : \phi(x) \longrightarrow Q$, wobei die einzelnen Variablen x_i aus den atomaren Ausdrücken von $\phi(x)$ auf die einzelnen Variablen x_k, y_r und Konstanten in $\phi(x, y)$ aus Q abgebildet werden.

Im Weiteren werden wir meistens [für diese Abbildungen der Variablen aus den Ausdrücken der Integritätsbedingungen auf die Attribut- und Nullwerte der Tupel aus den Instanzen oder die Variablen und Konstanten der Ausdrücke aus den Anfragen] verkürzt die Abbildung von den Ausdrücken der Integritätsbedingungen auf die Tupel der Instanzen oder die Ausdrücke der Anfragen schreiben.

Definition 2.10. (Aktiver Trigger, nach [Jur18, Ren19]): Ein *aktiver Trigger* ist ein Trigger g_i, für den gilt:

- Wenn b_i eine TGD ist, dann existiert keine Erweiterung von g_i zu einem Homomorphismus vom Kopf der TGD zu $\mathcal{O}$ ($\psi(x, y) \longrightarrow \mathcal{O}$).

- Wenn b_i eine EGD ist, dann ist $g_i(x_i) \neq g_i(x_j)$ für mindestens ein Gleichheitsatom $(x_i = x_j)$ aus dem Kopf der EGD.

Ein CHASE-Objekt $\mathcal{O}$ *genügt* b_i, genau dann wenn kein aktiver Trigger für b_i in $\mathcal{O}$ existiert.

Seien folgende CHASE-Objekte I, Q und die Integritätsbedingungen b_1 (TGD), b_2 (EGD) gegeben.

$$
\begin{aligned}
I =& \{\text{STUDENTEN}(1,\ \mu_{Na}\ , \text{Fabian}, \text{Informatik}),\\
&\ \text{STUDENTEN}(1, \text{Fieber}, \text{Fabian}, \text{Informatik}),\\
&\ \text{TEILNEHMER}(2, 1),\\
&\ \text{NOTEN}(2, 1, \text{WS } 18/19, 3.7)\}\\
Q =& (\exists y_{Na}, y_{Vo}, y_{St} : \text{STUDENTEN}(x_{Ma}, y_{Na}, y_{Vo}, y_{St}) \wedge \text{TEILNEHMER}(x_{Mo}, x_{Ma}))\\
&\longrightarrow x_{Ma}, x_{Mo}\\
b_1 =& \text{STUDENTEN}(x_{Ma}, x_{Na}, x_{Vo}, x_{St}) \wedge \text{TEILNEHMER}(x_{Mo}, x_{Ma})\\
&\longrightarrow \exists y_{Se}, y_{No} : \text{NOTEN}(x_{Mo}, x_{Ma}, y_{Se}, y_{No})\\
b_2 =& \text{STUDENTEN}(x_{Ma}, x_{Na1}, x_{Vo1}, x_{St1}) \wedge \text{STUDENTEN}(x_{Ma}, x_{Na2}, x_{Vo2}, x_{St2})\\
&\longrightarrow (x_{Na1} = x_{Na2}), (x_{Vo1} = x_{Vo2})
\end{aligned}
$$

Für die Instanz I und die Integritätsbedingung b_1 existieren zwei Trigger. Beide bilden den STUDENTEN-Ausdruck aus dem Rumpf von b_1 jeweils auf eines der STUDENTEN-Tupel aus I ab und den TEILNEHMER-Ausdruck aus dem Rumpf von b_1 auf das TEILNEHMER-Tupel aus I. Da es auch eine Abbildung des NOTEN-Ausdrucks aus dem Kopf von b_1 auf das NOTEN-Tupel in I gibt, sind beide keine aktiven Trigger.

Im Gegensatz zu b_1 gibt es für die Integritätsbedingung b_2 einen aktiven Trigger auf I. Hier werden die STUDENTEN-Ausdrücke aus dem Rumpf von b_2 auf die beiden STUDENTEN-Tupel aus I abgebildet. Der Trigger ist aktiv, da für das Gleichheitsatom $(x_{Na1} = x_{Na2})$, mit Abbildungen $g_2(x_{Na1}) = \mu_{Na}$ und $g_2(x_{Na2}) =$ "Fieber", die Ungleichheit $g_2(x_{Na1}) \neq g_2(x_{Na2})$ gilt.

Für die Anfrage Q ist b_1 ein aktiver Trigger. Hier existiert eine Abbildung von den STUDENTEN- und TEILNEHMER-Ausdrücken des TGD-Rumpfes auf die STUDENTEN- und TEILNEHMER-Ausdrücke des Anfragerumpfes. Außerdem existiert keine Erweiterung des Triggers zu einem Homomorphismus von dem NOTEN-Ausdruck aus dem Kopf von b_1 zu einem Ausdruck aus Q.

Zwischen b_2 und Q existiert keine Abbildung der Ausdrücke der EGD auf die Ausdrücke der Anfrage und somit auch kein Trigger. Der CHASE würde b_2 also nicht auf Q anwenden können.

Definition 2.11. (CHASE-Schritt, [FKMP03]): Sei $g_i : \phi(x) \longrightarrow \mathcal{O}_i$ ein aktiver Trigger für eine Integritätsbedingung $b_i \in \mathcal{B}$ und ein CHASE-Objekt $\mathcal{O}_i$. Dann nennt man die Anwendung von b_i unter g_i auf $\mathcal{O}_i$ mit dem Ergebnis eines modifizieren CHASE-Objektes $\mathcal{O}_{i+1}$ einen *CHASE-Schritt* $\mathcal{O}_i \xrightarrow{b_i, g_i} \mathcal{O}_{i+1}$.

Wie genau ein CHASE-Schritt für TGDs und EGDs auf Instanzen und Anfragen abläuft, wird in den nachfolgenden Abschnitten definiert und an Beispielen erklärt. Der komplette Algorithmus ist jeweils in Standard-CHASE auf Instanzen($\mathcal{B}$, I) (siehe Abschnitt 2.2.1) und Standard-CHASE auf Anfragen($\mathcal{B}$, Q) (siehe Abschnitt 2.2.2) angegeben.

2.2.1. CHASE auf Instanzen

Algorithmus 1 beschreibt das Konzept des Standard-CHASE auf Instanzen. Die Zeilen 5 bis 11 stellen den CHASE-Schritt auf Instanzen dar.

Definition 2.12. (CHASE-Schritt auf Instanzen, [FKMP03]): Seien alle Bedingungen für einen CHASE-Schritt $I_i \xrightarrow{b_i, g_i} I_{i+1}$ nach Definition 2.11 mit einer Instanz I_i als CHASE-Objekt gegeben.

(a) Sei b_i eine TGD.

- Die Anwendung von b_i erweitert I_i um Tupel $R_j(w_1, ..., w_p)$ mit den Relationennamen R_j gleich den Bezeichnern der Ausdrücke $R_j(a_1, ..., a_p)$ aus dem Kopf der TGD. Falls a_k für eine existenzquantifizierte Variable y_k aus der TGD steht, wird der Homomorphismus g_i um die Abbildung von y_k auf einen neuen benannten Nullwert μ_k erweitert. Die Attribut- und Nullwerte w_k ergeben sich dabei aus der Anwendung des Homomorphismus auf die Variablen der Ausdrücke des TGD-Kopfes, $g_i(a_k)$.

(b) Sei b_i eine EGD.

- Falls für ein Gleichheitsatom $(x_i = x_j)$ die Abbildungen $g_i(x_i)$ und $g_i(x_j)$ unterschiedliche Konstanten ergeben, dann schlägt der CHASE fehl $(I_{i+1} = \perp)$.

- Falls eine Variable des Gleichheitsatoms $(x_i = x_j)$ auf eine Konstante abgebildet wird und die andere auf einen benannten Nullwert (z. B. $g_i(x_i) = 1$, $g_i(x_j) = \mu_{Ma}$), dann wird der Nullwert überall durch die Konstante ersetzt $(h(\mu_{Ma}) = 1$ mit $h : I_i \longrightarrow I_{i+1})$.

- Falls beide Variablen des Gleichheitsatoms $(x_i = x_j)$ auf unterschiedlich benannte Nullwerte abgebildet werden (z. B. $g_i(x_i) = \mu_{No1}$, $g_i(x_j) = \mu_{No2}$), dann wird ein Nullwert überall durch den anderen ersetzt. Die Auswahl erfolgt über den durch die Namenskonvention vorgeschriebenen Index der Nullwerte. Die Konvention besagt, dass verschiedene Nullwerte für ein Attribut durchnummeriert werden. So kann man den Nullwert mit dem größeren Index durch den Nullwert mit dem kleineren Index ersetzen $(h(\mu_{No2}) = \mu_{No1}$ mit $h : I_i \longrightarrow I_{i+1})$.

  Diese Regeln werden für jedes Gleichheitsatom aus b_i auf I_i angewandt. Bei der Anwendung wird ein Homomorphismus h zwischen den Instanzen I_i und I_{i+1} gebildet, der die Werte die ersetzt werden (z. B. $g_i(x_j) = \mu_{Ma}$) auf die ersetzenden Werte (z. B. $g_i(x_i) = 1$) abbildet $(h(g_i(x_j)) = g_i(x_i) \longrightarrow h(\mu_{Ma}) = 1$ mit $h : I_i \longrightarrow I_{i+1})$.

Die Anwendung eines CHASE-Schrittes überführt die Instanz I_i in eine modifizierte Instanz I_{i+1}. Schlägt ein CHASE-Schritt fehl, wird anstatt einer Instanz das Symbol für Falschheit $\perp$ zurückgegeben.

Definition 2.13. (Standard-CHASE auf Instanzen, [FKMP03]): Sei $\mathcal{B}$ eine Menge von TGDs und EGDs und I eine Instanz.

- Eine *CHASE-Sequenz* auf I mit $\mathcal{B}$ ist eine (endliche oder unendliche) Sequenz von CHASE-Schritten $I_i \xrightarrow{b_i,g_i} I_{i+1}$, mit $i \in \mathbb{N}_0, I_0 = I$ und b_i eine Integritätsbedingung aus $\mathcal{B}$.
- Ein *endlicher CHASE* auf I mit $\mathcal{B}$ ist eine endliche Sequenz $I_i \xrightarrow{b_i,g_i} I_{i+1}, 0 \leq i \leq m$, mit der Bedingung, dass entweder (a) $I_m = \perp$ oder (b) es existiert keine Integritätsbedingung b_m in $\mathcal{B}$ und es gibt keinen Homomorphismus g_m, sodass b_m mit g_m auf I_m abgebildet werden kann. Wir bezeichnen I_m als das Ergebnis des endlichen CHASEs – für den Fall (a) das Ergebnis eines *fehlgeschlagenen endlichen CHASEs* und für den Fall (b) das Ergebnis eines *erfolgreichen endlichen CHASEs*.

Mit anderen Worten werden beim Standard-CHASE solange CHASE-Schritte angewandt, bis der CHASE entweder fehlschlägt oder es keine aktiven Trigger mehr für alle $b_i \in \mathcal{B}$ auf dem CHASE-Objekt $\mathcal{O}_i$ gibt. Die Auswahl der Integritätsbedingungen erfolgt dabei nicht-deterministisch. Dieser Nichtdeterminismus in Verbindung mit möglichen nicht konfluenten Integritätsbedingungen führt dazu, dass die Terminierung des Standard-CHASEs nicht entscheidbar ist und verschiedene CHASE-Sequenzen über dem gleichen Objekt mit gleichen Integritätsbedingung unterschiedliche Ergebnisse haben können.

Algorithmus 1 Standard-CHASE auf Instanzen($\mathcal{B}$, I)

Require: Menge von Integritätsbedingungen (TGDs, EGDs) $\mathcal{B}$, Instanz I_0
Ensure: Modifizierte Instanz I_m

```
1:  while I_i ≠ ⊥ ∧ ∃ aktive Trigger für ein b ∈ B und I_i, 0 ≤ i ≤ m do
2:    for all b ∈ B do
3:      for all Trigger g für b und I_i do
4:        if g ist ein aktiver Trigger then
5:          if b ist eine TGD then
6:            Erweitere g falls nötig, füge neue Tupel zu der Instanz I_i hinzu
7:          else if b ist eine EGD then
8:            if Verglichene Werte sind verschiedene Konstanten then
9:              I_m = ⊥
10:           else
11:             Ersetzte Nullwerte durch anderen Nullwerten oder Konstanten
```

Seien die Instanz I und die Integritätsbedingung b (TGD) wie folgt gegeben:

$$
\begin{aligned}
I =& \{\text{STUDENTEN}(1, \text{Fieber}, \text{Fabian}, \text{Informatik}),\\
& \quad \text{TEILNEHMER}(2, 1)\}\\
b =& \text{STUDENTEN}(x_{Ma}, x_{Na}, x_{Vo}, x_{St}) \wedge \text{TEILNEHMER}(x_{Mo}, x_{Ma})\\
& \quad \longrightarrow \exists y_{Se}, y_{No} : \text{NOTEN}(x_{Mo}, x_{Ma}, y_{Se}, y_{No}).
\end{aligned}
$$

Wie in Algorithmus 1 in den Zeilen 1 bis 4 beschrieben, wird beim Standard-CHASE der CHASE-Schritt der Zeilen 5 bis 11 für alle aktiven Trigger g angewandt. Das geschieht für alle Integritätsbedingungen $b \in \mathcal{B}$, solange bis es für keine Integritätsbedingung einen aktiven Trigger auf I_m gibt oder der CHASE fehlschlägt.

In unserem Beispiel suchen wir also nach Triggern für die Integritätsbedingungen in $\mathcal{B} = \{b\}$ und die Instanz $I \equiv I_0$. Wählen wir als Erstes die Integritätsbedingung b (bei mehreren Integritätsbedingungen in $\mathcal{B}$ müssten wir theoretisch eine nicht-deterministische Auswahl treffen). Die Trigger sind zu finden, indem zwischen dem Rumpf von b und den Tupeln von I_0 nach Homomorphismen gesucht wird. Es existiert ein Homomorphismus g_0 mit jeweils einer Abbildung zwischen den STUDENTEN- und TEILNEHMER-Tupeln aus I_0 und den STUDENTEN- und TEILNEHMER-Ausdrücken im Rumpf von b ($g_0(x_{Ma}) = 1,\ g_0(x_{Na}) = \text{Fieber},\ g_0(x_{Vo}) = \text{Fabian},\ g_0(x_{St}) = \text{Informatik},\ g_0(x_{Mo}) = 2$).

Bei der Prüfung auf einen aktiven Trigger sucht man nach Erweiterungen von g_0 zu einem Homomorphismus, der den Kopf von b auf I_0 abbildet. Da es keine Abbildung vom NOTEN-Ausdruck nach I_0 gibt, können wir mit dem aktiven Trigger g_0 fortfahren und den CHASE-Schritt $I_0 \xrightarrow{b,g_0} I_1$ durchführen. Dabei unterscheidet man b nach TGDs und EGDs. In unserem Fall wird I_0 mit Tupeln, die aus der Anwendung von g_0 auf den Ausdrücken aus dem TGD-Kopf stammen, erweitert. Da im Kopf der TGD existenzquantifizierte Variablen vorkommen wird aber zuerst g_0 um die Abbildung von ihnen auf neue benannte Nullwerte erweitert ($g_0(y_{Se}) = \mu_{Se},\ g_0(y_{No}) = \mu_{No}$). Nun kann ein entsprechendes NOTEN-Tupel zu I_0 hinzugefügt werden. Da g_0 unser einziger Trigger für b ist und b die einzige Integritätsbedingung in $\mathcal{B}$, wird kein weiter CHASE-Schritt mehr ausgeführt und wir erhalten die modifizierte Instanz I_1:

$$\begin{aligned} I_1 = \{&\text{STUDENTEN}(1, \text{Fieber}, \text{Fabian}, \text{Informatik}),\\ &\text{TEILNEHMER}(2, 1),\\ &\text{NOTEN}(2, 1, \mu_{Se}, \mu_{No})\}. \end{aligned}$$

Im zweiten Durchlauf wählen wir wieder die Integritätsbedingung b und suchen nach Triggern zwischen ihr und I_1. Wieder finden wir nur einen Homomorphismus g_1 der jeweils die STUDENTEN- und TEILNEHMER-Tupel aus I_1 auf die STUDENTEN- und TEILNEHMER-Ausdrücke in b abbildet ($g_1(x_{Ma}) = 1,\ g_1(x_{Na}) = \text{Fieber},\ g_1(x_{Vo}) = \text{Fabian},\ g_1(x_{St}) = \text{Informatik},\ g_1(x_{Mo}) = 2$). Doch diesmal finden wir bei der Prüfung auf aktive Trigger eine Erweiterung von g_1 mit $g_1(y_{Se}) = \mu_{Se},\ g_1(y_{No}) = \mu_{No}$, zu einem Homomorphismus, der den NOTEN-Ausdruck vom TGD-Kopf auf das neue NOTEN-Tupel in I_1 abbildet. Somit handelt es sich bei g_1 um keinen aktiven Trigger zu I_1. Da g_1 unser einziger Trigger für b ist und b die einzige Integritätsbedingung in $\mathcal{B}$, wird kein CHASE-Schritt ausgeführt und wir erhalten die Instanz I_1 als endgültiges Ergebnis des Standard-CHASEs.

2.2.2. CHASE auf Anfragen

Der Hauptunterschied zum CHASE auf Instanzen liegt beim CHASE auf Anfragen in der Suche und Anwendung der Trigger im CHASE-Schritt. Genauer gesagt, liegt der Unterschied hauptsächlich in den Abbildungen zwischen den Integritätsbedingungen und der Anfrage sowie in den Ersetzungsregeln zwischen zwei Anfragen bei der Anwendung von EGDs. Wie im Algorithmus 2 zu sehen, sind alle anderen Schritte identisch, mit der Ausnahme, dass der CHASE auf Anfragen nicht fehlschlagen kann und stattdessen eine leere Anfrage liefert.

Die folgenden Definitionen und Beispiele sind bis auf die genannten Unterschiede und Feinheiten identisch zu denen aus dem vorherigen Abschnitt 2.2.1. Eine detailliertere Aufstellung der Unterschiede ist im Konzept dieser Arbeit (siehe Kapitel 4) zu finden.

Definition 2.14. (CHASE-Schritt auf Anfragen, nach [Ren19]): Seien alle Bedingungen für einen CHASE-Schritt $Q_i \xrightarrow{b_i,g_i} Q_{i+1}$ nach Definition 2.11 mit einer Anfrage Q_i als CHASE-Objekt gegeben.

(a) Sei b_i eine TGD.

- Die Anwendung von b_i erweitert den Rumpf von Q_i um die Ausdrücke aus dem Kopf der TGD. Falls der TGD-Kopf eine existenzquantifizierte Variable y_k enthält, wird der Homomorphismus g_i um die Abbildung von y_k auf eine für die Anfrage neue existenzquantifizierte Variable y_{ki} erweitert. Die Variablen und Konstanten der neuen Ausdrücke in der Anfrage ergeben sich dabei aus der Anwendung des Homomorphismus auf die Variablen der Ausdrücke des TGD-Kopfes.

(b) Sei b_i eine EGD.

- Falls für ein Gleichheitsatom $(x_i = x_j)$ die Abbildungen $g_i(x_i)$ und $g_i(x_j)$ unterschiedliche Konstanten ergeben, dann wird eine leere Anfrage zurückgegeben $(Q_{i+1} = \emptyset)$.

- Falls eine Variable des Gleichheitsatoms $(x_i = x_j)$ auf eine Konstante abgebildet wird und die andere auf eine existenzquantifizierte oder ausgezeichnete Variable (z. B. $g_i(x_i) = 1$, $g_i(x_j) = x_{Ma}$), dann wird die Variable überall durch die Konstante ersetzt $(h(x_{Ma}) = 1$ mit $h : Q_i \longrightarrow Q_{i+1})$.

- Falls eine Variable des Gleichheitsatoms $(x_i = x_j)$ auf eine ausgezeichnete Variable abgebildet wird und die andere auf eine existenzquantifizierte Variable (z. B. $g_i(x_i) = x_{Ma}$, $g_i(x_j) = y_{Ma}$), dann wird die existenzquantifizierte Variable überall durch die ausgezeichnete Variable ersetzt $(h(y_{Ma}) = x_{Ma}$ mit $h : Q_i \longrightarrow Q_{i+1})$.

- Falls beide Variablen des Gleichheitsatoms $(x_i = x_j)$ auf unterschiedlich existenzquantifizierte Variablen abgebildet werden (z. B. $g_i(x_i) = y_{No1},\ g_i(x_j) = y_{No2}$), dann wird eine existenzquantifizierte Variable überall durch die anderen ersetzt. Die Auswahl erfolgt über den durch die Namenskonvention vorgeschriebenen Index der existenzquantifizierten Variablen. Die Konvention besagt, dass verschiedene existenzquantifizierte Variablen für ein Attribut durchnummeriert werden. So kann man die Variablen mit dem größeren Index durch die Variablen mit dem kleineren Index ersetzen $(h(y_{No2}) = y_{No1}$ mit $h : Q_i \longrightarrow Q_{i+1})$.

  Diese Regeln werden für jedes Gleichheitsatom aus b_i auf Q_i angewandt. Bei der Anwendung wird ein Homomorphismus h zwischen den Anfragen Q_i und Q_{i+1} gebildet, der die Werte die ersetzt werden (z. B. $g_i(x_j) = \mu_{Ma}$) auf die ersetzenden Werte (z. B. $g_i(x_i) = 1$) abbildet $(h(g_i(x_j)) = g_i(x_i) \longrightarrow h(\mu_{Ma}) = 1$ mit $h : Q_i \longrightarrow Q_{i+1})$.

Die Anwendung eines CHASE-Schrittes überführt die Anfrage Q_i in eine modifizierte Anfrage Q_{i+1}. Schlägt ein CHASE-Schritt fehl, wird (anstatt dem Symbol für Falschheit $\perp$ wie bei Instanzen) eine leere Anfrage zurückgegeben.

Definition 2.15. (Standard-CHASE auf Anfragen, nach Schema von [FKMP03]): Sei $\mathcal{B}$ eine Menge von TGDs und EGDs und Q eine Anfrage. Dann gilt:

- Eine *CHASE-Sequenz* auf Q mit $\mathcal{B}$ ist eine (endliche oder unendliche) Sequenz von CHASE-Schritten $Q_i \xrightarrow{b_i,g_i} Q_{i+1}$, mit $i \in \mathbb{N}_0, Q_0 = Q$ und b_i eine Integritätsbedingung aus $\mathcal{B}$.

- Ein *endlicher CHASE* auf Q mit $\mathcal{B}$ ist eine endliche Sequenz $Q_i \xrightarrow{b_i,g_i} Q_{i+1}, 0 \leq i \leq m$, mit der Bedingung, dass entweder (a) $Q_m = \emptyset$ oder (b) es existiert keine Integritätsbedingung b_m in $\mathcal{B}$ und es gibt keinen Homomorphismus g_m, sodass b_m mit g_m auf den Rumpf von Q_m abgebildet werden kann. Wir bezeichnen Q_m als das Ergebnis des endlichen CHASEs – für den Fall (a) das Ergebnis eines *abgebrochenen endlichen CHASEs* und für den Fall (b) das Ergebnis eines *erfolgreichen endlichen CHASEs*.

Seien die Anfrage Q und die Integritätsbedingung b (TGD) wie folgt gegeben:

$$
\begin{aligned}
Q =& (\exists y_{Na}, y_{Vo}, y_{St} : \text{STUDENTEN}(x_{Ma}, y_{Na}, y_{Vo}, y_{St}) \wedge \text{TEILNEHMER}(x_{Mo}, x_{Ma})) \\
& \longrightarrow x_{Ma}, x_{Mo} \\
b =& \text{STUDENTEN}(x_{Ma}, x_{Na}, x_{Vo}, x_{St}) \wedge \text{TEILNEHMER}(x_{Mo}, x_{Ma}) \\
& \longrightarrow \exists y_{Se}, y_{No} : \text{NOTEN}(x_{Mo}, x_{Ma}, y_{Se}, y_{No}).
\end{aligned}
$$

Algorithmus 2 Standard-CHASE auf Anfragen($\mathcal{B}$, Q)

Require: Menge von Integritätsbedingungen (TGDs, EGDs) $\mathcal{B}$, Anfrage Q_0
Ensure: Modifizierte Anfrage Q_m

```
1: while Q_i ≠ ∅ ∨ ∃ aktive Trigger für ein b ∈ B und Q_i, 0 ≤ i ≤ m do
2:   for all b ∈ B do
3:     for all Trigger g für eine Integritätsbedingung b do
4:       if g ist ein aktiver Trigger then
5:         if b ist eine TGD then
6:           Erweitere g falls nötig, erweitere den Rumpf von Q_i um den Kopf von g(b)
7:         else if b ist eine EGD then
8:           if Verglichene Werte sind verschiedene Konstanten then
9:             Q_m = ∅
10:          else if Verglichene Werte sind Konstante c und Variable a then
11:            Ersetze a durch c
12:          else if Verglichene Werte sind ausgezeichnete Variable x und existenzqu. Variable y then
13:            Ersetze y durch x
14:          else
15:            Ersetzte existenzquantifizierte Variable durch andere existenzqu. Variable
```

Genau wie im Beispiel mit einer Instanz im Abschnitt 2.2.1 suchen wir wieder nach Trigger für die Integritätsbedingungen in $\mathcal{B} = \{b\}$ und dem CHASE-Objekt – dieses mal die Anfrage $Q \equiv Q_0$. Wir wählen wieder als Erstes die Integritätsbedingung b. Die Trigger sind zu finden, indem zwischen dem Rumpf von b und dem Rumpf von Q_0 nach Homomorphismen gesucht wird. Auch hier existiert ein Homomorphismus g_0 mit jeweils einer Abbildung zwischen den STUDENTEN- und TEILNEHMER-Ausdrücken aus dem Anfragerumpf von Q_0 und den STUDENTEN- und TEILNEHMER-Ausdrücken im Rumpf von b ($g_0(x_{Ma}) = x_{Ma},\ g_0(x_{Na}) = y_{Na},\ g_0(x_{Vo}) = y_{Vo},\ g_0(x_{St}) = y_{St},\ g_0(x_{Mo}) = y_{Mo}$). Wieder prüfen wir auf einen aktiven Trigger und suchen nach Erweiterungen von g_0 zu einem Homomorphismus, der den Kopf von b auf den Rumpf von Q_0 abbildet. Da keine zu finden sind, führen wir also den CHASE-Schritt $Q_0 \xrightarrow{b,g_0} Q_1$ durch. Dabei unterscheidet man b wieder nach TGDs und EGDs. Da im Kopf der TGD existenzquantifizierte Variablen vorkommen, wird g_0 um die Abbildung von ihnen auf für die Anfrage neue existenzquantifizierte Variablen erweitert ($g_0(y_{Se}) = y_{Se},\ g_0(y_{No}) = y_{No}$). Bei der Anwendung der TGD im CHASE-Schritt fügen wir den NOTEN-Ausdruck aus dem TGD-Kopf zu Q_0 hinzu. Wieder war g_0 unser einziger Trigger und b die einzige Integritätsbedingung. Also wird kein weiterer CHASE-Schritt mehr ausgeführt und wir erhalten die modifizierte Anfrage:

$$\begin{aligned} Q_1 =& (\exists y_{Na}, y_{Vo}, y_{St}, y_{Se}, y_{No} : \text{STUDENTEN}(x_{Ma}, y_{Na}, y_{Vo}, y_{St}) \\ & \wedge \text{TEILNEHMER}(x_{Mo}, x_{Ma}) \wedge \text{NOTEN}(x_{Mo}, x_{Ma}, y_{Se}, y_{No})) \\ & \longrightarrow x_{Ma}, x_{Mo}. \end{aligned}$$

Genau wie im Beispiel mit der Instanz, gibt es jetzt keinen aktiven Trigger mehr von einer Integritätsbedingung in $\mathcal{B}$ zu Q_1. Wir erhalten also die Anfrage Q_1 als endgültiges Ergebnis des Standard-CHASEs.

In dem nächsten Abschnitt werden wir uns weitere CHASE-Varianten anschauen und vergleichen. Darauf folgt ein Abschnitt zu der *BACKCHASE-Phase*, die jetzt an unser Anfragebeispiel anknüpfen würde, um die Anfrage letztendlich zu optimieren.

2.2.3. CHASE-Varianten

Wenn wir bis jetzt vom CHASE gesprochen haben, dann meinten wir immer den (im vorherigen Abschnitt definierten) Standard-CHASE. Neben diesem gibt es auch weitere CHASE-Arten, welche sich leicht im Ablauf unterscheiden und somit auch unterschiedliche Eigenschaften (wie z. B. in der Terminierung) aufweisen. Der *Oblivious-CHASE* vernachlässigt z. B. die Prüfung auf aktive Trigger. Somit ist seine Anwendung um einiges performanter als der Standard-CHASE. Dafür wendet der Oblivious-CHASE die TGDs und EGDs aber auch dann an, wenn der Kopf einer Integritätsbedingung bereits erfüllt ist. Im Falle einer TGD würde man in ungünstigen Fällen unendlich oft neue Tupel zu einer Instanz oder Ausdrücke zu einer Anfrage hinzufügen. Dabei wird der Oblivious-CHASE noch in zwei Varianten unterschieden. Der *skolemisierte* Oblivious-CHASE kann, im Gegensatz zum *naiven* Oblivious-CHASE, mithilfe einer Skolemfunktion die Anzahl der in Instanzen eingefügten Nullwerte reduzieren. Dabei fallen die Nullwerte in sich zusammen, welche von denselben allquantifizierten Variablen abhängen.

Als Beispiel seien folgende Instanz und TGD gegeben:

$$\begin{aligned} I_0 = \{&\text{STUDENTEN}(3, \text{Mueller}, \text{Max}, \text{Informatik}),\\ &\text{TEILNEHMER}(2, 3)\}\\ b = &\text{STUDENTEN}(x_{Ma}, x_{Na}, x_{Vo}, x_{St}) \wedge \text{TEILNEHMER}(x_{Mo}, x_{Ma})\\ &\longrightarrow \exists y_{Se}, y_{No} : \text{NOTEN}(x_{Mo}, x_{Ma}, y_{Se}, y_{No}). \end{aligned}$$

Die Anwendung des naiven Oblivious-CHASE auf I_0 mit b ergibt:

$$\begin{aligned} I_1 = \{&\text{STUDENTEN}(3, \text{Mueller}, \text{Max}, \text{Informatik}),\\ &\text{TEILNEHMER}(2, 3),\\ &\text{NOTEN}(2, 3, \mu_{Se1}, \mu_{No1})\}. \end{aligned}$$

Da eine Überprüfung auf aktive Trigger (im Gegensatz zum Standard-CHASE) entfällt, wird der naive Oblivious-CHASE erneut angewandt:

$$\begin{aligned} I_2 = \{&\text{STUDENTEN}(3, \text{Mueller}, \text{Max}, \text{Informatik}),\\ &\text{TEILNEHMER}(2, 3),\\ &\text{NOTEN}(2, 3, \mu_{Se1}, \mu_{No1}),\\ &\text{NOTEN}(2, 3, \mu_{Se2}, \mu_{No2})\}. \end{aligned}$$

Der naive Oblivious-CHASE würde in diesem Beispiel nicht terminieren und die TGD b immer wieder auf I_i anwenden. Die Instanz würde in diesem Fall unendlich groß werden.

Der Oblivious- sowie der Standard-CHASE haben das Problem, dass die Reihenfolge der eingechaseten Abhängigkeiten nicht-deterministisch ist. Somit ist die Terminierung des Algorithmus für ein gegebenes CHASE-Objekt und eine Menge von Integritätsbedingungen (TGDs und EGDs) unentscheidbar (bei der Einschränkung der Integritätsbedingungen auf FDs und JDs bleibt die Terminierung entscheidbar [MMS79]). Aus diesem Grund sind Bedingungen unter denen der CHASE terminiert immer noch Stand der Forschung. Ebenso aus diesem Grund wurde der *Core-CHASE* in [DNR08] eingeführt. Mit zwei Zwischenschritten wird für den CHASE eine universelle Lösung[2] bestimmt, falls diese existiert. In dem sogenannten parallelen CHASE-Schritt werden als erstes alle aktiven Trigger parallel angewendet. Im zweiten Schritt wird dann der sogenannte Kern berechnet. Dabei entfallen alle Lösungen aus dem parallelen CHASE-Schritt, die von anderen Lösungen überdeckt werden. Da dieser Kern bis auf Isomorphie für

[2] bis auf Homomorphismen eindeutig [FKMP03]

ein CHASE-Objekt einzigartig ist, gilt der Core-CHASE als deterministisch. Diese Eigenschaft kommt aber mit einer so hohen Komplexität daher, dass der Core-CHASE in der Regel nicht in Anwendungsprogrammen eingesetzt wird.

Zusammengefasst terminiert der Core-CHASE genau dann, wenn es eine universelle Lösung gibt. Dies ist für den Standard-CHASE nicht immer der Fall. Für ihn ist es wahrscheinlicher, dass eine nichtdeterministische Folge von CHASE-Schritten terminiert, als dass alle Folgen von CHASE-Schritten terminieren. Noch seltener terminiert der Oblivious-CHASE, da der Test auf aktive Trigger entfällt. Der *naive* Oblivious-CHASE terminiert dabei für eine noch geringere Anzahl von Fällen als der *skolemisierte* Oblivious-CHASE, der in manchen Fällen die Anzahl der eingefügten Tupel reduziert.

Für genaue Definitionen und Beispiele zu den CHASE-Varianten wird hier auf [DHI12, GMS12] und auf das Lehrvideo von Andreas Görres und Yvonne Düwel aus dem Modul "Neueste Entwicklungen in der Informatik" (vorhanden am DBIS Lehrstuhl der Uni Rostock) verwiesen.

2.2.4. BACKCHASE-Phase

Wenn bisher vom CHASE auf Anfragen die Rede war, wurde ein wichtiger Teil davon bis jetzt ausgelassen. Schon in der Einleitung ist ein Beispiel aufgeführt worden, in dem wir eine Anfrage optimiert haben. Dabei wurde die SQL-Anfrage:

```
SELECT Matrikelnr, Modulnr
FROM STUDENTEN NATURAL JOIN NOTEN
```

mit der Bedingung, dass alle Studenten aus STUDENTEN, die eine Note in NOTEN haben, auch als Teilnehmer des entsprechenden Moduls in TEILNEHMER auftauchen müssen, in eine Anfrage ohne `NATURAL JOIN` umformuliert:

```
SELECT Matrikelnr, Modulnr
FROM TEILNEHMER.
```

Stellen wir die Anfrage und die Integritätsbedingung jetzt in unserer Schreibweise da:

$$\begin{aligned} Q =& (\exists y_{Na}, y_{Vo}, y_{St}, y_{Se}, y_{No} : \text{STUDENTEN}(x_{Ma}, y_{Na}, y_{Vo}, y_{St}) \wedge \text{NOTEN}(x_{Mo}, x_{Ma}, y_{Se}, y_{No})) \\ & \longrightarrow x_{Ma}, x_{Mo} \\ b =& \text{STUDENTEN}(x_{Ma}, x_{Na}, x_{Vo}, x_{St}) \wedge \text{NOTEN}(x_{Mo}, x_{Ma}, x_{Se}, x_{No}) \\ & \longrightarrow \text{TEILNEHMER}(x_{Mo}, x_{Ma}). \end{aligned}$$

Wenn wir jetzt den zuvor definierten CHASE auf b und Q anwenden, dann erweitern wir den Anfragerumpf mit dem TEILNEHMER-Ausdruck aus dem Kopf der TGD. Das Ergebnis von CHASE$(\{b\}, Q)$[3] entspricht also der Anfrage:

$$\begin{aligned} Q_1 =& (\exists y_{Na}, y_{Vo}, y_{St}, y_{Se}, y_{No} : \text{STUDENTEN}(x_{Ma}, y_{Na}, y_{Vo}, y_{St}) \\ & \wedge \text{NOTEN}(x_{Mo}, x_{Ma}, y_{Se}, y_{No}) \wedge \text{TEILNEHMER}(x_{Mo}, x_{Ma})) \\ & \longrightarrow x_{Ma}, x_{Mo} \end{aligned}$$

[3] Schreibweise für den CHASE nach Algorithmus Standard-CHASE auf Anfragen($\mathcal{B}$, Q) entspricht dem Einchasen der Menge an Integritätsbedingungen $\{b\}$ in die Anfrage Q

Unsere Anfrage wurde also nicht wie in der Einleitung optimiert, sondern um ein zusätzlichen `JOIN` erweitert. Dieses Ergebnis ist auch beabsichtigt und wird als *Aufblähen* einer Anfrage bezeichnet. Um jetzt eine optimierte Anfrage zu erhalten, müssen wir aus dieser aufgeblähten Anfrage noch eine *minimale äquivalente Teilanfrage* extrahieren. Genau dafür werden wir die sogenannte *BACKCHASE-Phase* nutzen.

Die BACKCHASE-Phase [DPT06] führt als erstes alle minimalen Teilanfragen SQ von Q auf, die äquivalent zu Q sind. Die Teilanfragen erhalten wir, indem Ausdrücke aus dem Rumpf der ursprünglichen Anfrage gestrichen werden, mit der Bedingung, dass die Variablen im Anfragekopf weiterhin in der neuen Teilanfrage enthalten bleiben. Zwei Anfragen gelten dann als äquivalent, wenn sie angewandt auf eine Instanz (die die Integritätsbedingungen erfüllt) die gleichen Ergebnisse liefern. Als Ergebnis des BACKCHASEs wird dann die Teilanfrage mit den wenigsten Konjunktionen im Rumpf ausgewählt.

In unserem Beispiel erhalten wir also die Teilanfragen:

$$
\begin{aligned}
SQ_1 =& (\exists y_{Na}, y_{Vo}, y_{St}, y_{Se}, y_{No} : \text{STUDENTEN}(x_{Ma}, y_{Na}, y_{Vo}, y_{St}) \\
& \wedge \text{NOTEN}(x_{Mo}, x_{Ma}, y_{Se}, y_{No}) \wedge \text{TEILNEHMER}(x_{Mo}, x_{Ma})) \\
& \longrightarrow x_{Ma}, x_{Mo} \\
SQ_2 =& (\exists y_{Na}, y_{Vo}, y_{St}, y_{Se}, y_{No} : \text{STUDENTEN}(x_{Ma}, y_{Na}, y_{Vo}, y_{St}) \wedge \text{NOTEN}(x_{Mo}, x_{Ma}, y_{Se}, y_{No})) \\
& \longrightarrow x_{Ma}, x_{Mo} \\
SQ_3 =& (\exists y_{Se}, y_{No} : \wedge\text{NOTEN}(x_{Mo}, x_{Ma}, y_{Se}, y_{No})) \longrightarrow x_{Ma}, x_{Mo} \\
SQ_4 =& \text{TEILNEHMER}(x_{Mo}, x_{Ma}) \longrightarrow x_{Ma}, x_{Mo},
\end{aligned}
$$

wobei SQ_3 nicht äquivalent zu Q ist (SQ_3 erfüllt nicht die Integritätsbedingungen und würde nicht das gleiche Ergebnis wie Q liefern). Als Ergebnis der BACKCHASE-Phase würden wir also SQ_4 erhalten, da sie am wenigsten Konjunktionen im Rumpf enthält. Dieses Ergebnis deckt sich auch mit dem aus der Einleitung, indem wir in der `FROM`-Klausel STUDENTEN `JOIN` NOTEN durch TEILNEHMER ersetzt haben.

Weitere Anwendungsfälle des BACKCHASEs wären z. B. der Fall II aus der Tabelle 1.1, Answering Queries Using Views. Dabei werden Anfragen zuerst mit der CHASE-Phase um Sichten erweitert, um dann mit der BACKCHASE-Phase die minimal benötigten Sichten zu berechnen. Auch hier wird auf ein Lehrvideo aus dem Modul “Neueste Entwicklungen in der Informatik” von Florian Rose und Jacques Toussaint verwiesen, indem sie AQuV anhand von verständlichen Beispielen erklären (vorhanden am DBIS Lehrstuhl der Uni Rostock).

Ein weiteres Anwendungsbeispiel für den BACKCHASE ist das Provenance Management, welches den Fall VI in der Tabelle 1.1 widerspiegelt. Hier wird zuerst der CHASE für die Auswertung von Anfragen (d. h. Datentransformation), die Schema- und Datenentwicklung und den Datenaustausch angewandt. Danach wird dann der BACKCHASE angewandt, um die Anfrageauswertung zu invertieren, d. h. um eine erweiterte Antwort der *why-Provenance* als Zeugenbasis der Anfrage zu berechnen [AH19].

In dieser Arbeit werden wir uns auf die CHASE-Phase beschränken, d. h. mit dem Aufblähen (einchasen von Integritätsbedingungen) einer Anfrage. Die BACKCHASE-Phase ist aktueller Bestandteil einer eigenen Masterarbeit am DBIS Lehrstuhl der Uni Rostock. Für eine genaue Definition und weitere Beispiele wird hier auf das Paper [DPT06] verwiesen.

2.3. Einordnung der Grundlagen in diese Arbeit

Den Grundlagen wurde in dieser Arbeit viel Zeit gewidmet. Das war wichtig, um die CHASE-Objekte sowie die CHASE-Schritte auf den CHASE-Objekten anhand der einheitlichen Definitionen im Konzept (siehe Kapitel 4) zu vereinheitlichen. Dort werden wir dann auch die in den Abschnitten 2.2.1 CHASE auf Instanzen und 2.2.1 CHASE auf Anfragen vorgestellten ähnlichen Algorithmen zusammenfassen können. Dafür werden wir nicht nur die Unterschiede der CHASE-Objekte behandeln müssen, sondern auch die Unterschiede in den Homomorphismen zwischen Integritätsbedingungen und den Objekten (Trigger) sowie zwischen den Objekten $\mathcal{O}_i$ und $\mathcal{O}_{i+1}$ nach einem CHASE-Schritt $\mathcal{O}_i \xrightarrow{b_i, g_i} \mathcal{O}_{i+1}$. Am Ende soll ein und derselbe Algorithmus sowohl auf Instanzen als auch auf Anfragen angewandt werden können, um Integritätsbedingungen in die jeweiligen CHASE-Objekte einzuchasen.

Als CHASE-Variante wählen wir den Standard-CHASE. Der Oblivious-CHASE würde zwar die Implementierung erleichtern und eine kürzere Laufzeit mit sich bringen, terminiert in der Praxis aber in zu wenigen Fällen. Der Core-CHASE wiederum könnte zwar auch bei den Fällen terminieren, bei denen der Standard-CHASE kein Ergebnis liefert, wäre aber mit einer zu hohen Komplexität in der Implementierung und längeren Laufzeiten verbunden. In ChaTEAU ist der Standard-CHASE bereits einzeln für Instanzen und Anfragen implementierten. Für die Vereinheitlichung der CHASE-Methoden in ChaTEAU bietet der Standard-CHASE also einen guten Ausgleich zwischen Komplexität und Terminierungseigenschaften und kann an bestehende Implementierungen anknüpfen.

Bei der Vereinheitlichung werden wir uns auf die CHASE-Phase beschränken. Die BACKCHASE-Phase läuft für verschiedene CHASE-Objekte zu unterschiedlich ab, als dass sie mit dem gleichen Konzept vereinheitlicht werden kann. Die ganze BACKCHASE-Phase bietet genug Komplexität für eine eigenständige Arbeit am DBIS Lehrstuhl der Uni Rostock.

In dem nächsten Kapitel werden wir den Stand der Technik begutachten. Dabei werden wir uns die bisherige Implementierung von ChaTEAU und weitere CHASE-Tools genauer anschauen. Im darauf folgenden Konzept-Kapitel wird dann vorgestellt, wie die Standard-CHASE-Phase auf Instanzen und Anfragen vereinheitlicht werden kann.

3. Stand der Technik

Neben dem aktuellen Stand von ChaTEAU werden wir uns in diesem Abschnitt die Tools anschauen, die im Sommersemester 2019 Untersuchungsgegenstand des Moduls "Komplexe Softwaresysteme" waren. Diese Tools implementieren die gängigsten Umsetzungen des CHASE-Algorithmus. Im Gegensatz zu ChaTEAU wird der CHASE aber jeweils nur auf einem CHASE-Objekt sinnvoll unterstützt.

Die Anfrageoptimierung von `Graal` wurde von Jasper Roloff untersucht (siehe Abschnitt 3.2). Das einchasen von Integritätsbedingungen in eine Instanz mithilfe von `Graal` war Gegenstand meiner eigenen Untersuchung. Willi Brekenfelder und Sebastian Rutofski haben sich mit dem Anfragen-Optimierer `PDQ` beschäftigt (siehe Abschnitt 3.3). Die Funktionalität des letzten Tools in diesem Abschnitt – `Llunatic` – wurde von Ruven Kronenberg und Sergej Schimanski beschrieben (siehe Abschnitt 3.4).

3.1. ChaTEAU

Was ist ChaTEAU? ChaTEAU (*Chase for Transforming, Evolving, and Adapting databases and queries, Universal approach*) ist ein Java-Programm, welches Integritätsbedingungen in Form von TGDs und EGDs in Instanzen und Anfragen einchasen kann. Der implementierte CHASE-Algorithmus orientiert sich sehr stark an der Theorie aus [BKM+17] und ist im groben in den Algorithmen 1 und 2 dargestellt.

Die Theorienähe erkennt man auch am inneren Aufbau von ChaTEAU. Die Grundlage für alle Konstrukte bildet die Klasse `Term`, in der Konstanten, Variablen und Nullwerte gespeichert werden können. Konstanten können vom Typ `integer, double` oder `string` sein und werden dynamisch einer passenden Objekt-Variablen zugewiesen. Nullwerte und Variablen werden in einem Term als eigenes Objekt gespeichert. Unterscheidbar bleiben die verschiedenen Termtypen durch die Enumeration `TermType` als Objekt-Variable. Atome bilden die nächst höhere Abstraktionsebene. Dabei wird zwischen relationalen und Gleichheitsatomen unterschieden. `RelationalAtom` kann eine beliebige Menge von Termen beinhalten, während `EqualityAtom` nur zwei Terme beinhaltet, zwischen denen eine Gleichheitsrelation gelten soll. Die Integritätsbedingungen (`IntegrityConstraint`) bestehen aus einem Kopf (`head`) und einem Rumpf (`body`), welche aus einer Menge von relationalen Atomen bestehen. Die Ausnahme bildet hier der EGD-Kopf, welcher aus einer Menge von Gleichheitsatomen besteht. Anfragen werden von TGDs abgeleitet und bestehen so ebenfalls aus relationalen Atomen. Instanzen (`Instance`) bestehen aus einer Beschreibung des Schemas, einer Menge der möglichen Attribute und aus Tupeln, wobei die Tupel ebenfalls nur aus relationalen Atomen bestehen. Abbildungen zwischen den Termen werden mit Homomorphismen beschrieben. Die Klasse `Homomorphism` speichert für jede Abbildung zwischen Termen ein `TermMapping`-Objekt und ist ausschlaggebend für die Suche nach und das Anwenden von aktiven Triggern, zwischen den Integritätsbedingungen und dem CHASE-Objekt. Abbildung 3.1 verdeutlicht beispielhaft den Aufbau einer Instanz aus Termen.

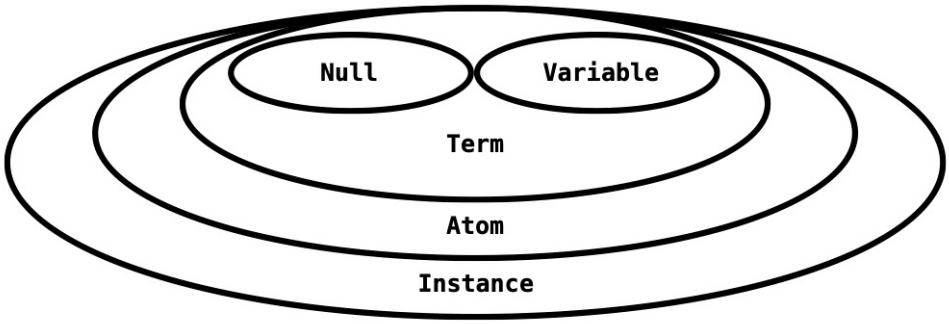

Abbildung 3.1.: Grober Aufbau einer Instanz in ChaTEAU

Was kann ChaTEAU und was kann es nicht? ChaTEAU wurde in zwei Phasen entwickelt. In der Ersten implementierte Martin Jurklies im Rahmen seiner Masterarbeit [Jur18] alle wichtigen Grundlagen, um den CHASE auf Instanzen anzuwenden. In der zweiten Phase fügte Fabian Renn im Rahmen seiner Bachelorarbeit [Ren19] eine geeignete Eingabe für die CHASE-Objekte und Parameter mittels XML-Dateien hinzu. Neben der Eingabe wurde auch die Ausgabe des CHASE-Ergebnisses als XML- sowie CSV-Datei hinzugefügt. Zusätzlich wird am Ende des Programms eine Log-Datei ausgegeben, die alle protokollierten Schritte des CHASEs enthält. Zum Schluss wurde der CHASE teilweise auf Anfragen umgesetzt, was wegen des knappen Zeitrahmens der Arbeit nicht fertiggestellt werden konnte.

Im Stand von ChaTEAU nach den beiden Arbeiten ist das einchasen von TGDs in Instanzen ohne Nullwerte korrekt umgesetzt. EGDs können Instanzen nicht verändern und führen bei Unstimmigkeiten im EGD-Kopf lediglich zum Abbruch des CHASEs und zur Ausgabe von fehlerhaften Konstanten. Der CHASE auf Anfragen funktioniert noch nicht sinnvoll, da Trigger zwischen den Variablen der Integritätsbedingungen und denen der Anfrage nicht erkannt werden, wenn die Variablen nicht zufälligerweise exakt gleich benannt sind. Genauso wenig werden Gleichheiten in Anfragen beim einchasen von EGDs erkannt. Die Eingabe von Instanzen und Anfragen funktioniert, hat aber noch ein paar kleine Probleme, die in Abschnitt 5.3 gelöst werden.

In den Beispielen für die Eingabe- und der Ausgabe-Dateien von ChaTEAU (siehe Listing 3.1 und 3.2) wurden die Definition des Schemas aus der Ein- und Ausgabe entfernt. Die Ausgabe wird in einem `input`-Tag geschrieben, um sie wieder als Eingabe weiterer CHASE-Phase zu nutzen. Außerdem wurde die Ausgabe auf das neue NOTEN-Tupel reduziert. Nullwerte werden in ChaTEAU mit dem Präfix `#N` definiert. Existenzquantifizierte Variablen haben den Präfix `#E` und ausgezeichnete Variablen den Präfix `#V`. Die Log-Datei, die während der Ausführung in ChaTEAU entsteht, ist im Anhang A zu finden. Äquivalent zum Grundlagen-Kapitel 2, würde die Ein- und Ausgabe formal wie folgt definiert werden:

$$\begin{aligned} I_{Eingabe} = \{&\text{STUDENTEN}(3, \text{Mueller}, \text{Max}, \text{Informatik}),\\ &\text{TEILNEHMER}(2, 3)\} \end{aligned}$$

$$\begin{aligned} b_{Eingabe} = &\text{STUDENTEN}(x_{Ma}, x_{Na}, x_{Vo}, x_{St}) \wedge \text{TEILNEHMER}(x_{Mo}, x_{Ma})\\ &\longrightarrow \exists y_{Se}, y_{No} : \text{NOTEN}(x_{Mo}, x_{Ma}, y_{Se}, y_{No}) \end{aligned}$$

$$\begin{aligned} I_{Ausgabe} = \{&\text{STUDENTEN}(3, \text{Mueller}, \text{Max}, \text{Informatik}),\\ &\text{TEILNEHMER}(2, 3),\\ &\text{NOTEN}(2, 3, \mu_{Se1}, \mu_{No1})\}. \end{aligned}$$

Listing 3.1: Beispiel ChaTEAU – Eingabe beim einchasen einer TGD in einer Instanz

```
<input>
    <schema>
        <relations\>
        <dependencies>
            <tgd>
                <body>
                    <atom name="Studenten">
                        <variable name="#V_matrikelnummer_1" />
                        <variable name="#V_nachname_1" />
                        <variable name="#V_vorname_1" />
                        <variable name="#V_studiengang_1" />
                    </atom>
                    <atom name="Teilnehmer">
                        <variable name="#V_modulnummer_1" />
                        <variable name="#V_matrikelnummer_1" />
                    </atom>
                </body>
                <head>
                    <atom name="Noten">
                        <variable name="#V_modulnummer_1" />
                        <variable name="#V_matrikelnummer_1" />
                        <variable name="#E_semester_1" />
                        <variable name="#E_note_1" />
                    </atom>
                </head>
            </tgd>
        </dependencies>
    </schema>
    <instance>
        <relationalAtom name="Studenten">
            <tuple id="Studenten_1">
                <constant name="3" />
                <constant name="Mueller" />
                <constant name="Max" />
                <constant name="Informatik" />
            </tuple>
        </relationalAtom>
        <relationalAtom name="Teilnehmer">
            <tuple id="Teilnehmer_1">
                <constant name="2" />
                <constant name="3" />
            </tuple>
        </relationalAtom>
    </instance>
</input>
```

Listing 3.2: Beispiel ChaTEAU – Ausgabe beim einchasen einer TGD in einer Instanz

```
<input>
    <schema>
        <relations/>
        <dependencies/>
    </schema>
    <instance>
        <relationalAtom name="Noten">
            <tuple>
                <constant name="2" />
                <constant name="3" />
                <constant name="#N_semester_1" />
                <constant name="#N_note_1" />
            </tuple>
        </relationalAtom>
    </instance>
</input>
```

3.2. Graal

Was ist Graal? `Graal` wird auf der eigenen Homepage [Gra19] als "ein Java-Toolkit zur Anfrage von Wissensdatenbanken im Rahmen existenzieller Regeln, auch bekannt als Datalog" beschrieben. Liest man sich die Hauptmerkmale von `Graal` durch, die softwareseitig als einzelne Module designt wurden, dann findet man unter anderem folgende Bestandteile:

- eine Datenschicht, die generische Schnittstellen bereitstellt, um verschiedene Arten von Daten zu speichern und sie mit (verbundenen) konjunktiven Anfragen abzufragen;
- eine ontologische Schicht, wobei eine Ontologie eine Reihe von existenziellen Regeln ist;
- eine Wissensbasisschicht, wobei eine Wissensbasis aus einer Faktenbasis (Abstraktion der Daten aus der Datenschicht) und einer Ontologie besteht;
- einen Regelanalysator, der eine syntaktische und strukturelle Analyse eines existenziellen Regelsatzes durchführt;
  er kann verwendet werden, um zu überprüfen, ob der Regelsatz zu einer bekannten entscheidungsfähigen Klasse gehört (auch indem er diesen Satz in Teilmengen unterteilt, die nach verschiedenen Paradigmen verarbeitet werden können, während die Vollständigkeit der Anfragebeantwortung erhalten bleibt);
- Hilfswerkzeuge zur Zerlegung und Kennzeichnung von Regeln;
- Algorithmen zur Verarbeitung von Ontologie-behafteten Anfragen:
  - Algorithmen zum Umschreiben von Anfragen, die den Kern der `Graal`-Techniken bilden. Diese Algorithmen verwenden die Regeln, um die Eingabeanfrage in eine minimale Menge (d. h. Vereinigung) von konjunktiven Anfragen umzuschreiben. Um die Antworten auf eine Anfrage auf einer Wissensdatenbank zu berechnen, kann man die umgeschriebene Anfrage auf der Faktenbasis bewerten.
  - Vorwärtsverkettungsalgorithmen (CHASE-Algorithmen), die Regeln auf die Faktenbasis bis zur Sättigung[1] anwenden. Um die Antworten auf eine Anfrage auf einer Wissensdatenbank zu berechnen, kann man diese Anfrage auf der gesättigten Faktenbasis auswerten. Diese Algorithmen implementieren sogenannte eingeschränkte (oder Standard) CHASE- und Frontier-Restricted CHASE-Algorithmen (die sich wie der Skolem CHASE verhalten).

Was kann Graal und was kann es nicht? Die Anfrageoptimierung in `Graal` ist eher als ein Nebeneffekt des "Query Rewritings" zu sehen, bei dem Anfragen unter Beachtung von Integritätsbedingungen umgeschrieben und so vereinfacht werden. Der Query-Rewriter in `Graal` wird "Pure" genannt und ist wie folgt definiert: "Pure is a query rewriter: given a set of rules R and a conjunctive query q, it rewrites q using the rules in R such that, for any data set D, the answers to q over the knowledge base (R,D) can be computed by evaluating the rewritten query directly over D " [Gra19]. In den Tests konnte `Graal` nur drei[2] von sechs getesteten Benchmark-Anfragen (mehr dazu in Abschnitt 5.4) optimieren. Auch bei den optimierten Anfragen wurde nicht erkannt, wenn die gleiche Relation bereits in der Anfrage enthalten war. Ebenso kam es zu Ersetzungen von gleichgesetzten Variablen durch Relationen mit Ausdrücken, die keinerlei Beziehung zur Anfrage hatten.

[1] bis das Anwenden der Regeln die Faktenbasis nicht mehr verändert
[2] Optimiert wurden A1, B1, B2. Nicht optimiert wurden A2, A3, B3.

Als Fazit der Untersuchungen bleibt zu sagen, dass `Graal` (gemäß der Beschreibung) Anfragen nur umschreiben kann. Die Optimierung der Anfragen ist aber eher als Seiteneffekt zu betrachten. In wieweit der Query-Rewriter den CHASE nutzt, konnte im Rahmen der Untersuchungen nicht geklärt werden.

Bei der Untersuchung des CHASE-Algorithmus auf Instanzen mithilfe von `Graal`, wurde zuerst versucht, eine SQLite-Datenbank an die Datenschicht-Schnittstelle von `Graal` zu übergeben. Diese Herangehensweise traf schnell auf Probleme, da `Graal` bei übergebenen Nullwerten in der Instanz mit einer Java-Exception antwortet. Instanzen ohne Nullwerte lassen sich verarbeiten. Es ist sogar möglich mithilfe einer s-t TGD neue Relationen zu erzeugen. Von diesen kann man sich aber nur die Attribute ausgeben lassen, die auch im Quellschema vorhanden sind. Sobald man Attribute miteinbezieht, die aus den existenzquantifizierten Variablen des Zielschemas der s-t TGD entstanden sind, bleibt die Ausgabe leer (die Attribute müssten eigentlich neue Nullwerte enthalten). `Graal` unterstützt leider auch keine konjunktiven Anfragen an die Datenbank, die einen Verbund von Relationen beinhalten. Diese Probleme führen zu der Annahme, dass die Datenschicht-Schnittstelle von `Graal` noch nicht ausgereift genug ist. Als zweiten Ansatz für die Untersuchungen des CHASE wurde die Datenbank als Datalog-Datenbasis an `Graal` übergeben (siehe Beispiel am Ende dieses Abschnitts). Hier hat man den Vorteil, Nullwerte mithilfe von Variablen in den Fakten zu simulieren. Auch das Problem mit den konjunktiven Anfragen tritt bei diesem Ansatz nicht mehr auf. Nach dem Erfolg mit den konjunktiven Anfragen kann man denken, dass der Fehler gefunden und behoben wurde und der implementierte CHASE-Algorithmus in `Graal` jetzt mittels verschiedener TGDs und EGDs getestet werden kann. Es stellt sich heraus, dass das doch nicht so einfach ist. Neue Tupel, die mithilfe einer TGD theoretisch hinzugefügt werden sollen, tauchten in der Ausgabe leider immer noch nicht auf. Trotz des Misserfolges beim Testen der TGDs, galt es noch zu untersuchen, ob `Graal` mit EGDs arbeiten kann (so wollten wir auch testen, ob die Variablen in einem Tupel wirklich als Nullwerte benutzt werden können). Doch anstatt zwei Tupel gleichzusetzen (wie wir es nach unserer Konstruktion erwarten), erhalten wir auf einmal eine Ausgabe, in der die TGD vom vorangegangenen Test angewandt werden (siehe Beispiel am Ende dieses Abschnitts). Die EGD hat also in gewisser Weise die TGD aktiviert, ansonsten das Ergebnis aber nicht beeinflusst. Hier fällt ebenfalls auf, dass `Graal` Nullwerte intern mit einem speziellen Präfix versieht. Somit ist klar, dass unsere Verwendung der Variablen in einem Tupel nicht den gewünschten Effekt hat, die Nullwerte zu simulieren. Beim Versuch, die Variablen in den Fakten dementsprechend mit dem richtigen Präfix zu benennen, wird diese intern trotzdem wie jede andere Variable gehandhabt. Interessant ist, dass auch eine EGD auf eine durch eine zweite TGD erzeugte Relation die TGD aktiviert.

Im folgenden `Graal`-Beispiel wir die Eingabe von TGDs, EGDs und einer Instanz als Fakten in Datalog-Schreibweise gezeigt. Die Ausgabe der veränderten Instanz wird über einer Anfrage an diese realisiert. Im Beispiel ist zu erkennen, wie die EGD (auf der durch die zweite TGD erzeugten Fachschaft-Relation) die erste TGD aktiviert:

`@rules` %= (TGD_1, TGD_2, EGD)
student(Ma,N,V,S,I) :- noten(Ma,Mo,No).
fachschaft(S,I,F) :- student(M,N,V,S,I).
F1=F2 :- fachschaft(S1,I1,F1), fachschaft(S2,I2,F2),
S1=S2.
`@facts`
student(7, mustermann, max, e-technik, ief).
student(7, musterfrau, max, e-technik, ief).
student(8, johannes,paul,itti,ief).
student(8, X,paul,itti,ief).
student(8, Y,paul,itti,ief).
noten(10, 0003, 2.7).
noten(11, 0003, 2.7).

= `Query` =
?(M,N) :- student(M, N, V, S, I).
= `Answers` =
{M->"11",N->EE1}
{M->"8",N->I0}
{M->"7",N->musterfrau}
{M->"8",N->I1}
{M->"7",N->mustermann}
{M->"10",N->EE5}
{M->"8",N->johannes}.

TGDs werden von `Graal` also nur berücksichtigt, wenn man dazu eine EGD schreibt (reihenfolgeunabhängig), die sich auf die Relation (oder auf eine aus der Relation durch eine zweite TGD entstandenen Relation) aus dem TGD-Kopf bezieht. EGDs können mit `Graal` nicht angewandt werden.

Mir ist es, trotz mehreren weiteren Versuchen und lesen von verschiedenen Begleit-Artikeln, nicht gelungen, den CHASE auf Instanzen in `Graal` sinnvoll zu untersuchen. Das erste Problem (dass die Schnittstelle der Datenschicht die übergebenen Datenbanken nicht korrekt in das Datalogformat überführen kann) kann mit der Umwandlung der Datenbank in eine Datalog-Datenbasis umgangen werden. Aber die Probleme im zweiten Test zeigen, dass `Graal` noch nicht ausgereift genug für die Anwendung des CHASE-Algorithmus auf Instanzen unter Beachtung von TGDs und EGDs ist.

3.3. PDQ

Was ist PDQ? Mit `PDQ` lassen sich Anfragen unter Berücksichtigung von Abhängigkeiten, Kosten und Beschränkungen semantisch optimieren. Dabei sucht das Tool nicht nach einem Anfrageplan, sondern versucht die Beantwortbarkeit der Anfrage zu beweisen. Die Anfragepläne werden dann aus den gefundenen Beweisen konstruiert. Da die abgeleiteten Pläne im Laufe der Erkundung des Beweisraumes verglichen werden und direkt der effizienteste Plan abgeleitet werden kann, benötigt PQD keine eigene Optimierungsphase, um den besten Anfrageplan zu bestimmen [BCT14].

Was kann PDQ und was kann es nicht? Von den getesteten sechs Benchmark-Anfragen (mehr dazu in Abschnitt 5.4) konnte `PDQ` fünf[3] erwartungsgemäß optimieren. Darunter konnten unter anderem Verbundtreue-Kriterien mithilfe von FDs oder JDs angewandt werden und Schlüssel-Fremdschlüssel-Verbunde eingespart oder Ketten von ihnen aufgelöst werden (sogar wenn der Schlüssel in der Kette wechselte). Einzig eine Anfrage mit einer Mehrattribut-Fremdschlüssel-Bedingung mit einer internen FD führte in `PDQ` zu einer Exception und zur Rückgabe einer leeren Anfrage. Des Weiteren ist ein gewisser Nichtdeterminismus in der Anfrageoptimierung aufgefallen. Manche Anfragen wurden in einem Durchlauf optimiert und in dem nächsten nicht mehr.

Im folgenden `PDQ`-Beispiel (aus dem Bericht von Willi Brekenfelder und Sebastian Rutofski) wird die Optimierung einer Anfrage mit einer Kette von Schlüssel-Fremdschlüssel-Verbund-Bedingungen mit wechselnden Schlüsseln gezeigt (entspricht Benchmark-Anfrage B2, siehe Abschnitt 5.4). Die Ein- und Ausgabe wurde kompakt als relationenalgebraische Ausdrücke geschrieben:

`Schema`:
UNIANGEHOERIGE = {Matrikelnr, Nachname}
STUDENTEN = {Matrikelnr, Studiensemester}
IEF-MITGLIEDER = {Matrikelnr, IEF-Nr}
IEF-BALLSPORTLER = {IEF-Nr, Sportart}
IEF-FAUSTBALLER = {IEF-Nr, Position}
`Bedingung`:
IEF-Faustballer (IEF-Nr)
$\subseteq$ IEF-BALLSPORTLER (IEF-Nr)
$\subseteq$ IEF-MITGLIEDER (IEF-Nr),
IEF-Mitglieder (IEF-Nr)
$\subseteq$ STUDENTEN(Matrikelnr)
$\subseteq$ UNIANGEHOERIGE(Matrikelnr)

`Anfrage`:
$\pi_{\text{Matrikelnr ,Nachname}}$ (
$\sigma_{\text{Semester }=10}$(
r(UNIANGEHOERIGE)
$\bowtie$ r(STUDENTEN)
$\bowtie$ r(IEF-MITGLIEDER)
$\bowtie$ r(IEF-BALLSPORTLER)
$\bowtie$ r(IEF-FAUSTBALLER)))
`Optimierte Anfrage`:
$\pi_{\text{Matrikelnr ,Nachname}}$(
r(UNIANGEHOERIGE)
$\bowtie$ r(IEF-MITGLIEDER)
$\bowtie$ r(IEF-FAUSTBALLER)).

[3] Optimiert wurden A1, A2, A3, B1, B2. Nicht optimiert wurde B3.

Zu bemerken ist hier, dass die eigentliche Ein- und Ausgabe von `PDQ` als Vorbild für ChaTEAU diente. In beiden Tools werden die XML-Dateien mit unter anderem dem Datenbankschema, Integritätsbedingungen und Anfragen eingelesen und dann das Ergebnis als XML-Datei ausgegeben.

3.4. Llunatic

Was ist Llunatic? `Llunatic` ist ein allgemeines Framework fürs Cleaning von Daten und Mapping von Schemata. Es implementiert den CHASE-Algorithmus, sodass TGDs und EGDs verarbeitet werden können. Das Tool bietet drei Anwendungsmöglichkeiten [Llu19]:

a) traditionelle Datenbereinigung und Datenreparatur, basierend auf Cleaning mittels EGDs;

b) traditioneller Datenaustausch (Mapping) mit s-t TGDs, TGDs und EGDs;

c) eine neue Form von Mapping und Bereinigung, in denen Daten von den Quellen zum Ziel bewegt werden und das Ziel gemäß den Bereinigungsbedingungen repariert wird.

Was kann Llunatic und was kann es nicht? Daten einer Target-Tabelle können mittels EGDs gecleant werden. Dabei kann man auch Daten einer Source-Tabelle mit einbeziehen. Beim Cleaning können sogenannte Lluns entstehen, welche als Stellvertreter für Konstanten dienen und vom Benutzer selbst ersetzt werden müssen. Des Weiteren werden beim Cleaning Konstanten durch andere Konstanten und Nullwerte durch Konstanten ersetzt. `Llunatic` unterscheidet dabei zwischen dem *forward* und dem *backward Repairing*. Forward Repairing repariert Unstimmigkeiten in dem Kopf einer EGD, da davon ausgegangen wird, dass der Rumpf korrekterweise erfüllt ist. Backward Repairing hingegen geht davon aus, dass der Rumpf der EGD nicht korrekterweise erfüllt ist, und repariert diesen, sodass der Kopf der EGD erhalten bleibt. Beim Mapping ist es mit `Llunatic` möglich, mittels s-t TGDs einzelne oder mehrere Source-Tabellen auf einzelne oder mehrere Target-Tabellen abzubilden. Nullwerte, die für noch nicht existierende Attribute angelegt werden, werden in einer Skolemfunktion in Abhängigkeit der bereits existierenden Attribute der Instanz gespeichert. Die verschiedenen Lösungen, die in `Llunatic` beim einchasen von den Integritätsbedingungen in die Instanzen entstehen, werden in einem sogenannten CHASE-Tree gespeichert. Falls die Terminierung wegen zu vieler Ergebnisse zu lange dauern würde, kann man den Ergebnisraum mithilfe verschiedener Cost Manager einschränken und so den CHASE-Tree verkürzen:

- Der Standard Cost Manager gibt alle Lösungen aus.
- Der Maximum Size Cost Manager schränkt die Generierung der Lösungen pro CHASE-Tree-Ebene auf ein Maximum ein.
- Beim Forward-Only Cost Manager wird auf ein backward Repairing zur Generierung von Lösungen verzichtet.
- Der Sampling Cost Manager führt zufällig verschiedene Repairing-Varianten durch bis eine vorgegebene Anzahl an Lösungen generiert wurden oder der CHASE terminiert.
- Beim Certain-Region Cost Manager wird eine Region angegeben, von der ausgegangen wird, dass die Daten in ihr korrekt sind. Alle Repairing-Ansätze werden so gewählt, dass die von der angegebenen Region abgeleiteten Attribute geändert werden, falls es zu einem Widerspruch kommt.
- Der Frequency Cost Manager wählt die Repairing-Variante so, dass die am häufigsten auftretenden Daten nicht verändert werden. Ähneln Werte diesen häufigen Daten, wird forward Repairing angewandt, andernfalls backward Repairing.

Beim Versuch `Llunatic` in einen unendlichen CHASE laufen zu lassen, stoppte die Bearbeitung mit einer Fehlermeldung. Sie wies darauf hin, dass die TGDs nicht schwach azyklisch sind und eine Terminierung nicht sichergestellt werden kann. Zyklische Strukturen werden in `Llunatic` also noch vor der Ausführung erkannt. Das einzige Problem, das während der Untersuchung von `Llunatic` festgestellt werden konnte, tauchte bei einem Versuch auf, Mapping und Cleaning gleichzeitig durchzuführen. In dem gewählten Szenario war es ohne Probleme möglich, einzeln ein Mapping mittels s-t TGD oder ein Cleaning mittels EGD auszuführen. Beide Bedingungen zusammen resultierten in einer Fehlermeldung. Im folgenden Beispiel (aus dem Bericht von Ruven Kronenberg und Sergej Schimanski) ist ein Szenario beschrieben, in dem `Llunatic` Mapping und Cleaning nicht gleichzeitig ausgeführt werden kann:

```
<scenario>
    <source>
        < type>GENERATE</type>
        <generate>
            <! [CDATA[
                SCHEMA:
                A(x, z)
                B(x, y)
                INSTANCE:
                A(x: 1, z: "Foo")
                A(x: 2, z: "Bar")
                B(x: 1, y: 1)
                B(x: 2, y: 1)
             ]]>
        </generate>
    </source>
    <target>
        <type>GENERATE</type>
         <generate>
                <! [CDATA[
                    SCHEMA:
                    C(x, y, z)
                    INSTANCE:
                    C(x: 3, y: 2, z: "Foo")
                    C(x: 4, y: 2, z: "Bar")
                 ]]>
            </generate>
    </target>
    <dependencies>
        <! [CDATA[
            STTGDs:
            A(x: $x1, z: $z), B(x: $x2, y: $y), $x1 == $ x2 ->
            C(x: $x1, z: $z, y: $y).
            ExtEGDs:
            e1: C(y: $y, z: $z1), C(y: $y, z: $z2)-> $z1 == $z2.
        ]]>
    </dependencies>
</scenario>
```

Dieser Fehler konnte jedoch umgangen werden, sobald die s-t TGD so umformuliert wurde, dass die Gleichheit der Variablen bereits in ihr gefordert wurde:

```
1 A(x: $x, z: $z), B(x: $x, y: $y) -> C(x: $x1, z: $z, y: $y)
```

Als Fazit ist zu sagen, dass `Llunatic` nicht jede Mapping-Integritätsbedingung mit jeder Cleaning-Integritätsbedingung zusammen korrekt interpretieren kann. Das Mapping und Cleaning erfolgt bei korrekter Eingabe aber fehlerfrei.

Zusammenfassung Zusammengefasst ist keines der vorgestellten Tools als universelles CHASE-Tool zu bezeichnen. Daher wird mit der Erweiterung von ChaTEAU um den CHASE auf Anfragen, meines Wissens nach, das erste universelle CHASE-Tool entwickelt, welches das einchasen von Integritätsbedingungen in Instanzen und Anfragen nativ unterstützt.

4. Konzept

Im Grundlagen-Kapitel 2 wurde aufgezeigt, dass der CHASE auf Instanzen und auf Anfragen sehr ähnlich abläuft. Die bisherigen Tools, die den CHASE implementiert haben, beschränken sich aber immer auf einen – oder zumindest auf ähnlich ablaufende Anwendungsfälle – und nutzen die Universalität des CHASEs nicht aus. So haben Martin Jurklies [Jur18] und Fabian Renn [Ren19] den CHASE in ChaTEAU zwar für Instanzen und Anfragen implementiert, unterscheiden intern aber je nach übergebenem CHASE-Objekt und führen für diesen dann jeweils eine andere CHASE-Methode aus – `chase(Instance i)` für Instanzen und `chase(Query q)` für Anfragen.

In diesem Kapitel wird ein Konzept erarbeitet, mit dem der CHASE auf Instanzen und Anfragen vereinheitlicht werden kann. Das Konzept lässt sich dann nutzen, um im Kapitel 5 ChaTEAU unter anderem um eine `chase(CHASEobject o)` Methode zu erweitern.

Um so eine universelle CHASE-Methode zu entwickeln, werden wir zuerst untersuchen, wie die Theorie des CHASEs zu der aktuellen Unterscheidung der Methoden in ChaTEAU geführt hat. Diese Unterschiede, die in der Darstellung der beiden CHASE-Objekte und die in den CHASE-Schritten auf ihnen, werden dann nach und nach behoben. Dabei werden auch die nötigen Änderungen an ChaTEAU betrachtet.

4.1. Zusammenfassung der Unterschiede

Die grundlegendsten Unterschiede finden wir schon in der Beschreibung unserer CHASE-Objekte. Während wir Instanzen immer als Menge von Tupeln geschrieben haben, wurden Anfragen als ein prädikatenlogischer Ausdruck definiert. Trotzdem konnten wir bei der Suche nach Triggern den Rumpf einer Integritätsbedingung genau so mit einer Instanz vergleichen wie mit einer Anfrage. Dazu haben wir die Variablen aus dem Rumpf einer solchen Integritätsbedingung einmal auf Konstanten und Nullwerte von einer Instanz und einmal auf Variablen und Konstanten aus den Ausdrücken des Rumpfes der Anfrage abgebildet. Solange der Relationenname und die Anzahl der Elemente übereingestimmt haben, haben wir also einfach die Elemente der Integritätsbedingung positionsabhängig auf die Elemente der CHASE-Objekte abgebildet.

Das Tupel

$$\text{STUDENTEN}(1,\ \mu_{Na}\ , \text{Fabian}, \text{Informatik})$$

unterscheidet sich ohne weiteren Kontext nur dahingehend von dem atomaren Ausdruck

$$\text{STUDENTEN}(x_{Ma}, y_{Na}, x_{Vo}, x_{St}),$$

dass im Tupel Konstanten und Nullwerte stehen und im Ausdruck Variablen und Konstanten. Diese Ähnlichkeit in der Darstellung der Tupel und Ausdrücke werden wir im Abschnitt 4.2.1 ausnutzen, um Instanzen in Anfragen sowie Anfragen in Instanzen umzuformen.

Neben den offensichtlichen Unterschieden zwischen den CHASE-Objekten, müssen wir auch noch die CHASE-Schritte $\mathcal{O}_i \xrightarrow{b_i, g_i} \mathcal{O}_{i+1}$ für Instanzen und Anfragen unterscheiden. Genauer gesagt liegt die Abweichung der einzelnen CHASE-Schritte untereinander in der Anwendung von TGDs und EGDs.

TGDs fügen in Instanzen neue Tupel und in dem Rumpf einer Anfrage weitere Ausdrücke hinzu. Dabei entstehen die Tupel und Ausdrücke aus der Anwendung eines Homomorphismus g_i auf den TGD-Kopf. Dieser Homomorphismus wird auf der Suche nach einem Trigger zwischen TGD-Rumpf und CHASE-Objekt gebildet. Falls es existenzquantifizierte Variablen im TGD-Kopf gibt, muss der Homomorphismus bei Anfragen und bei Instanzen aber noch erweitert werden. Für Instanzen als CHASE-Objekt genügt einer Erweiterung um die Abbildung der existenzquantifizierten Variablen des TGD-Kopfes auf neue Nullwerte. Dagegen muss bei Anfragen eine Abbildung von den existenzquantifizierten Variablen des TGD-Kopfes auf für die Anfrage neue existenzquantifizierte Variablen zu g_i hinzugefügt werden. An dieser Stelle reicht eine Vereinheitlichung der CHASE-Objekte also noch nicht vollends aus. Die CHASE-Objekte müssen auch nach der Vereinheitlichung noch unterscheidbar bleiben, um TGDs im CHASE-Schritt für Instanzen und Anfragen unterschiedlich behandeln zu können.

EGDs definieren über ihre Gleichheitsatome einen Homomorphismus $h : \mathcal{O}_i \longrightarrow \mathcal{O}_{i+1}$ für zwei gleiche CHASE-Objekte. Der Unterschied zwischen Instanzen und Anfragen liegt in den Elementen, die auf einander abgebildet werden können. Während sich bei Instanzen diese Abbildungen nur zwischen Konstanten und/oder Nullwerten ergeben, kommt es bei Anfragen zu Abbildungen zwischen Konstanten und/oder existenzquantifizierten sowie ausgezeichneten Variablen. Die dabei auftauchenden Kombinationen von Abbildungen und ihre Ersetzungsregeln (siehe Definition 2.8) können fast alle, nach der Vereinheitlichung der CHASE-Objekte, gemeinsam behandelt werden (siehe Abschnitt 4.2.2). Das Problem, was zu lösen bleibt, ist, dass der CHASE auf Instanzen fehlschlagen kann, wenn verschiedene Konstanten aufeinandertreffen und $\perp$ (Symbol für Falschheit) zurückgibt, und der CHASE auf Anfragen in diesem Fall eine leere Anfrage $\emptyset$ als Ergebnis zurückgibt. Auch hier müssen die CHASE-Objekte nach der Vereinheitlichung noch unterscheidbar bleiben, um das Aufeinandertreffen von Konstanten in den Gleichheitsatomen einer EGD unterschiedlich behandeln zu können.

CHASE-Objekt:	Instanz	Anfrage
Besteht aus:	Tupel = Konstanten, Nullwerten	Ausdrücken = Konstanten, existenzqu.-, ausgezeichneten Variablen
TGD erzeugt:	neue Tupel mit: neuen Nullwerten	neue Ausdrücke mit: neuen existgenzqu. Variablen
Abbildungen durch EGDs zwischen: Rückgabe bei $c \neq c$:	Konstanten u./o. Nullwerten $\perp$	Konstanten u./o. existenzqu. sowie ausgezeichneten Variablen $\emptyset$

Tabelle 4.1.: Übersicht der Unterschiede

Die Unterschiede in den Definitionen des CHASEs auf Instanzen und Anfragen (siehe Zusammenfassung in Tabelle 4.1) haben dazu geführt, dass es in ChaTEAU bis jetzt zwei verschiedene Umsetzungen für den CHASE gibt. Im nächsten Abschnitt geht es um die Vereinheitlichung der CHASE-Objekte und um die Änderungen in der Handhabung der TGDs und EGDs in den CHASE-Schritten auf den vereinheitlichten CHASE-Objekten.

4.2. Behebung der Unterschiede

Nachdem wir die wichtigsten Unterschiede zwischen dem CHASE auf Instanzen und dem CHASE auf Anfragen identifiziert haben, können diese jetzt behoben werden. Dafür müssen wir als Erstes entscheiden, ob Instanzen in Anfragen oder Anfragen in Instanzen umgeformt werden sollen. Diese Entscheidung wird dann ebenfalls Einfluss darauf haben, ob und wie die CHASE-Schritte erweitert und abgeändert werden müssen.

4.2.1. CHASE-Objekte vereinheitlichen

Für die Vereinheitlichung der CHASE-Objekte werden wir die Struktur eines bestehenden CHASE-Objektes erweitern. Im Folgenden werden die Konzepte der Umwandlung der CHASE-Objekte ineinander beschrieben und die jeweiligen Vor- und Nachteile bewertet. Dabei spielt auch die bisherige Implementierung von ChaTEAU eine große Rolle, da ursprünglich nur der CHASE auf Instanzen implementiert wurde und der später hinzugefügte CHASE auf Anfragen einige Methoden ohne weitere Anpassungen mitbenutzt. Somit gibt es bei der Erkennung von Triggern auf Anfragen und in der Ersetzung von Termen bei der Einarbeitung von EGDs in Anfragen noch ungelöste Probleme [Ren19].

Instanz als Anfrage Die eben angesprochenen Probleme in der bisherigen Implementierung des CHASE auf Anfragen würden uns in ChaTEAU begegnen, wenn wir Instanzen in Anfragen umwandeln. Das Fazit dieses Abschnitts vorweggreifend, wird die Entscheidung in dieser Arbeit gegen dieses Verfahren fallen. Da es aber dennoch einige Vorzüge bietet, die bei einer frischen Implementierung des CHASE auch Vorteile gegenüber der Umwandlung von Anfragen in Instanzen hat, werden wir das Konzept dahinter als mögliche Grundlage anderer Arbeiten trotzdem betrachten.

Rufen wir uns noch einmal die Gegenüberstellung eines Tupels in Instanzen und eines Ausdrucks in Anfragen auf,

$$\begin{aligned}\text{Tupel} &: \text{STUDENTEN}(1,\ \mu_{Na}\ , \text{Fabian}, \text{Informatik})\\ \text{Ausdruck} &: \text{STUDENTEN}(x_{Ma}, y_{Na}, x_{Vo}, x_{St})\end{aligned}$$

dann haben wir bereits festgestellt, dass der einzige Unterschied die erlaubten Elemente in den jeweiligen Konstrukten sind. Während Konstanten in beiden erlaubt sind, gibt es in Instanzen Nullwerte und in Anfragen existenzquantifizierte und ausgezeichnete Variablen. Bis auf die Nullwerte entspricht eine Instanz also einem Ausdruck ohne Variablen (auch *Fakt* genannt). Somit kann man die Tupel einer Instanz auch als Anfrage schreiben, deren Rumpf nur aus konjungierten Fakten besteht und deren Kopf leer bleibt. Da unsere Definition der Anfragen keine Nullwerte in den Ausdrücken erlaubt, ersetzen wir diese durch existenzquantifizierte Variablen.

Als Beispiel wandeln wir die Instanz I in die Anfrage Q_I um. Dabei werden die Konstanten der Tupel in den Ausdrücken übernommen und der Nullwert μ_{Na} durch die existenzquantifizierte Variable y_{Na} ersetzt. Da wir bei dieser Umformung keine ausgezeichneten Variablen erzeugen, bleibt der Kopf von Q_I leer.

$$\begin{aligned}I =&\{\text{STUDENTEN}(1,\ \mu_{Na}\ , \text{Fabian}, \text{Informatik}),\\ &\text{TEILNEHMER}(2,1)\}\\ Q_I =&(\exists y_{Na} : \text{STUDENTEN}(1, y_{Na}, \text{Fabian}, \text{Informatik})\\ &\wedge \text{TEILNEHMER}(2,1)) \longrightarrow ()\end{aligned}$$

Um die Auswirkung dieser Umwandlung auf die CHASE-Schritte zu untersuchen, wenden wir jetzt den CHASE auf unsere frühere Instanz an. Konstanten und existenzquantifizierte Variablen verhalten sich weiterhin genau so, wie die Konstanten und Nullwerte einer Instanz. Bei der Suche nach Triggern wird der Rumpf einer Integritätsbedingung genau so auf die Konstanten und existenzquantifizierten Variablen der Ausdrücke abgebildet, wie auf die Konstanten und Nullwerte in Tupeln. Beim Anwenden einer TGD wird der Kopf, unter Anwendung des durch den Trigger definierten Homomorphismus, genau so als konjungierter Ausdruck in die Anfrage eingefügt wie neue Tupel in eine Instanz. Beim Anwenden einer EGD können existenzquantifizierte Variablen in Anfragen genau so wie Nullwerte in Instanzen durch sich selbst, andere existenzquantifizierte Variablen oder eben durch Konstanten ersetzt werden. Der Fall, dass eine existenzquantifizierte durch eine ausgezeichnete Variable ersetzt wird, kommt nach Konstruktion der Anfrage nicht vor, da wir die Ausdrücke nur mit Konstanten und existenzquantifizierten Variablen füllen. Der einzige bleibende Unterschied ist, dass der CHASE auf Anfragen nicht fehlschlägt. Aus diesem Grund muss man zwischen ursprünglichen Anfragen und Instanzen, die als Anfrage geschrieben wurden, unterscheiden, um das Ergebnis je nachdem interpretieren zu können. Eine leere Anfrage, die aus einer Instanz entstanden ist, bedeutet nämlich, dass der CHASE fehlgeschlagen ist. Mit der Möglichkeit der Unterscheidung kann man die Ergebnisanfrage dann auch wieder in eine Instanz zurück wandeln, wenn sie aus einer entstanden ist.

Als Beispiel chasen wir die TGD b in Q_I ein. Der CHASE findet mit Abbildung des TGD-Rumpfes auf den Anfragerumpf einen aktiven Trigger. Unter Anwendung dieser Abbildung wird die Anfrage um den Noten-Ausdruck aus dem TGD-Kopf erweitert. Bei der Rückwandlung von Q_I in eine Instanz werden die Ausdrücke im Rumpf wieder als Tupel geschrieben und die existenzquantifizierten Variablen durch Nullwerte ersetzt. Die Nullwerte entsprechen dann eventuell nicht mehr den originalen Nullwerten, doch die Gleichheiten zwischen den Nullwerten bleiben erhalten. Wenn man auf die originalen Nullwerte besteht, kann man bei der Ersetzung der Nullwerte durch existenzquantifizierte Variablen eine Hash-Tabelle verwenden und dieses bei der Rückwandlung mit einbeziehen. Die so entstandene Instanz entspricht dem Ergebnis des einchasen von b in I (CHASE$(\{b\}, I)$). So haben wir mit dem CHASE auf einer Anfrage eine Integritätsbedingung in eine Instanz eingechaset:

$$
\begin{aligned}
b =& \text{STUDENTEN}(x_{Ma}, x_{Na}, x_{Vo}, x_{St}) \wedge \text{TEILNEHMER}(x_{Mo}, x_{Ma}) \\
& \longrightarrow \exists y_{Se}, y_{No} : \text{NOTEN}(x_{Mo}, x_{Ma}, y_{Se}, y_{No}) \\
\text{CHASE}(\{b\}, Qi) =& (\exists y_{Na}, y_{Se}, y_{No} : \text{STUDENTEN}(1, y_{Na}, \text{Fabian}, \text{Informatik}) \\
& \wedge \text{TEILNEHMER}(2, 1) \wedge \text{NOTEN}(2, 1, y_{Se}, y_{No})) \longrightarrow () \\
\text{Ergebnis-Instanz} : I_{QI} =& \{\text{STUDENTEN}(1,\ \mu_{Na}\ , \text{Fabian}, \text{Informatik}), \\
& \text{TEILNEHMER}(2, 1), \\
& \text{NOTEN}(2, 1, \mu_{Se}, \mu_{No})\}.
\end{aligned}
$$

Ein Vorteil der Umwandlung von Instanzen in Anfragen ist, dass das Konstrukt der Anfrage nicht verändert werden muss. Auch die Ersetzungsregeln in den CHASE-Schritten könnten gleich bleiben. Sogar die bei Instanzen notwendige Erweiterung des Homomorphismus um die Abbildung von existenzquantifizierten Variablen auf neue Nullwerte würde entfallen, da am Ende alle neuen existenzquantifizierten Variablen durch neue Nullwerte ersetzt werden. Das Ergebnis müsste dann nur noch richtig interpretiert und wenn nötig zurück gewandelt werden. Dazu könnte eine Objektidentifizierungs-Variable oder eine Namenskonvention eingeführt werden, mit der sich unterscheiden lässt, ob die Anfrage aus einer Instanz entstanden ist oder diese eine ursprüngliche Anfrage war.

Das Problem an diesem Konzept ist, dass die Logik hinter einer aus einer Instanz entstandenen Anfrage wenig Sinn machen würde. So würden aus zwei STUDENTEN-Tupeln mit unterschiedlicher Matrikelnummer, zwei STUDENTEN-Ausdrücke in der Anfrage mit unterschiedlicher Matrikelnummer entstehen.

Diese Anfrage würde also rein logisch kein Ergebnis liefern, da ein Widerspruch in den Bedingungen besteht. Ebenso macht der leere Kopf der Anfrage keinen Sinn, da man dann sowieso ein leeres Ergebnis erhalten würde, was wiederum eine Optimierung überflüssig macht. Die als Anfrage umgeschriebene Instanz ist also nur formal eine Anfrage, bei der man rein logisch nicht von einer Anfragen-Optimierung durch den CHASE sprechen kann. Eigentlich würde eine Optimierung einer solchen Anfrage immer die leere Anfrage ergeben.

Vom Aufwand her wäre dieses Konzept eine akzeptable Lösung, um den CHASE auf Instanzen und Anfragen zu vereinheitlichen. Durch die bisherige Entwicklung in ChaTEAU, indem der CHASE auf Anfragen noch fehlerbehaftet ist, werden wir uns aber für das folgende Konzept – Anfragen in Instanzen umwandeln – entscheiden.

Anfrage als Instanz In dieser Arbeit werden wir das Konzept verfolgen, eine Anfrage in eine Instanz umzuformen. Das hat für ChaTEAU den Vorteil, dass wir die bestehende funktionierende Struktur des CHASEs auf Instanzen nutzen und erweitern können. Die Umformung einer Anfrage in eine Instanz haben wir bereits beim Konzept der Darstellung einer Instanz als Anfrage bei der Rückwandlung, der aus einer Instanz entstandenen Anfrage, durchgeführt. Unsere Anfrage war in diesen Fall allerdings auf Konstanten und existenzquantifizierte Variablen reduziert. Da wir für die ausgezeichneten Variablen kein Äquivalent auf der Seite der Instanzen haben, können wir eine Anfrage diesmal nicht ohne Weiteres in das Konstrukt einer Instanz überführen.

Für die Lösung dieses Problems gibt es verschiedene Ansätze, bei denen Anfragen als Instanzen geschrieben werden. Das Prinzip dahinter nennt sich “frozen instance”. Allgemein wird für eine Anfrage Q eine symbolische Instanz I_Q gebildet, in der jeder Ausdruck aus Q in einem Tupel in I_Q repräsentiert wird [DNR08]. Für den Umgang mit den Variablen gibt es hierbei verschiedene Ansätze. Oft werden die existenzquantifizieren Variablen in Nullwerte umgeformt und die ausgezeichneten Variablen als gelabelte Nullwerte oder besondere Konstanten behandelt. Wir werden bei der Umformung einer Anfrage in eine Instanz einfach die Ausdrücke aus dem Rumpf der Anfrage direkt als Tupel einer Instanz schreiben. Das Konstrukt der Instanz muss neben Konstanten und Nullwerten also um existenzquantifizierte und ausgezeichnete Variablen erweitert werden. Um diese symbolische Instanz von einer normalen zu unterscheiden, wird sie im weiteren Verlauf der Arbeit als *entartete Instanz* bezeichnet.

Für die Umwandlung der Anfrage Q in die entartete Instanz I_Q, muss nur die konjunktiven Ausdrücke als *entartete Tupel* geschrieben werden. Dabei werden wir Tupel erzeugen, die existenzquantifizierte und ausgezeichnete Variablen als Attributwerte haben:

$$
\begin{aligned}
Q =& (\exists y_{Na}, y_{Vo}, y_{St} : \text{STUDENTEN}(x_{Ma}, y_{Na}, y_{Vo}, y_{St}) \wedge \text{TEILNEHMER}(x_{Mo}, x_{Ma})) \\
& \longrightarrow x_{Ma}, x_{Mo} \\
I_Q =& \{\text{STUDENTEN}(x_{Ma}, y_{Na}, y_{Vo}, y_{St}), \\
& \text{TEILNEHMER}(x_{Mo}, x_{Ma})\}.
\end{aligned}
$$

Dank der Einschränkung in der Definition von Anfragen (Definition 2.5), die verlangt, dass alle ausgezeichneten Variablen des Rumpfes einer Anfrage den Kopf der Anfrage definieren, können wir diesen jederzeit aus dem Rumpf der Anfrage ableiten und bei der Umwandlung vernachlässigen. So können wir den CHASE auf der entarteten Instanz durchführen und dann bei der Rückwandlung einen neuen Kopf aus allen ausgezeichneten Variablen erzeugen. Mit diesem Vorgehen entfällt auch die zusätzliche Anwendung der möglichen Variablenersetzungen durch eine EGD im Kopf der Anfrage.

Der große Vorteil dieses Konzeptes besteht darin, dass an der bestehenden, größten Teils funktionierenden Logik des CHASEs auf Instanzen in ChaTEAU aufgesetzt werden kann. Instanzen sind sogar bereits so implementiert, dass sie Variablen in den Tupeln beinhalten können (siehe Aufbau einer Instanz aus Termen, Abschnitt 3.1, Abbildung 3.1). Ebenso wird durch die Übernahme der originalen Variablen aus den Anfragen eine Umwandlung und Rückumwandlung der CHASE-Objekte vereinfacht. Wir müssen uns so keine Hash-Tabellen zwischenspeichern, um die ursprünglichen Bezeichnungen der Variablen zu erhalten.

Wie schon angesprochen, hat dieses Konzept aber auch Nachteile im Vergleich zur Umwandlung von Instanzen in Anfragen. So muss das logische Konstrukt der Instanz verändert werden, um Variablen in den Tupeln zu unterstützen, und erzwingt daher auch viele Änderungen in den CHASE Schritten selbst. Hier treffen allerdings keine Elemente aufeinander, die in den einzelnen CHASE-Fällen nicht auch aufeinander treffen würden. So reicht fast die Übernahme der Ersetzungsregeln für Anfragen in den CHASE auf Instanzen aus, um die entarteten Instanzen zu unterstützen.

Im nächsten Abschnitt wird Genaueres zur Anpassung der CHASE-Schritte auf entartete Instanzen erklärt. Einige Ersetzungsregeln werden wir vom CHASE auf Anfragen übernehmen können, während wir andere extra behandeln müssen.

4.2.2. CHASE-Schritte angleichen

Um das korrekte einchasen von Integritätsbedingungen in ursprüngliche Instanzen und Anfragen als entartete Instanzen zu gewährleisten, reicht eine einfache Übernahme der Ersetzungsregeln für Anfragen leider nicht ganz aus. In diesem Abschnitt klären wir, an welcher Stelle es reicht, die verschiedenen Ersetzungsregeln für Instanzen und für Anfragen zusammenzufassen und wo und warum eine Unterscheidung nötig bleibt.

Zusammenfassen Bei der Erweiterung der bisherigen Regeln für Instanzen können wir alle Abbildungsregeln für Homomorphismen zwischen Anfragen (Definition 2.8) übernehmen. Ebenso ergänzen wir die Regeln für die Anwendung einer EGD im CHASE-Schritt um die Regeln für den CHASE auf Anfragen (Definition 2.14). Eine Ausnahme bildet das Aufeinandertreffen von verschiedenen Konstanten, da hier ein Widerspruch zu der bereits bestehenden Regel für Instanzen entstehen würde.

Unterscheiden Die eben angesprochene Ausnahme für das Aufeinandertreffen zweier Konstanten im Kopf einer EGD rührt daher, dass der CHASE auf Anfragen nicht fehlschlagen kann, sondern stets die leere Anfrage $\emptyset$ als optimierte Anfrage liefert. Um dieses Problem zu lösen, muss man genau wie bei der Umwandlung von Instanzen in Anfragen eine ursprüngliche Instanz von einer entarteten Instanz aka ehemaligen Anfrage unterscheiden können. Anhand so einer Objektidentifizierungs-Variable oder Namenskonvention zur Unterscheidung kann dann das Ergebnis des CHASE auf entarteten Instanzen korrekt interpretiert werden, sprich, ursprüngliche Anfragen wieder in solche zurück wandeln, um beim fehlgeschlagenen CHASE eine leere Anfrage zurückgeben. Diese Unterscheidung ist außerdem nötig, um in entarteten Instanzen, die aus einer Anfrage entstanden sind, keine neuen Nullwerte, sondern neue existenzquantifizierte Variablen bei der Anwendung von TGDs hinzuzufügen. Hier muss das einzige Mal innerhalb des CHASE-Algorithmus eine Unterscheidung zwischen den Objekten bestehen bleiben.

Bisher haben wir in diesem Kapitel die Unterschiede zwischen dem CHASE auf Instanzen und dem CHASE auf Anfragen (siehe Abschnitt 2.2.1 und 2.2.2) identifiziert (siehe Abschnitt 4.1) und mit den

entarteten Instanzen ein geeignetes Konzept für die Vereinheitlichung der beiden Algorithmen und den zugehörigen CHASE-Objekten beschrieben (siehe Abschnitt 4.2). Im nächsten Abschnitt werden die Ergebnisse des Konzepts anhand von formalen Definitionen zusammengefasst.

4.3. CHASE auf entartete Instanzen

Nachdem beschrieben wurde, wie wir die CHASE-Objekte und den CHASE-Algorithmus auf ihnen vereinheitlichen können, werden wir jetzt den CHASE auf den vorgestellten entarteten Instanzen formal definieren. Die Erkenntnisse aus dem Abschnitt 4.2 fassen wir dabei angelehnt an den Definitionen aus dem CHASE-Abschnitt 2.2 zusammen.

Definition 4.1. (Entartete Instanz): Für eine Anfrage $Q : \exists y : \phi(x, y) \longrightarrow x_1, ..., x_m$, nennen wir die Menge der entarteten Tupel, die aus $\phi(x, y)$ gebildet werden, die entartete Instanz I_Q zu Q. Ein entartetes Tupel, einer Relation entsprechend des Bezeichners eines Ausdrucks, entsteht dabei aus der exakten Übernahme der einzelnen Elemente des atomaren Ausdrucks. Äquivalent zu den Ausdrücken bestehen die entarteten Tupel nur aus Konstanten, ausgezeichneten und existenzquantifizierten Variablen (siehe Definition 2.5).

Für eine Instanz I verhält sich die entartete Instanz I_I wie die Instanz I nach Definition 2.4 und speichert dementsprechend Tupel bestehend aus Konstanten und Nullwerten. Je nachdem aus welchem CHASE-Objekt die entartete Instanz entstanden ist, lässt sie also unterschiedliche Elemente in den Tupeln zu. Der Ursprung der entarteten Instanz wird durch den Tag $_I$ oder $_Q$ unterschieden. Daher unterscheiden wir im Weiteren nicht mehr zwischen entarteten und nicht entarteten Tupeln.

Da die entarteten Instanzen ein Konstrukt für die beiden CHASE-Objekte – Instanzen und Anfragen – bilden, müssen diese in den meisten Definitionen nicht mehr unterschieden werden. Per Definition können Variablen nur aus Anfragen und Nullwerte nur aus Instanzen in die entarteten Instanzen eingefügt werden. Daher reicht größtenteils die Unterscheidung der Elemente in den Tupeln der entarteten Instanz aus. Für den Fall, dass das ursprüngliche CHASE-Objekt nicht ausschlaggebend für die Definition ist, schreiben wir die entartete Instanz als I_O.

Definition 4.2. (Homomorphismus für entartete Instanzen, nach [GMS12]): Seien I_{O1} und I_{O2} zwei entartete Instanzen. Ein *Homomorphismus* $h : I_{O1} \longrightarrow I_{O2}$ ist eine Abbildung unter folgenden Regeln:

(1) Eine Konstante c kann nur auf sich selbst abgebildet werden: $h(c) = c$,

(2) Ein Nullwert μ_i kann auf eine Konstante c, sich selbst oder einen anderen Nullwert μ_j abgebildet werden: $h(\mu_i) = [c|\mu_i|\mu_j]$,

(3) Eine ausgezeichnete Variable x_i kann nur auf eine Konstante c oder auf sich selbst abgebildet werden: $h(x_i) = [c|x_i]$,

(4) Eine existenzquantifizierte Variable y_i kann auf eine Konstante c, auf eine ausgezeichnete Variable x_j, auf sich selbst oder auf eine andere existenzquantifizierte Variable y_j abgebildet werden: $h(y_i) = [c|x_j|y_i|y_j]$,

wobei für jedes Tupel $t = (a_1, ..., a_n)$ aus den Relationen in I_{O1} ein Tupel $h(t)$ in den Relationen aus I_{O2} existiert, mit $h(t) = (h(a_1), ..., h(a_n))$.
Die CHASE-Objekte I_{O1} und I_{O2} gelten als *äquivalent* (geschrieben: $I_{O1} \leftrightarrow I_{O2}$) genau dann, wenn es einen Homomorphismus von I_{O1} nach I_{O2} ($h_1 : I_{O1} \longrightarrow I_{O2}$) und einen Homomorphismus von I_{O2} nach I_{O1} ($h_2 : I_{O2} \longrightarrow I_{O1}$) gibt.

Definition 4.3. (Trigger für entartete Instanzen, nach [Jur18, Ren19]): Die Homomorphismen zwischen den Integritätsbedingungen in $\mathcal{B}$ ($b_i = [TGD|EGD]$) und einer entarteten Instanz I_O werden *Trigger* genannt. Dabei wird der Rumpf von b_i auf I_O abgebildet. Der Rumpf von b_i wird auf die Tupel aus den Relationen in I_O abgebildet: $g_i : \phi(x) \longrightarrow I_O$, wobei die einzelnen Variablen x_i aus den atomaren Ausdrücken von $\phi(x)$ auf die einzelnen Konstanten, Nullwerte oder Variablen der Tupel abgebildet werden.

Definition 4.4. (Aktiver Trigger für entartete Instanzen, nach [Jur18, Ren19]): Ein *aktiver Trigger* ist ein Trigger g_i, für den gilt:

- Wenn b_i eine TGD ist, dann existiert keine Erweiterung von g_i zu einem Homomorphismus vom Kopf der TGD zu I_O ($\psi(x, y) \longrightarrow I_O$).
- Wenn b_i eine EGD ist, dann ist $g_i(x_i) \neq g_i(x_j)$ für mindestens ein Gleichheitsatom ($x_i = x_j$) aus dem Kopf der EGD.

Eine entartete Instanz I_O *genügt* b_i, genau dann wenn kein aktiver Trigger für b_i in I_O existiert.

Für die nächsten Definitionen galt es zu entscheiden, ob die Unterscheidung zwischen fehlgeschlagenem und abgebrochenem CHASE je nach Ursprung der entarteten Instanz im CHASE-Algorithmus getroffen wird oder ob man den CHASE immer fehlschlagen lässt und dann das Ergebnis je nach Ursprung interpretiert. Da eine entartete Instanz, die aus einer Anfrage entstanden ist, auch wieder in eine Anfrage umgewandelt werden wird, reicht eine Interpretation der Ausgabe $\perp$ als leere Anfrage $\emptyset$ an dieser Stelle aus. Diese Entscheidung erspart es aber nicht, überhaupt eine Unterscheidung im Algorithmus zu treffen. Bei der Anwendung einer TGD mit existenzquantifizierten Variablen im Kopf, muss immer noch unterschieden werden, wie genau der Homomorphismus erweitert wird, sprich, ob neue Nullwerte oder neue existenzquantifizierte Variablen eingefügt werden.

Das Konzept des CHASE auf entartete Instanzen ist im Algorithmus 3 beschrieben. Die Zeilen 5 bis 23 stellen den CHASE-Schritt auf Instanzen da. Die Unterscheidung des Ursprungs einer entarteten Instanz beim einchasen einer TGD ist mit $I_O \equiv I_I$ für eine ursprüngliche Instanz und mit $I_O \equiv I_Q$ für eine ursprüngliche Anfrage beschrieben. Die Erweiterung des Triggers g mit $y \longrightarrow \mu$ steht für die Erweiterung um Abbildungen von den existenzquantifizierten Variablen des TGD-Kopfes auf neue benannte Nullwerte. Die Erweiterung mit $y \longrightarrow y$ steht für die Erweiterung um Abbildungen von den existenzquantifizierten Variablen des TGD-Kopfes auf neue existenzquantifizierte Variablen.

Definition 4.5. (CHASE-Schritt auf entartete Instanzen, nach [FKMP03]): Seien alle Bedingungen für einen CHASE-Schritt $I_{Oi} \xrightarrow{b_i, g_i} I_{Oi+1}$ nach Definition 2.11 mit einer entarteten Instanz I_{Oi} als CHASE-Objekt gegeben.

(a) Sei b_i eine TGD.

- Die Anwendung von b_i erweitert I_{Oi} um Tupel $R_j(w_1, ..., w_p)$ mit den Relationennamen R_j gleich den Bezeichnern der Ausdrücke $R_j(a_1, ..., a_p)$ aus dem Kopf der TGD. Falls a_k für eine existenzquantifizierte Variable y_k aus der TGD steht, wird der Homomorphismus g_i um die Abbildung von y_k auf
  - einen benannten Nullwert erweitert, falls I_{Oi} aus einer Instanz entstanden ist,
  - eine neue existenzquantifizierte Variable erweitert, falls I_{Oi} aus einer Anfrage entstanden ist.

  Die Konstanten, Nullwerte und Variablen w_k ergeben sich dabei aus der Anwendung des Homomorphismus auf die Variablen der Ausdrücke des TGD-Kopfes, $g_i(a_k)$.

(b) Sei b_i eine EGD.

- Falls für ein Gleichheitsatom $(x_i = x_j)$ die Abbildungen $g_i(x_i)$ und $g_i(x_j)$ unterschiedliche Konstanten ergeben, dann schlägt der CHASE fehl ($I_{Oi+1} = \perp$).

- Falls eine Variable des Gleichheitsatoms $(x_i = x_j)$ auf eine Konstante abgebildet wird und die andere auf einen benannten Nullwert (z. B. $g_i(x_i) = 1$, $g_i(x_j) = \mu_{Ma}$), dann wird der Nullwert überall durch die Konstante ersetzt ($h(\mu_{Ma}) = 1$ mit $h : I_{Oi} \longrightarrow I_{Oi+1}$).

- Falls eine Variable des Gleichheitsatoms $(x_i = x_j)$ auf eine Konstante abgebildet wird und die andere auf eine existenzquantifizierte oder ausgezeichnete Variable (z. B. $g_i(x_i) = 1$, $g_i(x_j) = x_{Ma}$), dann wird die Variable überall durch die Konstante ersetzt ($h(x_{Ma}) = 1$ mit $h : I_{Oi} \longrightarrow I_{Oi+1}$).

- Falls beide Variablen des Gleichheitsatoms $(x_i = x_j)$ auf unterschiedlich benannte Nullwerte abgebildet werden (z. B. $g_i(x_i) = \mu_{No1},\ g_i(x_j) = \mu_{No2}$), dann wird ein Nullwert überall durch den anderen ersetzt. Die Auswahl erfolgt über den durch die Namenskonvention vorgeschriebenen Index der Nullwerte. Die Konvention besagt, dass verschiedene Nullwerte für ein Attribut durchnummeriert werden. So kann man den Nullwert mit dem größeren Index durch den Nullwert mit dem kleineren Index ersetzen ($h(\mu_{No2}) = \mu_{No1}$ mit $h : I_{Oi} \longrightarrow I_{Oi+1}$).

- Falls eine Variable des Gleichheitsatoms $(x_i = x_j)$ auf eine ausgezeichnete Variable abgebildet wird und die andere auf eine existenzquantifizierte Variable (z. B. $g_i(x_i) = x_{Ma}, g_i(x_j) = y_{Ma}$), dann wird die existenzquantifizierte Variable überall durch die ausgezeichnete Variable ersetzt ($h(y_{Ma}) = x_{Ma}$ mit $h : I_{Oi} \longrightarrow I_{Oi+1}$).

- Falls beide Variablen des Gleichheitsatoms $(x_i = x_j)$ auf unterschiedlich existenzquantifizierte Variablen abgebildet werden (z. B. $g_i(x_i) = y_{No1},\ g_i(x_j) = y_{No2}$), dann wird eine existenzquantifizierte Variable überall durch die andere ersetzt. Die Auswahl erfolgt über den durch die Namenskonvention vorgeschriebenen Index der existenzquantifizierten Variablen. Die Konvention besagt, dass verschiedene existenzquantifizierte Variablen für ein Attribut durchnummeriert werden. So kann man die Variablen mit dem größeren Index durch die Variablen mit dem kleineren Index ersetzen ($h(y_{No2}) = y_{No1}$ mit $h : I_{Oi} \longrightarrow I_{Oi+1}$).

  Diese Regeln werden für jedes Gleichheitsatom aus b_i auf I_{Oi} angewandt. Bei der Anwendung wird ein Homomorphismus h zwischen den entarteten Instanzen I_{Oi} und I_{Oi+1} gebildet, der die Werte die ersetzt werden (z. B. $g_i(x_j) = \mu_{Ma}$) auf die ersetzenden Werte (z. B. $g_i(x_i) = 1$) abbildet ($h(g_i(x_j)) = g_i(x_i) \longrightarrow h(\mu_{Ma}) = 1$ mit $h : I_{Oi} \longrightarrow I_{Oi+1}$).

Die Anwendung eines CHASE-Schrittes überführt die entartete Instanz I_{Oi} in eine modifizierte entartete Instanz I_{Oi+1}.

Definition 4.6. (Standard-CHASE auf entartete Instanzen, nach [FKMP03]): Sei $\mathcal{B}$ eine Menge von TGDs und EGDs und I_O eine entartete Instanz.

- Eine *CHASE-Sequenz* auf I_O mit $\mathcal{B}$ ist eine (endliche oder unendliche) Sequenz von CHASE-Schritten $I_{Oi} \xrightarrow{b_i, g_i} I_{Oi+1}$, mit $i \in \mathbb{N}_0, I_{O0} = I_O$ und b_i eine Integritätsbedingung aus $\mathcal{B}$.

- Ein *endlicher CHASE* auf I_O mit $\mathcal{B}$ ist eine endliche Sequenz $I_{Oi} \xrightarrow{b_i, g_i} I_{Oi+1}, 0 \leq i \leq m$, mit der Bedingung, dass entweder (a) $I_{Om} = \perp$ oder (b) es existiert keine Integritätsbedingung b_m in $\mathcal{B}$ und es gibt keinen Homomorphismus g_m, sodass b_m mit g_m auf I_{Om} abgebildet werden kann. Wir bezeichnen I_{Om} als das Ergebnis des endlichen CHASEs – für den Fall (a) das Ergebnis eines *fehlgeschlagenen endlichen CHASEs* und für den Fall (b) das Ergebnis eines *erfolgreichen endlichen CHASEs.*

Mit anderen Worten werden beim Standard-CHASE solange CHASE-Schritte angewandt, bis der CHASE entweder fehlschlägt oder es keine aktiven Trigger mehr für alle $b_i \in \mathcal{B}$ auf dem CHASE-Objekt I_{Oi} gibt. Die Auswahl der Integritätsbedingungen erfolgt dabei nicht-deterministisch, weshalb die Terminierung des Standard-CHASEs nicht entscheidbar ist und verschiedene CHASE-Sequenzen über dem gleichen Objekt und gleicher Integritätsbedingung unterschiedliche Ergebnisse haben können.

Definition 4.7. (Interpretation des Ergebnisses): Sei I_{Im} das Ergebnis des CHASEs auf einer ursprünglichen Instanz I als entartete Instanz I_{I0}, dann entspricht I_{Im} dem Ergebnis des CHASEs auf I.
Sei I_{Qm} das Ergebnis des CHASEs auf einer ursprünglichen Anfrage Q als entartete Instanz I_{Q0}, dann würde der CHASE genau dann abbrechen und die leere Anfrage $\emptyset$ als Ergebnis liefern, wenn $I_{Qm} = \perp$. Das Ergebnis des erfolgreichen endlichen CHASEs auf Q entspricht der Umwandlung I_{Qm} in eine Anfrage. Bei der Umwandlung werden für jedes Tupel aus I_{Qm} Ausdrücke mit Bezeichnern entsprechend des Relationennamens der Tupel und Elementen gleich denen der Tupel in eine Konjunktion geschrieben. Diese Konjunktion der atomaren Ausdrücke bildet den Rumpf der Ergebnisanfrage. Der Kopf der Ergebnisanfrage wird aus allen ausgezeichneten Variablen des Rumpfes gebildet.

Algorithmus 3 Standard-CHASE auf entarteten Instanzen($\mathcal{B}$, I_O)

Require: Menge von Integritätsbedingungen (TGDs, EGDs) $\mathcal{B}$, Instanz I_{O0}
Ensure: Modifizierte Instanz I_{Om}

```
1: while I_Oi ≠ ⊥ ∧ ∃ aktive Trigger für ein b ∈ B und I_Oi, 0 ≤ i ≤ m do
2:   for all b ∈ B do
3:     for all Trigger g für b und I_Oi do
4:       if g ist ein aktiver Trigger then
5:         if b ist eine TGD then
6:           if I_O ≡ I_I then
7:             Erweitere g falls nötig (y ⟶ μ)
8:           else if I_O ≡ I_Q then
9:             Erweitere g falls nötig (y ⟶ y)
10:          Füge neue Tupel zu der Instanz I_Oi hinzu
11:        else if b ist eine EGD then
12:          if Verglichene Werte sind verschiedene Konstanten then
13:            I_m = ⊥
14:          else if Verglichene Werte sind Konstante c und Nullwert ν then
15:            Ersetzte ν durch c
16:          else if Verglichene Werte sind Konstante c und Variable a then
17:            Ersetze a durch c
18:          else if Verglichene Werte sind verschiedene Nullwerte ν_1 und ν_2 then
19:            Ersetze ν_2 durch ν_1
20:          else if Verglichene Werte sind ausgezeichnete Variable x und existenzqu. Variable y then
21:            Ersetze y durch x
22:          else if Verglichene Werte sind verschiedene existenzqu. Variablen y_1 und y_2 then
23:            Ersetzte y_2 durch y_1
```

Sei die Anfrage Q und die Integritätsbedingung b (TGD) wie beim Beispiel für den CHASE auf Anfragen im Abschnitt 2.2.2 gegeben und sei I_Q die aus Q entstandene entartete Instanz (siehe Beispiel im Abschnitt 4.2.1):

$$
\begin{aligned}
Q =& (\exists y_{Na}, y_{Vo}, y_{St} : \text{STUDENTEN}(x_{Ma}, y_{Na}, y_{Vo}, y_{St}) \wedge \text{TEILNEHMER}(x_{Mo}, x_{Ma})) \\
& \longrightarrow x_{Ma}, x_{Mo} \\
I_Q =& \{\text{STUDENTEN}(x_{Ma}, y_{Na}, y_{Vo}, y_{St}), \\
& \text{TEILNEHMER}(x_{Mo}, x_{Ma})\} \\
b =& \text{STUDENTEN}(x_{Ma}, x_{Na}, x_{Vo}, x_{St}) \wedge \text{TEILNEHMER}(x_{Mo}, x_{Ma}) \\
& \longrightarrow \exists y_{Se}, y_{No} : \text{NOTEN}(x_{Mo}, x_{Ma}, y_{Se}, y_{No}).
\end{aligned}
$$

Genau wie im Beispiel mit einer normalen Instanz im Abschnitt 2.2.1 suchen wir wieder nach Triggern für die Integritätsbedingungen in $\mathcal{B} = \{b\}$ und der entarteten Instanz $I_Q \equiv I_{Q0}$. Wir wählen wieder als Erstes die Integritätsbedingung b. Die Trigger finden wir, indem wir zwischen dem Rumpf von b und den Tupeln von I_{Q0} nach Homomorphismen suchen. Auch hier existiert ein Homomorphismus g_0 mit jeweils einer Abbildung zwischen den STUDENTEN- und TEILNEHMER-Tupeln von I_{Q0} und den STUDENTEN- und TEILNEHMER-Ausdrücken im Rumpf von b ($g_0(x_{Ma}) = x_{Ma},\ g_0(x_{Na}) = y_{Na},\ g_0(x_{Vo}) = y_{Vo},\ g_0(x_{St}) = y_{St},\ g_0(x_{Mo}) = y_{Mo}$). An dieser Stelle unterscheidet sich der Homomorphismus g_0 in keiner Weise von dem aus dem Beispiel für den CHASE auf Anfragen. Wieder prüfen wir auf einen aktiven Trigger und suchen nach Erweiterungen von g_0 zu einem Homomorphismus, der den Kopf von b auf I_{Q0} abbildet. Da wir keinen finden, führen wir den CHASE-Schritt $I_{Q0} \xrightarrow{b,g_0} I_{Q1}$ durch. Dabei unterscheiden wir b wieder nach TGDs und EGDs. Da im Kopf der TGD existenzquantifizierte Variablen vorkommen, wird g_0 um die Abbildung von ihnen auf für die entartete Instanz neue existenzquantifizierte Variablen erweitert ($g_0(y_{Se}) = y_{Se},\ g_0(y_{No}) = y_{No}$). Wieder gleichen alle hinzugefügten Abbildungen denen aus dem Beispiel für den CHASE auf Anfragen. Bei der Anwendung der TGD im CHASE-Schritt wird I_{Q0} mit Tupeln, die aus der Anwendung von g_0 auf den Ausdrücken aus dem TGD-Kopf stammen, erweitert. Wieder war g_0 unser einziger Trigger und b die einzige Integritätsbedingung. Also wird kein weiterer CHASE-Schritt mehr ausgeführt und wir erhalten die modifizierte entartete Instanz:

$$\begin{aligned} I_{Q1} = \{&\text{STUDENTEN}(x_{Ma}, y_{Na}, y_{Vo}, y_{St}),\\ &\text{TEILNEHMER}(x_{Mo}, x_{Ma}),\\ &\text{NOTEN}(x_{Mo}, x_{Ma}, y_{Se}, y_{No})\}. \end{aligned}$$

Genau wie im Beispiel mit der Instanz oder der Anfrage, gibt es jetzt keinen aktiven Trigger mehr von einer Integritätsbedingung in $\mathcal{B}$ zu I_{Q_1}. Wir erhalten also die entartete Instanz I_{Q1} als Ergebnis des CHASEs. An dieser Stelle sind wir im Gegensatz zu den vorherigen Beispielen aber noch nicht ganz fertig und müssen unser Ergebnis noch richtig interpretieren. Da unsere entartete Instanz aus einer Anfrage entstanden ist, müssen wir I_{Q1} in eine Anfrage zurückformen. Dafür übernehmen wir für jedes Tupel einen Ausdruck konjungiert in unseren Anfragerumpf. Das Tupel STUDENTEN$(x_{Ma}, y_{Na}, y_{Vo}, y_{St})$ wird also zum Beispiel zum Ausdruck STUDENTEN$(x_{Ma}, y_{Na}, y_{Vo}, y_{St})$, bei dem der Name der zugehörigen Relation des Tupels als Bezeichner des Ausdrucks genutzt wird und die Elemente der Position nach übernommen werden. Der Kopf der Anfrage wird dann aus den ausgezeichneten Variablen des Rumpfes gebildet (in unseren Fall die Variablen x_{Ma}, x_{Mo}). Die daraus resultierende Anfrage gleicht dem Ergebnis aus dem Beispiel des CHASEs auf Anfragen, also dem Ergebnis des CHASE$(\{b\}, Q)$.

$$\begin{aligned} Q_{IQ1} = (&\exists y_{Na}, y_{Vo}, y_{St}, y_{Se}, y_{No} : \text{STUDENTEN}(x_{Ma}, y_{Na}, y_{Vo}, y_{St})\\ &\wedge \text{TEILNEHMER}(x_{Mo}, x_{Ma}) \wedge \text{NOTEN}(x_{Mo}, x_{Ma}, y_{Se}, y_{No}))\\ &\longrightarrow x_{Ma}, x_{Mo} \end{aligned}$$

Es wurde folglich mit dem CHASE auf einer entarteten Instanz eine Integritätsbedingung in eine Anfrage eingechaset. Im nächsten Kapitel werden wir dieses Konzept in ChaTEAU implementieren. Neben den Änderungen, die nötig sind, um die CHASE-Algorithmen auf Instanzen und Anfragen zu vereinheitlichen, werden wir außerdem noch weitere Anpassungen erläutern, welche die Funktionalität von ChaTEAU verbessern und erweitern.

5. Implementierung

In diesem Kapitel wird zuerst der allgemeine grobe Ablauf von ChaTEAU sowie ein detaillierter Ablauf der CHASE-Methode erläutert. An Hand des groben Ablaufs lässt sich die im Abschnitt 5.2 erläuterte Einarbeitung des Konzepts besser nachvollziehen. Dort wird beschrieben, wie die Umwandlung einer Anfrage in eine entartete Instanz, die entsprechenden Anpassungen der CHASE-Regeln auf diese entarteten Instanzen und die Rückwandlung der ursprünglichen Anfragen in ChaTEAU umgesetzt wurden. Alle weiteren Änderungen, die die Funktionalität von ChaTEAU korrigiert und erweitert haben, werden gesammelt im Abschnitt 5.3 vorgestellt. Hier hilft ebenfalls der grobe Ablauf sowie die detailliertere Beschreibung der CHASE-Methode bei der Einordnung der Änderungen. Den Abschluss des Kapitels bilden einige ausgewählte Beispiele, die für gegebene Eingaben, die Ausgabe von ChaTEAU und somit das Ergebnis des implementierten CHASEs aufzeigen.

5.1. Allgemeiner Aufbau und Ablauf von ChaTEAU

Grober Ablauf Der allgemeine grobe Ablauf dient zur Einordnung der Konzept-Implementierung in ChaTEAU. Das Schema des beschriebenen groben Ablaufs von ChaTEAU ist in der Abbildung 5.1 dargestellt.

In der `main`-Methode von ChaTEAU wird als Erstes die zu verarbeitende XML-Datei ausgewählt. Eingelesen wird die Datei dann in dem aufgerufenen `InputReader`-Objekt in der `readFile`-Methode. Hier werden neben den Integritätsbedingungen auch das Schema, die Attribute und die im Abschnitt 5.3 näher beschriebenen Schema-Tags eingelesen. Wenn in der Datei eine Instanz definiert wurde, wird aus den eingelesenen Instanz-Atomen (Tupeln zu Relationen) zusammen mit dem Schema, den Attributen und den Schema-Tags, eine Instanz gebildet. Diese Instanz erhält zusätzlich ein `OriginTag i`, welches den Ursprung der entstandenen Instanz als Instanz definiert. Ist in der Datei anstatt einer Instanz eine Anfrage definiert, wird diese eingelesen. Aus dem Rumpf der Anfrage werden dann alle Atome (Ausdrücke) in Instanz-Atome gespeichert, die wiederum zusammen mit dem Schema, den Attributen und den Schema-Tags eine Instanz bilden. Wichtig ist hier, dass der Instanz der `OriginTag q` zugewiesen wird, der den Ursprung der entstandenen Instanz als Anfrage definiert. Die so entstandene Instanz wird zusammen mit den Integritätsbedingungen der `main`-Methode als `SingleInput`-Objekt übergeben.

Über die `main`-Methode wird als Nächstes die `chase`-Methode der `Chase`-Klasse mit der Instanz und den Integritätsbedingungen als Parameter aufgerufen. Nach dem durchgeführten CHASE-Algorithmus (mehr dazu im detaillierteren Ablauf der CHASE-Methode), gibt die `chase`-Methode eine Instanz zurück, welche das Ergebnis des CHASEs enthält.

Bei der Ausgabe wird in der `main`-Methode nach den `OriginTag`s der Instanz unterschieden. Ist die Instanz auch aus einer Instanz entstanden, sprich ist der `OriginTag` gleich `i`, wird die Instanz als XML- und CSV-Datei sowie auf der Konsole ausgegeben. Ist die Instanz wiederum aus einer Anfrage entstanden, ist das `OriginTag` also gleich `q`, dann wird zunächst die `getQuery`-Methode des Instanz-Objektes aufgerufen. Diese liefert eine Anfrage, in der alle Instanz-Atome (Tupel zu Relationen) in den Rumpf der Anfrage geschrieben wurden. Der Kopf der Anfrage wird aus allen ausgezeichneten Variablen des Rumpfes rekonstruiert.

Nach dieser Rückumwandlung der Instanz in eine Anfrage, wird diese ebenfalls als XML- und CSV-Datei sowie auf der Konsole ausgegeben. Unabhängig von der ursprünglichen Art des CHASE-Objektes, wird nach diesen Schritten die Ausführung von ChaTEAU beendet und eine Log-Datei ausgegeben, welche alle abgelaufenen Schritte protokolliert.

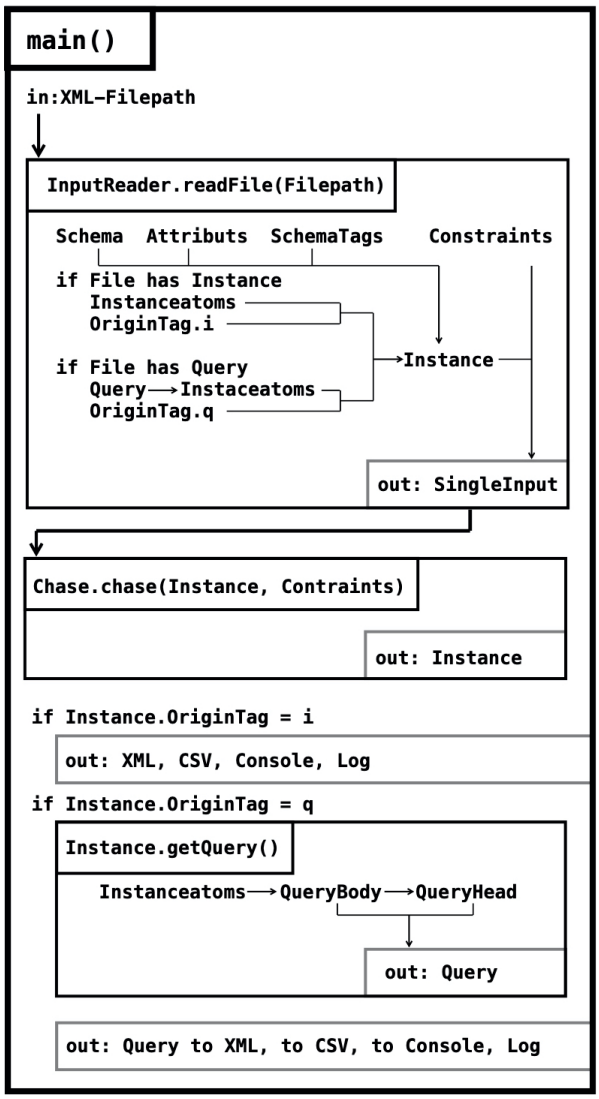

Abbildung 5.1.: Grober Ablauf von ChaTEAU

Die für eine andere Arbeit am Lehrstuhl für Datenbank- und Informationssysteme der Uni Rostock geplante Terminierungsprüfung für den CHASE sowie die BACKCHASE-Phase, können sehr gut in die aktuelle Struktur von ChaTEAU eingebunden werden. Eine Terminierungsprüfung würde nach dem Einlesen der Datei mit dem `SingleInput`-Objekt aufgerufen werden und so vor dem CHASE-Algorithmus stattfinden. Mit dieser Einordnung könnte die Ausführung von ChaTEAU bereits an dieser Stelle mit einer entsprechenden Fehlermeldung enden, wenn eine Terminierung des CHASEs nicht sichergestellt werden kann. Die BACKCHASE-Phase kann direkt an der bisherigen Ausgabe von ChaTEAU anknüpfen und je nach übergebenem CHASE-Objekt verschiedene Verfahren starten. Die Eingabe der BACKCHASE-Objekte und der Integritätsbedingungen kann direkt aus der Ausgabe der CHASE-Phase übernommen werden.

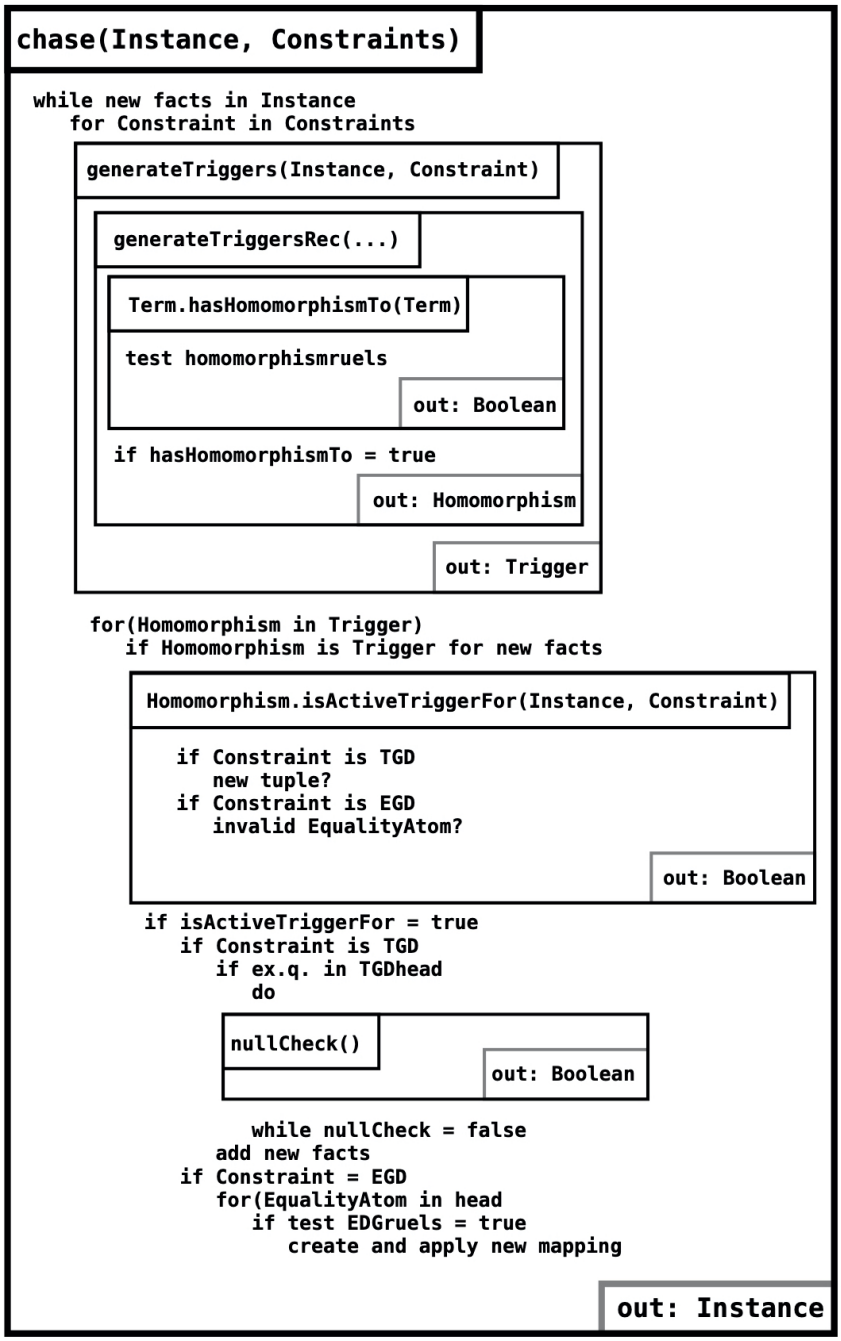

Abbildung 5.2.: Detaillierterer Ablauf der CHASE-Methode

Detaillierterer Ablauf der CHASE-Methode Der detailliertere Ablauf der CHASE-Methode von ChaTEAU hilft vor allem bei der Einordnung der weiteren Anpassungen aus Abschnitt 5.3. Das Schema des beschriebenen detaillierteren Ablaufs ist in Abbildung 5.2 dargestellt.

Wie bereits im groben Ablauf erwähnt, wird die CHASE-Methode der `Chase`-Klasse in der `main`-Methode mit der eingelesenen Instanz bzw. der umgewandelten Anfrage und den Integritätsbedingungen als Parameter aufgerufen. Folgende Schritte werden solange ausgeführt, bis in einem Durchlauf keine neuen Fakten in der Instanz hinzugefügt wurden. Zuerst wird für alle Integritätsbedingungen die `generateTriggers`-Methode aufgerufen, welche mit der `generateTriggersRec`-Methode rekursiv Homomorphismen zwischen der Integritätsbedingung und der Instanz zu Triggern formt und die Menge aller gefundenen Trigger zurückgibt. In der `generateTriggersRec`-Methode erfolgt mit der `hasHomomorphismTo`-Methode der `Term`-Klasse unter anderem auch die Überprüfung der Regeln, die wir für Homomorphismen zwischen

zwei CHASE-Objekten definiert haben (siehe Definition 2.8). Als Nächstes wird für alle Trigger geprüft, ob sie auch Trigger auf den neuen Fakten sind. Dieser Schritt verhindert die Prüfung auf aktive Trigger für alle gefundenen Trigger und optimiert somit die Ausführungszeit. Für die Trigger auf den neuen Fakten wird mit der `isActiveTrigger`-Methode des `Homomorphismus`-Objekts (Trigger) geprüft, ob im Falle einer TGD neue Tupel entstehen würden und im Falle einer EGD Gleichheitsatome verletzt werden würden. Ist das der Fall, handelt es sich tatsächlich um einen aktiven Trigger und wir unterscheiden die Integritätsbedingungen nach TGDs und EGDs. Ist die Integritätsbedingung eine TGD, werden die neuen Tupel entsprechend den Atomen aus dem Kopf der Integritätsbedingung erstellt. Sollten hierbei neue Nullwerte oder neue existenzquantifizierte Variablen entstehen, wird mit der `nullCheck`-Methode geprüft, ob die neuen Elemente wirklich neu in der Instanz sind. Ist dies der Fall, werden die neuen Tupel hinzugefügt; wenn nicht, dann wird die `nullCheck`-Methode mit neuen Nullwerten bzw. existenzquantifizierten Variablen aufgerufen. Ist die Integritätsbedingung eine EGD, werden für jedes Gleichheitsatom im Kopf die EGD-Regeln geprüft und gegebenen Falls neue Abbildungen zwischen der Instanz und der Instanz im nächsten CHASE-Schritt erstellt und direkt auf die Instanz angewandt. Ist dies alles geschehen, kann ChaTEAU in einen neuen Schleifendurchlauf starten. Die Rückgabe der CHASE-Methode ist die Instanz, in der die Integritätsbedingungen eingechaset wurden, welche dann in der `main`-Methode weiter verarbeitet werden kann.

Im folgenden Abschnitt wird die Implementierung des Konzepts aus Kapitel 4 beschrieben. Für die Einordnung der Vereinheitlichung der CHASE-Objekte aus Abschnitt 4.2.1, sprich, die Um- und Rückumwandlung der Anfrage in eine entartete Instanz (siehe Abschnitt 5.2.1 und 5.2.3), hilft der eben beschriebene grobe Ablauf von ChaTEAU. Zum Verständnis der Angleichung der CHASE-Schritte aus Abschnitt 5.2.2, sprich, die Anpassung der Regeln (siehe Abschnitt 5.2.2), hilft der detaillierte Ablauf der CHASE-Methode.

5.2. Konzept-Implementierung

In diesem Abschnitt wird aufgezeigt, wie das Konzept dieser Arbeit (siehe Kapitel 4) in ChaTEAU implementiert wurde. Zur Rekapitulation: Ziel war es, die beiden CHASE-Objekte Instanz und Anfrage zu vereinheitlichen und in einer allgemeinen CHASE-Methode zu verarbeiten. Mithilfe der Umwandlung der Anfragen in Instanzen, haben wir unser einheitliches CHASE-Objekt erzeugt – die entartete Instanz. Da wir nur noch entartete Instanzen für Instanzen und Anfragen an unsere CHASE-Methode übergeben, entspricht diese Methode nun unserer geforderten universalen CHASE-Methode `chase(CHASE-Objekt o)`, mit `CHASE-Objekt o` ≡ `entartete Instanz`.

5.2.1. Umwandlung Anfrage in Instanz

Wie Anfragen in Instanzen umgewandelt werden, wurde schon im groben Ablauf in Abschnitt 5.1 beschrieben. Kurz zusammen gefasst: Es werden in der `readFile()`-Methode des `InputReader`s alle Atome des Anfragerumpfes in ein `HashSet<>` aus Relationalenatomen gespeichert, welches als Instanz-Atome neben den zuvor eingelesenen Attributen, dem Schema, den Origin-Tag q und den Schema-Tags dem Konstruktor der Instanz-Klasse übergeben wird.

```
1 Query query = this.getQuery(queryElement);
2 HashSet<RelationalAtom> instanceAtoms = new HashSet<RelationalAtom>();
3 for(RelationalAtom queryAtom: query.getBody()) {
4     instanceAtoms.add(queryAtom);
5 }
6 Instance instance =
7 new Instance(instanceAtoms, attributes, schema, OriginTag.q, schemaTags);
```

Das `OriginTag` ist eine Enumeration mit den Werten `i` und `q`, welches in der Instanz als private Variable gespeichert wird. Es kann jederzeit mittels `getOriginTag()` ausgelesen werden und kennzeichnet somit den Ursprung der Instanz als Instanz (`OriginTag = i`) oder Anfrage (`OriginTag = q`). Das `OriginTag` spielt dort eine Rolle, wo die CHASE-Objekte unterscheidbar bleiben müssen.

5.2.2. Anpassung der Regeln

Anpassungen beim Finden der Trigger Wie in [Ren19] von Fabian Renn in den offenen Problemen beschrieben wurde, werden Trigger für gegebene Integritätsbedingungen bisher nur dann gefunden, wenn die Variablen im Rumpf der Anfrage identisch zu den Variablen im Rumpf der Integritätsbedingung benannt wurden. Die Ursache dafür ist die Prüfung der Homomorphismen-Regeln (siehe Definition 2.8) in der Erstellung der Trigger in der `generateTriggersRec`-Methode. Die `hasHomomorphismTo`-Methode, die von den Termen der Atome der Integritätsbedingungen mit den Termen aus den Instanz-Atomen aufgerufen wird, soll für die CHASE-Objekte zwischen einem CHASE-Schritt sicherstellen, dass die Abbildungsregeln zwischen ihnen eingehalten werden. Diese Regeln verhindern unter anderem die Abbildung von ausgezeichneten Variablen auf existenzquantifizierte Variablen sowie die Abbildung von Variablen auf Nullwerte. Wenn man sich die Regeln anschaut, ist das auch korrekt so – existenzquantifizierte Variablen dürfen z. B. auf alles außer auf Nullwerte abgebildet werden. Das Problem ist, dass diese Regeln auch für die Abbildungen zwischen Atomen aus den Integritätsbedingungen und Objektatomen geprüft wurden. Somit wurden z. B. Abbildungen von Variablen aus Integritätsbedingungen auf Nullwerte aus Instanzen nicht zugelassen. Ebenso verhinderte die Prüfung der Regeln die Abbildung von den allquantifizierten

Variablen aus den Integritätsbedingungen auf die Variablen der aus einer Anfrage entstandenen entarteten Instanz, wenn diese nicht gleich benannt waren. [1]

Um die `hasHomomorphismTo`-Methode nur noch für Terme zweier CHASE-Objekte und nicht mehr für Terme aus Atomen der Integritätsbedingung und denen der CHASE-Objekte aufzurufen, muss vorher geprüft werden, ob die Terme der Integritätsbedingungsatome bereits auf Terme aus dem CHASE-Objekt abgebildet werden. Dafür werden wir, nachdem sicher gestellt wurde, dass die Atome über der gleichen Relation definiert sind und die gleiche Anzahl an Termen haben, für jede bereits bestehende Abbildung im bisher erstellten Trigger prüfen, ob die Abbildungsquelle auf einen Term der Integritätsbedingungsatome passt. Ist das der Fall, sprich, wird der Term auf einen Term des CHASE-Objektes abgebildet, kann für den Ziel-Term der Abbildung (wird durch die `apply`-Methode der Termabbildung zurückgegeben) die `hasHomomorphismTo`-Methode mit dem zugehörigen Term des CHASE-Objektatoms aufgerufen werden. Gibt die Methode für einen Term `false` zurück, kann der Schleifendurchlauf beendet werden, da kein valider Homomorphismus für den Trigger gebildet werden kann. Ist die `boolean` Variable `hasHomomorphismTo` nach den Schleifendurchläufen `true`, dann kann der Homomorphismus erstellt werden und dem Trigger hinzugefügt werden.

```
1  boolean hasHomomorphismTo = false;
2  if (constraintAtom.getRelationName().equals(objectAtom.getRelationName())
3  && constraintAtom.getTerms().size() == objectAtom.getTerms().size()){
4      hasHomomorphismTo = true;
5      for(TermMapping th : h.getTermMappings()){
6          for(int i = 0; i < constraintAtom.getTerms().size(); i++){
7              if(th.getMappingSource().equals(constraintAtom.getTerms().get(i))){
8                  hasHomomorphismTo = th.apply(constraintAtom.getTerms().get(i)).
9                  hasHomomorphismTo(objectAtom.getTerms().get(i));
10                 if(!hasHomomorphismTo){
11                     break;
12         }}}
13         if(!hasHomomorphismTo){
14             break;
15 }}}
16 if (hasHomomorphismTo){
17     //generate Homomorphism with new mapping
18 }
```

Mit dieser vorangestellten Prüfung kann sicher gestellt werden, dass jede valide Abbildung von Atomen der Integritätsbedingungen auf Atome der entarteten Instanz gefunden wird. Die Korrektur bei der Erstellung der Trigger hat ebenfalls die Terminierungseigenschaften von ChaTEAU repariert, welche wichtig für die am DBIS Lehrstuhl der Uni Rostock laufende Masterarbeit zur Terminierungsprüfung sind. ChaTEAU verhält sich nun wie gewollt und terminiert nicht mehr bei zyklischen Strukturen in Verbindung mit nicht schwach azyklischen TGDs.

Zusätzlich zu der Prüfung der Homomorphismen-Regeln musste in der `generateTriggersRec`-Methode vor der Generierung der Homomorphismen zwischen Termen der Integritätsbedingungsatome und Termen der CHASE-Objektatome noch geprüft werden, ob bereits eine Abbildung mit einem anderen Ziel-Term eines CHASE-Objektatoms existiert. Hierfür wird für jede Abbildung in dem bereits erstellten Trigger geprüft, ob die Ziele der Abbildungen auf den CHASE-Objektatom-Term passen, falls die Quelle der Abbildung mit dem Term des Integritätsbedingung-Atoms übereinstimmt. Ohne diese Überprüfung wurden in manchen Fällen Trigger gefunden, die es auf den Instanzen nicht geben dürfte.

[1] Da Anfragen von TGDs abgeleitet werden, gibt es in ChaTEAU keine Unterscheidung zwischen allquantifizierten und ausgezeichneten Variablen, daher werden sie auch in `hasHomomorphismTo()` genau so überprüft wie zwei ausgezeichnete Variablen.

```
if (hasHomomorphismTo){
    boolean temp = true;
    ArrayList<Term> sourceTerms = constraintAtom.getTerms();
    ArrayList<Term> targetTerms = objectAtom.getTerms();
    for(TermMapping termMapping : h.getTermMappings()){
        for(int i = 0; i < sourceTerms.size(); i++){
            if(termMapping.getMappingSource().equals(sourceTerms.get(i))){
                if(!termMapping.getMappingTarget().equals(targetTerms.get(i))){
                    temp = false;
    }}}}
    if(temp){
        //generate Homomorphism with new mapping
}}
```

Ein weiteres Problem der `generateTriggersRec`-Methode war die Abbildung zweier Atome der Integritätsbedingung auf ein CHASE-Objektatom. So konnten Trigger entstehen, bei denen zwei Ausdrücke einer Integritätsbedingung, die über derselben Relation definiert sind, auf nur ein Atom (Tupel) der Instanz abgebildet wurden. Als Lösung wurde das CHASE-Objektatom nach Erstellung der Abbildung und vor dem nächsten rekursiven Aufruf der `generateTriggersRec`-Methode aus der Menge der CHASE-Objektatome vorübergehend entfernt.

Anpassungen beim Einchasen von Integritätsbedingungen Neben den Anpassungen an den Triggern, musste auch das Einchasen der Integritätsbedingungen an die Variablen in den entarteten Instanzen angepasst werden. Im Falle eines aktiven Triggers für eine TGD muss zwischen ursprünglichen Instanzen und Anfragen unterschieden werden, sobald existenzquantifizierte Variablen im TGD-Kopf existieren. Hier muss die existenzquantifizierte Variable des TGD-Kopfes im Falle einer ursprünglichen Instanz auf einen neuen Nullwert und im Falle einer ursprünglichen Anfrage auf eine neue existenzquantifizierte Variable abgebildet werden.

```
if (oldVariable.getVariableType() == VariableType.E){
    if(I.getOriginTag() == OriginTag.i){
        Null newNull = new Null(...);
    }
    else if(I.getOriginTag() == OriginTag.q){
        Variable newVariable = new Variable(VariableType.E, ...);
}}
```

Ist die Integritätsbedingung, für die der aktive Trigger gefunden wurde, eine EGD, muss in ChaTEAU nichts weiter verändert werden. Die bereits existierenden Methoden `getBiggerTerm` bzw. `getSmallerTerm` erstellen von vornherein die richtigen Abbildungen zwischen den CHASE-Objekten, sodass eine Übernahme der EGD-Regel für Anfragen nicht nötig ist.

5.2.3. Rückwandlung Instanz in Anfrage

Nach dem erfolgreichen einchasen der Integritätsbedingungen in eine Instanz, die aus einer Anfrage entstanden ist, müssen wir diese noch zurück wandeln. In Abschnitt 5.1 wurde bereits erwähnt, dass hierfür die `getQuery`-Methode des Instanz-Objektes aufgerufen wird. Dabei wird jedes relationale Atom der Instanz in den Anfragerumpf kopiert und jede ausgezeichnete Variable aus den Atomen in den Anfragekopf geschrieben.

```
1      HashSet<RelationalAtom> queryBody = new HashSet<RelationalAtom>();
2      HashSet<RelationalAtom> queryHead = new HashSet<RelationalAtom>();
3      ArrayList<Term> queryHeadTerms = new ArrayList<Term>();
4      for(RelationalAtom atom : this.getRelationalAtoms()){
5          queryBody.add(atom);
6          for(Term term : atom.getTerms()){
7              if(term.getTermType() == TermType.Variable){
8                  if(term.getTermValueVariable().
9                      getVariableType() == VariableType.V){
10                     if(!queryHeadTerms.contains(term)){
11                         queryHeadTerms.add(term);
12     }}}}}
13     RelationalAtom queryHeadAtom =
14         new RelationalAtom("queryHead", queryHeadTerms);
15     queryHead.add(queryHeadAtom);
16     return new Query(queryBody, queryHead);
17 }
```

Für den Fall, dass der CHASE fehlschlägt, weicht die Implementierung von der Beschreibung im Konzept (siehe Kapitel 4) ab. Hier wird nicht wie beschrieben nach dem Fehlschlag $\perp$ zurückgegeben und dann das Ergebnis gemäß der Herkunft interpretiert, sondern bereits im CHASE-Algorithmus je nach Origin-Tag eine spezifische Fehlermeldung ausgegeben und dann die leere Instanz übergeben. Der Grund dafür ist bereits im Konzept von [Jur18] beschrieben worden, in dem designtechnisch die Entscheidung getroffen wurde, dass die leere Instanz zurückgegeben wird, anstatt den Fehlschlag explizit zu deklarieren. So kann der Fehlschlag nach dem CHASE nicht mehr von der leeren Instanz als CHASE-Ergebnis unterschieden werden, und muss wie beschrieben in der CHASE-Methode gehandhabt werden. Diese Lösung wurde vom Autor selbst in Frage gestellt. Da das Umgehen mit dem Fehlschlag auch in dieser Arbeit nicht verbessert wurde, ist es eventuell an späteren Arbeiten einen besseren Ansatz zu finden.

5.3. Sonstige Anpassungen

Während der Arbeit an der Vereinheitlichung der CHASE-Objekte und der CHASE-Schritte sind immer mal wieder Probleme ans Licht gekommen, durch die die Funktionalität von ChaTEAU eingeschränkt wurde oder wegen denen der CHASE inkorrekte Ergebnisse lieferte. In diesem Abschnitt werden alle Anpassungen und Erweiterungen an ChaTEAU beschrieben, die nicht direkt mit dem Konzept in Verbindung stehen, die aber zum Teil notwendig waren, um es korrekt umzusetzen.

ST TGDs Rein formal war es bisher nur möglich TGDs und EGDs als Integritätsbedingungen zu formulieren. Um die angestrebten Fälle III und IV aus der Tabelle 1.1 vollends umzusetzen, fehlt noch die Möglichkeit, s-t TGDs als CHASE-Parameter zu übergeben. Diese Erweiterung ist nicht Bestandteil der Aufgabenstellung und war ursprünglich für eine spätere Arbeit gedacht. Bei der Bearbeitung konnte die Unterstützung für s-t TGDs aber relativ einfach hinzugefügt werden.

Da s-t TGDs TGDs mit Rumpf über einem Quellschema und Kopf über einem Zielschema sind, kann man s-t TGDs von der existierenden TGD-Klasse ableiten:

```
1 public class STTgd extends Tgd{
2     public STTgd(HashSet<RelationalAtom> body, HashSet<RelationalAtom> head){
3         super(body, head);
4 }}
```

Eingeben lässt sich eine s-t TGD genau wie eine TGD. Man muss nur beachten, dass die Relationen bei Eingabe einer s-t TGD mit einem `tag` gekennzeichnet werden müssen. Dieser gibt an, ob die Relation aus einem Quellschema (S für *Source*) oder einem Zielschema (T für *Target*) kommt.

```
1  <schema>
2      <relations>
3          <relation name="Studenten" tag="S"/>
4          <relation name="Teilnehmer" tag="S"/>
5          <relation name="Noten" tag="T"/>
6      </relations>
7      <dependencies>
8          <sttgd>
9              <body>
10                 <atom name="Studenten"/>
11                 <atom name="Teilnehmer"/>
12             </body>
13             <head>
14                 <atom name="Noten"/>
15             </head>
16         </sttgd>
17     </dependencies>
18 </schema>
```

Die `tag`s werden für jede Relation zu einer `HashMap<String, SchemaTag>` hinzugefügt, welche dann in der eingelesenen Instanz gespeichert wird. Der `String` stellt den Namen der Relation da. Das `SchemaTag` ist eine Enumeration mit den Werten `S` und `T`. Vor dem CHASE muss nur noch geprüft werden, ob der Rumpf einer s-t TGD nur über dem Quellschema und der Kopf nur über dem Zielschema definiert ist. Zu diesem Zweck wurde der Klasse `Instance` eine Methode hinzugefügt, die anhand des Schemas einer Instanz obige Bedingung prüft.

Die Methode `ConstraintCheck` der `Instance`-Klasse überprüft jede s-t TGD von einer übergebenen Menge aus Integritätsbedingungen. Falls es eine s-t TGD gibt, die Instanz aber keine Schema-Tags hat, dann gibt die Methode `false` zurück. Das gleiche geschieht, wenn der Name eines Atoms aus dem Rumpf der s-t TGD auf ein `T` oder der Name eines Atoms aus dem Kopf der s-t TGD auf ein `S SchemaTag` abgebildet wird. Falls die s-t TGD korrekt über den Relationen definiert wurde, kann sie durch eine äquivalente TGD ersetzt werden. Der CHASE erhält dann eine Kopie der Integritätsbedingungen, die nur aus TGDs und EGDs bestehen. Die originalen Integritätsbedingungen bleiben in einer Kopie erhalten.

```
1  public enum SchemaTag{
2      S,
3      T;
4  }
5
6  public class Instance implements ChaseObject{
7      private HashMap<String,SchemaTag> schemaTags;
8      ...
9      public boolean ConstraintCheck(Set<IntegrityConstraint> constraints){
10         boolean constraintCheck = true;
11         for(int i = 0; i < constraints.size(); i++){
12             IntegrityConstraint constraint = constraints.iterator().next();
13             if (constraint instanceof STTgd){
14                 if(!this.schemaTags.isEmpty()){
15                     STTgd sttgd = (STTgd) constraint;
16                     for(RelationalAtom bodyAtom : sttgd.getBody()){
17                         if(schemaTags.get(bodyAtom.getName()) == SchemaTag.T){
18                             constraintCheck = false;
19                     }}
20                     for(RelationalAtom headAtom : sttgd.getHead()){
21                         if(schemaTags.get(headAtom.getName()) == SchemaTag.S){
22                             constraintCheck = false;
23                     }}
24                     if(constraintCheck){
25                         Tgd oldSTTGD = new Tgd(sttgd.getBody(),
26                             sttgd.getHead());
27                         constraints.add(oldSTTGD);
28                         constraints.remove(constraint);
29                 }}
30                 else {
31                     constraintCheck = false;
32         }}}
33         return constraintCheck;
34 }}
```

Mit der Erweiterung der möglichen CHASE-Parameter um die s-t TGD wurde gleichzeitig auch die CHASE-Phase des Falls II – Answering Queries Using Views – aus der Tabelle 1.1 in ChaTEAU implementiert. In der CHASE-Phase von AQuV werden Sichten (Views) in Anfragen eingechaset. Da sich Sichten als s-t TGDs schreiben lassen, entspricht das Erweitern der Anfrage um den Kopf der s-t TGD genau dem Erweitern um die Sichten. Somit fehlen nur noch die Operationen im Fall II', um alle Parameter aus der Tabelle 1.1 abzudecken. Die Abhängigkeiten in den Fällen 0 und I, wie JDs oder FDs, lassen sich jeweils als TGDs oder EGDs schreiben. Die TGDs, EGDs und s-t TGDs der Fälle III bis VI werden im aktuellen Stand nativ unterstützt.

Ein- und Ausgabe Die Ein- und Ausgabe von ChaTEAU wurde zum Großteil in [Ren19] realisiert. In dieser Arbeit wurde die Ausgabe der Anfragen äquivalent zu der Ausgabe der Instanzen als CSV- und XML-Datei implementiert. Hinzu kamen weitere Verbesserungen und Bug Fixes in der Eingabe. So konnte ChaTEAU z. B. keine Nullwerte für Attribute des Typs `INT` einlesen. Ebenfalls haben wir uns in dieser Arbeit dazu entschlossen, die Integritätsbedingungen in der Reihenfolge einzulesen und zu verarbeiten, in der sie in der Eingabe definiert wurden. Ursprünglich wurden TGDs vor EGDs eingelesen und in eine nicht geordnete Menge geschrieben. Diese nicht geordnete Menge entsprach zwar dem theoretisch richtigen nicht deterministischen Auswählen der Integritätsbedingungen, führte aber in der Praxis eben auch zu nicht deterministischen Ergebnissen. So wurde das Einlesen der Integritätsbedingungen positionsabhängig und die Speicherung in eine ordnungsbasierende Struktur (`LinkedHashSet`) realisiert.

EGDs reparieren Wie bereits in Abschnitt 3.1 angesprochen, konnte ChaTEAU im Stand vor dieser Arbeit noch keine EGDs anwenden. Die Termabbildungen wurden zwar korrekt erstellt, aber nicht zu dem Homomorphismus hinzugefügt, der am Ende eines CHASE-Schrittes die Instanz aktualisiert. Nach der Korrektur ist aufgefallen, dass die Aktualisierung der Instanz direkt nach der Erstellung der Termabbildungen sinnvoller wäre, da ansonsten folgende Integritätsbedingungen immer auf der alten Instanz angewandt werden und am Ende nur die Ergebnisse zusammen gemischt werden. So wurden TGDs in ChaTEAU immer vor den EGDs auf die Instanz angewandt. Das führte in nicht konfluenten Kombinationen von TGDs und EGDs dazu, dass ein falsches Ergebnis erzeugt wurde. Ein Beispiel dazu ist im Abschnitt 5.4 unter dem Beispiel B.2.2 zu finden.

Als letzte Änderung an den EGDs musste noch eine Prüfung auf Ungleichheit zwischen zwei Konstanten hinzugefügt werden. Diese wurde, wie in [Jur18] begründet, absichtlich weggelassen, da eine Prüfung bereits bei der Erstellung der aktiven Trigger durchgeführt wird, und so an dieser Stelle ein Test auf den Term-Typ gleich Konstante ausreiche. Das Problem an dieser Entscheidung offenbarte sich auch erst in der Fehlermeldung vom abgebrochenen CHASE und auch nur dann, wenn vor den ungleichen Konstanten noch gleiche Konstanten überprüft wurden. In einigen Fällen kam es vor, dass der CHASE korrekter Weise abgebrochen wurde, in der Ausgabe dann aber zwei gleiche Konstanten als in Konflikt stehend genannt wurden.

Neue Null auf Vorhandensein prüfen In [Jur18] wurde bereits beschrieben, wie der `nullCounter` in ChaTEAU genutzt wird, um bei der Anwendung von TGDs existenzquantifizierte Variablen durch neue Nullwerte zu ersetzen. Der `nullCounter` wurde dafür pro Attribut hochgezählt und so immer ein Nullwert mit höherem Index erzeugt. Nicht beachtet wurden allerdings Nullwerte, die bereits in der Instanz vorhanden waren, sodass Gleichheiten zwischen Nullwerten entstanden, wo sie nicht gegeben waren. Als Lösung wurde die `nullCheck`-Methode in der CHASE-Klasse implementiert, die genutzt wird, um solange neue Nullwerte zu erzeugen, bis diese wirklich neu für die Instanz sind.

```
1 do
2 {
3     nullCounter.replace(oldVariable.getIndexName(),
4     nullCounter.get(oldVariable.getIndexName()) + 1);
5 }
6 while(!nullCheck(I, nullCounter, oldVariable.getIndexName()));
```

Die `nullCheck`-Methode wird mit der aktuellen Instanz, dem `nullCounter` und dem Attributnamen, für welches der Nullwert erzeugt werden soll, aufgerufen. Da verschiedene Relationen die gleichen Attribute und somit auch Nullwerte beinhalten können, suchen wir als Erstes im Schema nach den Relationen, die das Attribut enthalten, sowie nach der Position des Attributs. Diese Informationen speichern wir in eine `HashMap<String, Integer>`. Da die Atome und Terme an sich keine Angaben zu ihren zugehörigen Attributen speichern, erspart man sich mit dieser Vorbereitung das Durchlaufen aller Tupel der Instanz. Nun können wir direkt auf die Terme zugreifen, die auch wirklich die relevanten Nullwerte speichern können. Diese werden dann nur noch nach ihrem Origin-Tag unterschieden, um auf den Index des möglichen Nullwertes oder der möglichen existenzquantifizierten Variable zuzugreifen. Ist der Index bereits vergeben, wird `false` zurückgegeben und die `nullCheck`-Methode mit einem neuen `nullCounter` aufgerufen.

```
1  private static boolean nullCheck(Instance I,
2      HashMap<String,Integer> nullCounter, String attribut){
3      int index = nullCounter.get(attribut);
4      HashMap<String,Integer> relationAttPosition = new HashMap<String,Integer>();
5      Iterator iterSchema = I.getSchema().entrySet().iterator();
6      while(iterSchema.hasNext()){
7          Entry pairSchema  = (Entry)iterSchema.next();
8          ArrayList<String> att = (ArrayList<String>) pairSchema.getValue();
9          for(int i = 0; i<att.size(); i++){
10             if(att.get(i).equals(attribut)){
11                 relationAttPosition.put((String) pairSchema.getKey(), i);
12     }}}
13     Iterator iterRelationAttPos = relationAttPosition.entrySet().iterator();
14     while(iterRelationAttPos.hasNext()){
15         Entry pairRelationAttPos = (Entry)iterRelationAttPos.next();
16         for(RelationalAtom atom : I.getRelationalAtomsBySchema(
17             (String)pairRelationAttPos.getKey())){
18             Term term = atom.getTerms().get(
19             (Integer)pairRelationAttPos.getValue());
20             if(I.getOriginTag()==OriginTag.i){
21                 if(term.getTermType() == TermType.Null){
22                     if(term.getTermValueNull().getIndex() == index){
23                         return false;
24             }}}
25             else{
26                 if(term.getTermType() == TermType.Variable){
27                     if(term.getTermValueVariable().
28                         getVariableType() == VariableType.E){
29                         if(term.getTermValueVariable().getIndex() == index){
30                             return false;
31     }}}}}}
32     return true;
33 }
```

Anfrage-Prüfung Da wir den CHASE nur auf konjunktiven Anfragen definiert haben, muss die Konjunktivität in groben Ansätzen auch vor dem CHASE-Algorithmus überprüft werden. Hier haben wir uns dafür entschieden, inkorrekt formulierte Anfragen zum Teil richtig zu stellen und zum Teil nur anzumerken, den CHASE aber trotzdem auszuführen. So erkennt die Methode `queryHeadCheck` ob alle ausgezeichneten Variablen des Anfragekopfes im Rumpf der Anfrage stehen sowie ob alle ausgezeichneten Variablen des Anfragerumpfes auch im Anfragekopf wieder zu finden sind. Es wird außerdem überprüft, ob sich im Anfragekopf nur ausgezeichnete Variablen befinden. Die Auswertung der Prüfung erfolgt im `InputReader`, welcher bei fehlerhaften Eingaben eine Fehlermeldung ausgibt. Der Anfragekopf wird in jedem Fall angepasst, da die `getQuery`-Methode der Instanz den Kopf immer richtig aus dem Rumpf konstruiert.

```
1  public boolean queryHeadCheck(){
2      HashSet<RelationalAtom> queryHead = this.getHead();
3      HashSet<RelationalAtom> queryBody = this.getBody();
4      //Check ob alle #V des Bodys auch im Head vorkommen
5      for(RelationalAtom atomBody : queryBody){
6          for(Term termBody : atomBody.getTerms()){
7              if(termBody.getTermType() == TermType.Variable){
8                  if(termBody.getTermValueVariable().getVariableType()
9                      == VariableType.V){
10                     for(RelationalAtom atomHead : queryHead){
11                         if(!atomHead.getTerms().contains(termBody)){
12                             return false;
13     }}}}}}
14     //Check ob alle #V des Heads auch im Body vorkommen
15     boolean contained;
16     for(RelationalAtom atomHead : queryHead){
17         for(Term termHead : atomHead.getTerms()){
18             contained = false;
19             for(RelationalAtom atomBody : queryBody){
20                 if(atomBody.getTerms().contains(termHead)){
21                     contained = true;
22             }}
23             if(!contained){
24                 return false;
25     }}}
26     //Check ob im Head nur #V stehen
27     for(RelationalAtom atomHead : queryHead){
28         for(Term termHead : atomHead.getTerms()){
29             if(termHead.getTermType() != TermType.Variable){
30                 return false;
31             }
32             else if(termHead.getTermValueVariable().getVariableType()
33                 != VariableType.V){
34                 return false;
35     }}}
36     return true;
37 }
```

Ebenfalls im `InputReader` erfolgt ein Test mittels der `queryCheck`-Methode. Sie wird bereits für die aus der `getQuery`-Methode entstandenen Anfrage mit dem Schema der Instanz als Parameter aufgerufen und überprüft die Exklusivität der ausgezeichneten Variablen für die jeweiligen Attribute. Diese Einschränkung stammt aus dem Ursprung des CHASEs auf *Tableaus* und fordert eindeutige ausgezeichnete Variablen pro Attribut, unabhängig der Relation. Noch allgemeiner formuliert bedeutet das, dass es keine unterschiedlichen ausgezeichneten Variablen zu einem Attribut geben darf, auch dann nicht, wenn die Attribute zu verschiedenen Relationen gehören. Sollte der Test `false` zurückliefern, wird eine entsprechende Warnung ausgegeben. Aus Test-Zwecken wird der CHASE aber dennoch ausgeführt.

```
1  public boolean queryCheck(HashMap<String,ArrayList<String>> schema){
2      HashSet<RelationalAtom> queryBody = this.getBody();
3      //Hashmap um den Attributen eindeutig ihre #V zuzuordnen
4      HashMap<String,String> attributVmap = new HashMap<String,String>();
5      for(RelationalAtom atomBody : queryBody){
6          int i = 0;
7          for(Term termBody : atomBody.getTerms()){
8              if(termBody.getTermType() == TermType.Variable){
9                  if(termBody.getTermValueVariable().getVariableType()
10                     == VariableType.V){
11                     if(attributVmap.containsKey(schema.
12                         get(atomBody.getName()).get(i))){
13                         if(!attributVmap.get(schema.get(atomBody.getName())
14                             .get(i)).equals(termBody.
15                             getTermValueVariable().toString())){
16                             return false;
17                     }}
18                     else{
19                         attributVmap.put(schema.get(atomBody.getName()).get(i),
20                             termBody.getTermValueVariable().toString());
21             }}}
22             i++;
23     }}
24     return true;
25 }}
```

Aktive Trigger für TGDs verhindern, wenn Abbildung auf existierendes Tupel möglich Eine der wichtigsten sonstigen Anpassungen in diesem Abschnitt ist die Unterdrückung von aktiven Triggern für TGDs, wenn diese Tupel erzeugen würden, die auf Tupel der Instanz abgebildet werden könnten. Ein Beispiel ist eine TGD, die ein Tupel einfügen würde, welches sich ausschließlich in den Nullwerten von einem bereits existierenden Tupel unterscheidet, bzw. anstatt einer Konstanten einen Nullwert enthält. Äquivalente Beispiele sind Atome in Anfragen, die anstatt ausgezeichneter Variablen existenzquantifizierte Variablen haben, bzw. anstatt einer Konstanten eine ausgezeichnete oder existenzquantifizierte Variable haben oder sich nur in den existenzquantifizierten Variablen unterscheiden.

Das Problem ist entstanden, da bei der Überprüfung eines aktiven Triggers in der `isActiveTriggerFor`-Methode einfach in der Instanz nach dem neuen Tupel gesucht wurde und der aktive Trigger als valide galt, sobald es keine Übereinstimmung gab. Als Lösung wird jetzt anstatt der Standard-`contains`-Methode für `HashSets`, eine eigene `isContained`-Methode verwendet. Diese vergleicht die Tupel einer Instanz mit einem Tupel, wobei sie die möglichen Abbildungen von Nullwerten auf andere Nullwerte und Konstanten, bzw. Abbildungen von existenzquantifizierten Variablen auf andere existenzquantifizierte- bzw. ausgezeichnete Variablen und existenzquantifizierte bzw. ausgezeichnete Variablen auf Konstanten beachtet.

```
public boolean isContained(Instance instance, RelationalAtom headAtom){
    boolean isContained = false;
    for(RelationalAtom instanceAtom : instance.getRelationalAtomsBySchema(
        headAtom.getName())){
        for(int i = headAtom.getTerms().size() - 1; i >= 0 ; i--){
            isContained = true;
            if(!instanceAtom.getTerms().get(i).equals(headAtom.
                getTerms().get(i))){
                if(headAtom.getTerms().get(i).
                    getTermType() == TermType.Variable){
                    if(headAtom.getTerms().get(i).getTermValueVariable().
                        getVariableType() == VariableType.E){
                        continue;
                    }
                    else if(headAtom.getTerms().get(i).getTermValueVariable().
                        getVariableType() == VariableType.V){
                        if(instanceAtom.getTerms().get(i).
                            getTermType() == TermType.Const){
                        continue;
                }}}
                isContained = false;
                break;
        }}
        if(isContained){
            return true;
    }}
    return false;
}
```

Als Parameter erhält die Methode die Instanz sowie das Atom aus dem Kopf der TGD, auf welches die Termabbildung des Triggers bereits angewandt wurde. Nun werden für alle Atome der zugehörigen Relation alle Terme positionsabhängig mit den Termen des neuen Tupels verglichen. Sind zwei Terme ungleich, wird nicht wie zuvor gleich entschieden, dass die Atome ungleich sind, sondern es wird erst geprüft, ob die Ungleichheit durch eine existenzquantifizierte Variable aus dem neuen Tupel hervorgerufen wurde bzw. ob eine ausgezeichnete Variable auf eine Konstante trifft. Diese Fälle entsprechen zwei Atomen, die sich nach dem Erzeugen der neuen Nullwerte bzw. neuen existenzquantifizierten Variablen beim Anwenden der TGD, aufeinander abbilden lassen könnten.

Mit dieser Anpassung kann man sagen, dass ChaTEAU jetzt wirklich den Standard-CHASE nach den neusten Definitionen unterstützt. Anstatt aktive Trigger für TGDs nur auf Grundlage der Mengeneigenschaft der Instanz zu überprüfen, werden nun auch die Homomorphismen zwischen den einzelnen Tupeln beachtet. Als Ergebnis wurden vor allem die Lösungen der Benchmark-Anfragen aus den Beispielen im Anhang B dieser Arbeit verbessert. Bei ihnen wurden vorher oft zu viele Atome in die Anfragen eingechaset und die Anfragen so mit zu vielen Atomen aufgebläht. Was es genau mit diesen Benchmark-Anfragen auf sich hat, wird im nächsten Abschnitt beschrieben.

5.4. Laufende Beispiele

In diesem Abschnitt werden kurz die Beispiele aus dem Anhang B dieser Arbeit erläutert. Die Beispiele sollen zeigen, wie ChaTEAU verschiedene Kombinationen von Integritätsbedingungen in Instanzen und Anfragen einchaset. Zusätzlich zu den Beispielen, die beim Implementieren entstanden sind, werden auch gegebene Benchmark-Anfragen besprochen, die am DBIS Lehrstuhl der Uni Rostock genutzt werden, um den CHASE auf Anfragen in verschiedenen CHASE-Tools zu untersuchen. So wurden die Benchmark-Anfragen z. B. bereits im Abschnitt 3.2 und 3.3 bei der Untersuchung von `Graal` und `PDQ` erwähnt.

Die Beispiele I1 bis I4 (Beispiele mit Instanzen) sowie Q1 bis Q3 (Beispiele mit Anfragen) sind Beispiele, die über den Zeitraum dieser Arbeit als Funktionstests entstanden sind, und nach jeder Änderung am Quellcode überprüft wurden. Diese Beispiele testen daher verschiedene Funktionalitäten, die neben dem Konzept zum Teil auch die sonstigen Erweiterungen zeigen. Die Beispiele T1 und T2 testen die Terminierungseigenschaften von ChaTEAU und das Beispiel AQuV zeigt, wie AQuV in ChaTEAU verwendet werden kann. Die Beispiele A1 bis A3 und B1 bis B3 sind die Benchmark-Anfragen, die jeweils auf einander aufbauend komplexere Kombinationen von Integritätsbedingungen im Rahmen der semantischen Anfrageoptimierung testen. Vorab muss hier aber erwähnt werden, dass die meisten Benchmark-Anfragen erst mit der BACKCHASE-Phase optimiert werden und ChaTEAU bisher nur zeigt, wie diese Anfragen nach der CHASE-Phase der BACKCHASE-Phase übergeben werden würden.

I1) Anhand des ersten Beispiels auf Instanzen werden wir einmal die Struktur der Beispiele erläutern. ChaTEAU gibt, nach erfolgreichem Einlesen und vor dem Starten der CHASE-Methode, das eingelesene Objekt als `Input` und die Integritätsbedingungen als `Constraints` aus. Nach dem Beenden der CHASE-Methode wird dann das Ergebnis als `Output` ausgegeben. Nullwerte werden in ChaTEAU mit dem Präfix `#N` definiert. Existenzquantifizierte Variablen haben den Präfix `#E` und ausgezeichnete Variablen den Präfix `#V`. Beispiel I1 ist im Anhang wie folgt angegeben:

```
-Input (Instance):
STUDENTEN(#N_matrikelnummer_1, Mueller, Maxi, Elektrotechnik),
TEILNEHMER(7, 3, #N_semester_1),
STUDENTEN(3, Mueller, Maxi, Elektrotechnik),
TEILNEHMER(2, 3, #N_semester_2)

-Constraints:
S-T TGD:
STUDENTEN(#V_matrikelnummer_1, #V_nachname_1,
        #V_vorname_1, #V_studiengang_1),
TEILNEHMER(#V_modulnummer_1, #V_matrikelnummer_1, #V_semester_1)
->
NOTEN(#V_modulnummer_1, #V_matrikelnummer_1,
        #E_semester_1, #E_note_1)

-Output (Instance):
STUDENTEN(#N_matrikelnummer_1, Mueller, Maxi, Elektrotechnik),
NOTEN(7, 3, #N_semester_3, #N_note_1),
NOTEN(2, 3, #N_semester_4, #N_note_2),
TEILNEHMER(7, 3, #N_semester_1),
STUDENTEN(3, Mueller, Maxi, Elektrotechnik),
TEILNEHMER(2, 3, #N_semester_2)
```

Die Instanz und die s-t TGD entsprechen fast den Beispielen aus dem Grundlagenkapitel. Einzig die TEILNEHMER-Relation hat noch ein zusätzliches Attribut für das Semester, um in ChaTEAU zu testen, ob wirklich alle vorhandenen Nullwerte in der Instanz beachtet werden, wenn Tupel mit neuen Nullwerten entstehen. In den neu erzeugten NOTEN-Tupeln erkennt man, dass die Nullwerte für das Semester neu über der gesamten Instanz vergeben werden. Die übergebene Instanz sowie die s-t TGD würden nach der Schreibweise dieser Arbeit wie folgt aussehen:

$$
\begin{aligned}
I =& \text{STUDENTEN}(\mu_{Ma_1}, \text{Mueller}, \text{Maxi}, \text{Elektrotechnik}),\\
& \text{STUDENTEN}(3, \text{Mueller}, \text{Maxi}, \text{Elektrotechnik})\\
& \text{TEILNEHMER}(2, 3, \mu_{Se2})\\
& \text{TEILNEHMER}(7, 3, \mu_{Se1}).\\
b =& \text{STUDENTEN}(x_{Ma}, x_{Na}, x_{Vo}, x_{St}) \wedge \text{TEILNEHMER}(x_{Mo}, x_{Ma})\\
& \longrightarrow \exists y_{Se}, y_{No} : \text{NOTEN}(x_{Mo}, x_{Ma}, y_{Se}, y_{No})
\end{aligned}
$$

Neben dem Test, ob neue Nullwerte korrekt erzeugt werden, kann mit I1 außerdem überprüft werden, ob s-t TGDs richtig verarbeitet werden. Was man im `Input` nämlich nicht sieht, ist dass die STUDENTEN- und TEILNEHMER-Relationen in der Eingabe-Datei mit dem Schema-Tag `S` versehen sind und die NOTEN-Relation das Schema-Tag `T` hat. Zudem erkennt man, auch wenn es erst einmal trivial erscheint, dass kein NOTEN-Tupel für das STUDENTEN-Tupel mit dem Nullwert in der Matrikelnummer erzeugt wird.

I2) Das Beispiel I2 veranschaulicht das Gleichsetzen verschiedener Kombinationen von Nullwerten und Konstanten. So werden in den beiden STUDENTEN-Tupeln zweimal ein Nullwert mit einer Konstanten gleichgesetzt und einmal zwei Nullwerte gleichgesetzt, sodass beide STUDENTEN-Tupel aufeinander fallen. Außerdem erkennt man, dass der Nullwert mit dem größeren Index durch den Nullwert mit dem kleineren Index ersetzt wird.

I3) I3 ähnelt I1, bis auf dass wir hier eine TGD anstatt einer s-t TGD betrachten und schon ein NOTEN-Tupel in der `Input`-Instanz vorhanden ist. Wichtig an diesem Beispiel war, dass das vorhandene NOTEN-Tupel kein zweites Mal mit anderen Nullwerten eingefügt wird. Dieses Verhalten stellte sich erst nach dem Anpassen der `isActiveTriggerFor`-Methode ein, indem mit der `isContained`-Methode nun Abbildungen unter dem Tupel beachtet werden.

I4) Der leere `Output` zusammen mit der entsprechenden Fehlermeldung in I4 zeigt, wie ChaTEAU mit dem fehlschlagenden CHASE umgeht. In der Fehlermeldung werden durch die Anpassungen in diesem Kapitel nun auch die richtigen in Konflikt stehenden Konstanten genannt.

Q1) In Q1 wird eine Anfrage übergeben, die den Anforderungen der `queryHeadCheck`- und `queryCheck`-Methode nicht genügt. Daher werden vor der Ausgabe des `Input`s auch die jeweiligen Fehlermeldungen angezeigt. Der Anfragekopf im `Input` entspricht hier schon der durch die `getQuery`-Methode angepassten Anfrage. Neben der korrekten Anpassung des Anfragekopfes zeigt das Beispiel außerdem noch das Einchasen einer TGD in eine Anfrage. Wichtig ist hier nur, dass ausschließlich für das STUDENTEN-Tupel mit der ausgezeichneten Variable für die Matrikelnummer ein NOTEN-Tupel erstellt wird und bei dem neuen Tupel wirklich neue existenzquantifizierte Variablen erzeugt werden.

Q2) Q2 genügt ebenfalls nicht den Anforderungen der `queryHeadCheck`- und `queryCheck`-Methode und beginnt mit den entsprechenden Fehlermeldungen. Hier wird getestet, wie EGDs in Anfragen eingechaset werden. Man kann erkennen, wie die existenzquantifizierte Variable für den Vornamen durch eine ausgezeichnete Variable ersetzt wird und wie die existenzquantifizierte Variable mit dem größeren Index durch die existenzquantifizierte Variable mit dem kleineren Index für den Studiengang ersetzt wird. Wichtig ist, dass die EGD, die vor der TGD steht, auch vor der TGD angewendet wird, da ansonsten zwei neue NOTEN-Tupel erzeugt werden würden.

Q3) Diesmal betrachten wir mit Q3 eine Anfrage, die den Anforderungen an die `queryHeadCheck`- und `queryCheck`-Methode genügt. Es wird vor allem getestet, wie existenzquantifizierte und ausgezeichnete Variablen innerhalb einer EGD mit Konstanten gleichgesetzt werden. Die s-t TGD, die in der `chase`-Methode als einfache TGD vorliegt, soll hier keine neuen NOTEN-Tupel einfügen, da bereits ein NOTEN-Tupel mit der entsprechenden Modulnummer und Matrikelnummer vorliegt, und sich das neue Tupel nur in den existenzquantifizierten Variablen unterscheiden würde.

T1) T1 ist als Beispiel für den nicht terminierenden CHASE aus einer Datenbankvorlesung übernommen worden. Wie zu erkennen ist, würden durch den Zyklus in den nicht schwach azyklischen TGDs immer wieder neue Tupel in die Instanz hinzugefügt werden. Hier terminiert ChaTEAU nach den Anpassungen aus dem Abschnitt 5.2.2 wie erwartet nicht mehr, was im Beispiel durch den fehlenden `Output` zu erkennen ist.

T2) In T2 wurden die TGDs aus T1 schwach azyklisch konstruiert, da anstatt der existenzquantifizierten Variable für die AbtId eine ausgezeichnete Variable gewählt wurde. So kann ChaTEAU terminieren, da erkannt wird dass die neu zu erzeugenden Tupel auf Tupel in der Instanz abgebildet werden können.

AQuV) Im Beispiel für AQuV wird gezeigt, wie es möglich ist Views als s-t TGD geschrieben in eine Anfrage einzuarbeiten. Der `Output` entspricht nur dem Aufblähen der Anfragen in der CHASE-Phase. Erst eine BACKCHASE-Phase würde die geeigneten minimalen Teilanfragen aus dem Ergebnis extrahieren.

A1) Die Benchmark-Anfrage A1 hat das Ziel zu testen, ob ein Tool Schlüssel-Fremdschlüssel-Verbunde erkennen und einsparen kann. Dieser Optimierungsschritt wird aber erst in der BACKCHASE-Phase angewandt, sodass die TGD den `Output` von ChaTEAU im Bezug auf den `Input` nicht verändert hat.

A2) A2 testet das Verbundtreue-Kriterium mit einer FD. Die EGD (FD) sagt aus, dass das zweite Attribut das dritte Attribut funktional bestimmt. Solche Abhängigkeiten können mit ChaTEAU bereits eingechaset werden, sodass auch im `Output` die Variablen für das dritte Attribut gleichgesetzt wurden.

A3) A3 testet das Verbundtreue-Kriterium mit einer JD. Die TGD (JD) fügt in die Anfrage ein Atom ein, was nur aus ausgezeichneten Variablen besteht. Die beiden anderen Atome lassen sich jeweils auf das neue Atom abbilden, sodass eine BACKCHASE-Phase die Anfrage optimieren könnte.

B1) Das Beispiel B1 beinhaltet eine Kette von Schlüssel-Fremdschlüssel-Verbunden, die in mehreren TGDs dargestellt sind. Da ChaTEAU nur die CHASE-Phase implementiert, entspricht der `Output` dem `Input`.

B2) Das Beispiel B2 beinhaltet eine Kette von Schlüssel-Fremdschlüssel-Verbunden mit wechselnden Schlüsseln, die in mehreren TGDs dargestellt sind. Da ChaTEAU nur die CHASE-Phase implementiert, entspricht der `Output` dem `Input`.

B3) Die Benchmark-Anfrage B3 war die Einzige, die das Anfragen-Optimierungs-Tool PDQ nicht optimieren konnte. ChaTEAU schafft es, die Mehrattribut-Fremdschlüssel mit interner FD korrekt in die Benchmark-Anfrage einzuchasen. B3 ist im Anhang wie folgt angegeben:

```
-Input (Query):
R2(#V_d_1, #E_e_1, konstante5),
R2(#V_d_1, #V_e_1, #E_f_1)
->
queryHead(#V_d_1, #V_e_1)

-Constraints:
TGD:
R2(#V_d_1, #V_e_1, #V_f_1)
->
R1(#V_d_1, #V_e_1, #E_c_1)

EGD:
R1(#V_a1_1, #V_a2_1, #V_a3_1),
R1(#V_a1_1, #V_a2_2, #V_a3_2)
->
#V_a2_1 = #V_a2_2

-Output (Query):
R2(#V_d_1, #V_e_1, #E_f_1),
R1(#V_d_1, #V_e_1, #E_c_2),
R2(#V_d_1, #V_e_1, konstante5),
R1(#V_d_1, #V_e_1, #E_c_1)
->
queryHead(#V_d_1, #V_e_1)
```

Wie zu erkennen ist, wurden durch die TGD zwei neue Atome für die Relation R1 hinzugefügt. Die EGD ist nur auf der Relation R1 definiert, da diese Atome jetzt aber aus Atomen über der Relation R2 erzeugt wurden, ist es hier wichtig dass die Gleichheit auch in den Atomen für R2 übernommen wird.

6. Fazit und Ausblick

Fazit Ziel dieser Arbeit war die Konzeption und die Teil-Implementierung einer Verallgemeinerung der bisher getrennten CHASE-Algorithmen in ChaTEAU. Um alle wichtigen Unterschiede zwischen dem CHASE auf Instanzen und dem CHASE auf Anfragen zu identifizieren, wurde im Grundlagen-Kapitel (siehe Kapitel 2) besonders viel Wert auf einheitliche Definitionen gelegt. Für die Lösung der Aufgabe wurde die entartete Instanz als ein CHASE-Objekt vorgestellt, welches Instanzen sowie Anfragen darstellen kann und so einem angepassten verallgemeinerten CHASE-Algorithmus übergeben werden kann. Dieser CHASE-Algorithmus arbeitet größtenteils unabhängig von der ursprünglichen Form der entarteten Instanz und liefert ein Ergebnis, welches dem einchasen von Integritätsbedingungen auf den Instanzen oder Anfragen entspricht. Erwähnenswert ist, dass mit den Anpassungen an der `chase`-Methode auch die Terminierungseigenschaften von ChaTEAU korrigiert wurden.

Genau so viel Zeit wie in die Ausarbeitung und Implementierung des Konzepts in Kapitel 4 und 5, wurde in die notwendigen Anpassungen aus dem Abschnitt 5.3 investiert, die dazu geführt haben, dass der CHASE korrekte Ergebnisse liefert. So wurden unter anderem Fehler beim Einlesen der Eingabe, beim Einfügen von Nullwerten und existenzquantifizierten Variablen in das CHASE-Objekt, beim Anwenden von EGDs und bei der Prüfung auf aktive Trigger, behoben. Gerade Letzteres hat die Korrektheit des CHASE-Ergebnisses stark verbessert. Neben den Anpassungen, die der Fehlerbehebung dienten, wurden auch einige Erweiterungen der Funktionalität von ChaTEAU implementiert. So wird eine Anfrage jetzt vor der Ausführung validiert und ihr Kopf wenn nötig richtiggestellt. Ebenso konnte die Ausgabe für Anfragen äquivalent zu der für die Instanzen hinzugefügt werden. Die wichtigste Erweiterung ist die Unterstützung von s-t TGDs, welche indirekt Views als CHASE-Parameter implementieren. Somit fehlen nur noch die Operationen, um alle Parameter aus der Tabelle 1.1 abzudecken. Zusammengefasst kann man sagen, dass das Ziel dieser Arbeit erfüllt wurde.

Ausblick Die vielen weiteren Anpassungen in ChaTEAU bilden eine wichtige Grundlage für folgende Arbeiten. Doch auch wenn viele Fehler behoben wurden, sind weitere nicht ausgeschlossen. Es ist durchaus möglich, dass ChaTEAU auf komplexere Eingaben nicht erwartungsgemäß antwortet. An dieser Stelle hat es in dieser Arbeit noch an anwendungsspezifischen Beispielen gefehlt. Solche Anwendungsbeispiele sind ebenfalls eine gute Motivation, ChaTEAU um eine Datenbankschnittstelle zu erweitern. Anschließend kann auch die Laufzeit von ChaTEAU kritisch untersucht werden. Bisher werden noch zu oft alle Tupel einer entarteten Instanz in mehreren Schleifen durchlaufen, was mit geeigneten Hashfunktionen und dem Hashen der Tupel optimiert werden könnte. Weitere Erweiterungen von ChaTEAU wurden an anderer Stelle bereits angesprochen. So wäre eine Terminierungsprüfung, wie sie in `Llunatic` durchgeführt wird, denkbar um z. B. die Eingabe auf schwache Azyklizität zu überprüfen. Hierzu läuft aktuell eine Masterarbeit am DBIS Lehrstuhl der Uni Rostock, für die die Korrektur der Terminierungseigenschaften wichtig war. Ebenfalls angesprochen wurde die Erweiterung um die BACKCHASE-Phase, welche direkt an die bisherige Ausgabe von ChaTEAU angeschlossen werden kann. Hier könnte man die Verallgemeinerung der CHASE-Objekte allerdings nicht mehr nutzen, da die BACKCHASE-Phase für Instanzen, Anfragen und den verschiedenen Integritätsbedingungen, sowie Views und Operatoren zu unterschiedlich abläuft. Zusammenhängend mit der BACKCHASE-Phase, ist Provenence ein mögliches Thema für eine Erweiterung ChaTEAUs. Hier wäre eine zusätzliche Variable für die einzelnen Tupel umsetzbar,

welche dann eine Liste von Tupel-IDs speichert, die die Herkunft des Tupels definiert. Diese Tupel-IDs könnten dann mit Provenence-Operatoren bearbeitet werden, sodass die Variable immer alle wichtigen Provenence-Informationen enthält. Eine Unterstützung eines zweiten Nullwertes für nicht existierende Werte ist ebenfalls vorstellbar. Ein letzter Ausblick für künftige Arbeiten wäre die Auflösung der geforderten Konjunktivität der Anfragen. Ohne diese könnte man ChaTEAU um Bereichsanfragen erweitern. Zum einen wären dann Bedingungen zwischen einer Variablen und einer Konstanten, aber auch Bedingungen zwischen zwei Variablen möglich (z. B. $A < 3$ oder $A < B$). Ebenso wären Erweiterungen um einen benannten Anfragekopf mit mehreren Atomen denkbar, sodass dieser wie eine s-t TGD selbst wieder Relationen definiert.

Literaturverzeichnis

[ABU79] Aho, A. V. ; Beeri, C. ; Ullman, J. 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

[AH18] Auge, Tanja ; Heuer, Andreas: Inverses in Research Data Management: Combining Provenance Management, Schema and Data Evolution (Inverse im Forschungsdatenmanagement). In: *Proceedings of the 30th GI-Workshop Grundlagen von Datenbanken, Wuppertal, Germany, May 22-25, 2018.*, 2018, 108–113

[AH19] Auge, Tanja ; Heuer, Andreas: ProSA—Using the CHASE for Provenance Management. In: Welzer, Tatjana (Hrsg.) ; Eder, Johann (Hrsg.) ; Podgorelec, Vili (Hrsg.) ; Kamišalić Latifić, Aida (Hrsg.): *Advances in Databases and Information Systems*. Cham : Springer International Publishing, 2019. – ISBN 978–3–030–28730–6, S. 357–372

[BCT14] Benedikt, Michael ; Cate, Balder ten ; Tsamoura, Efthymia: Generating Low-cost Plans from Proofs. In: *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. New York, NY, USA : ACM, 2014 (PODS '14). – ISBN 978–1–4503–2375–8, 200–211

[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, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, 2017, 37–52

[CM77] Chandra, Ashok K. ; Merlin, Philip M.: Optimal Implementation of Conjunctive Queries in Relational Data Bases. In: *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*. New York, NY, USA : ACM, 1977 (STOC '77), 77–90

[DHI12] Doan, AnHai ; Halevy, Alon Y. ; Ives, Zachary G.: *Principles of Data Integration*. Morgan Kaufmann, 2012 http://research.cs.wisc.edu/dibook/. – ISBN 978–0–12–416044–6

[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

[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

[FKMP03] Fagin, Ronald ; Kolaitis, Phokion G. ; Miller, Renée J. ; Popa, Lucian: Data Exchange: Semantics and Query Answering. In: Calvanese, Diego (Hrsg.) ; Lenzerini, Maurizio (Hrsg.) ; Motwani, Rajeev (Hrsg.): *Database Theory — ICDT 2003*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2003. – ISBN 978–3–540–36285–2, S. 207–224

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

[Gra19] *Graal Homepage*. http://graphik-team.github.io/graal/, Dezember 2019. – Accessed: 2019-12-27, saved on USB stick

[HSS18] Heuer, Andreas ; Saake, Gunter ; Sattler, Kai-Uwe: *Datenbanken - Konzepte und Sprachen, 6. Auflage*. MITP, 2018 `https://mitp.de/IT-WEB/Datenbanken/Datenbanken-Konzepte-und-Sprachen-oxid.html`. – ISBN 978–3–9584577–6–8

[Jur18] Jurklies, Martin: CHASE und BACKCHASE: Entwicklung eines Universal-Werkzeugs für eine Basistechnik der Datenbankforschung, Universität Rostock, 2018. – Masterarbeit

[Llu19] *Llunatic Homepage*. http://www.db.unibas.it/projects/llunatic/, Dezember 2019. – Accessed: 2019-12-27, saved on USB stick

[LMS95] Levy, Alon Y. ; Mendelzon, Alberto O. ; Sagiv, Yehoshua: Answering Queries Using Views (Extended Abstract). In: *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. New York, NY, USA : ACM, 1995 (PODS '95). – ISBN 0–89791–730–8, 95–104

[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

[Ren19] Renn, Fabian: Erweiterung des CHASE-Werkzeugs ChaTEAU um Anfragetransformationen, Universität Rostock, 2019. – Bachelorarbeit

A. Beispiel-Log von ChaTEAU

```
Reading from /chateau/inputfiles/Beispiel1.xml

Check Constraints for incorrectly defined S–T TGDs.
Check succeeded.

Starting pre setup
Finished pre setup
Checking integrity constraint Studenten(#V_matrikelnummer_1, #V_nachname_1,
#V_vorname_1, #V_studiengang_1),
Teilnehmer(#V_modulnummer_1, #V_matrikelnummer_1)
->
Noten(#V_modulnummer_1, #V_matrikelnummer_1, #E_semester_1, #E_note_1)
Identified active Trigger {#V_modulnummer_1 -> 2,
#V_nachname_1 -> Mueller, #V_matrikelnummer_1 -> 3,
#V_vorname_1 -> Max, #V_studiengang_1 -> Informatik}
Applying homomorphism to tgd...
Added new Mapping #E_semester_1 -> #N_semester_1 to homomorphism
Added new Mapping #E_note_1 -> #N_note_1 to homomorphism
Applying Mappings...
Added new relational atom: Noten(2, 3, #N_semester_1, #N_note_1)
Checking integrity constraint Studenten(#V_matrikelnummer_1,
#V_nachname_1, #V_vorname_1, #V_studiengang_1),
Teilnehmer(#V_modulnummer_1, #V_matrikelnummer_1)
->
Noten(#V_modulnummer_1, #V_matrikelnummer_1, #E_semester_1, #E_note_1)

Writing Output to /chateau/inputfiles\Beispiel1Result.xml
and /chateau/inputfiles\Beispiel1Result.csv
Done
```

B. Laufende Beispiele in ChaTEAU

B.1. Instanzen

B.1.1. I1

–**Input** (Instance):
STUDENTEN(#N_matrikelnummer_1, Mueller, Maxi, Elektrotechnik),
TEILNEHMER(7, 3, #N_semester_1),
STUDENTEN(3, Mueller, Maxi, Elektrotechnik),
TEILNEHMER(2, 3, #N_semester_2)

–**Constraints**:
S–T TGD:
STUDENTEN(#V_matrikelnummer_1, #V_nachname_1, #V_vorname_1, #V_studiengang_1),
TEILNEHMER(#V_modulnummer_1, #V_matrikelnummer_1, #V_semester_1)
–>
NOTEN(#V_modulnummer_1, #V_matrikelnummer_1, #E_semester_1, #E_note_1)

–**Output** (Instance):
STUDENTEN(#N_matrikelnummer_1, Mueller, Maxi, Elektrotechnik),
NOTEN(7, 3, #N_semester_3, #N_note_1),
NOTEN(2, 3, #N_semester_4, #N_note_2),
TEILNEHMER(7, 3, #N_semester_1),
STUDENTEN(3, Mueller, Maxi, Elektrotechnik),
TEILNEHMER(2, 3, #N_semester_2)

B.1.2. I2

–Input (Instance):
STUDENTEN(3, Mueller, #N_vorname_1, #N_studiengang_2),
TEILNEHMER(7, 3),
STUDENTEN(3, #N_nachname_1, Maxi, #N_studiengang_1),
TEILNEHMER(2, 3)

–Constraints:
EGD:
STUDENTEN(#V_matrikelnummer_2, #V_nachname_2, #V_vorname_2, #V_studiengang_2),
STUDENTEN(#V_matrikelnummer_2, #V_nachname_1, #V_vorname_1, #V_studiengang_1)
–>
#V_vorname_1 = #V_vorname_2,
#V_studiengang_1 = #V_studiengang_2,
#V_nachname_1 = #V_nachname_2

–Output (Instance):
TEILNEHMER(7, 3),
TEILNEHMER(2, 3),
STUDENTEN(3, Mueller, Maxi, #N_studiengang_1)

B.1.3. I3

–Input (Instance):
TEILNEHMER(2, 3),
STUDENTEN(1, Meier, Maria, Elektrotechnik),
TEILNEHMER(7, 3),
STUDENTEN(3, Mueller, Maxi, Elektrotechnik),
NOTEN(7, 3, #N_semester_1, #N_note_1)

–Constraints:
TGD:
STUDENTEN(#V_matrikelnummer_1, #V_nachname_1, #V_vorname_1, #V_studiengang_1),
TEILNEHMER(#V_modulnummer_1, #V_matrikelnummer_1)
–>
NOTEN(#V_modulnummer_1, #V_matrikelnummer_1, #E_semester_1, #E_note_1)

–Output (Instance):
NOTEN(2, 3, #N_semester_2, #N_note_2),
TEILNEHMER(2, 3),
STUDENTEN(1, Meier, Maria, Elektrotechnik),
TEILNEHMER(7, 3),
STUDENTEN(3, Mueller, Maxi, Elektrotechnik),
NOTEN(7, 3, #N_semester_1, #N_note_1)

B.1.4. I4

```
-Input (Instance):
TEILNEHMER(7, 3),
STUDENTEN(3, Mueller, Maxi, Elektrotechnik),
STUDENTEN(3, Meier, Maria, Elektrotechnik),
TEILNEHMER(2, 3)

-Constraints:
EGD:
STUDENTEN(#V_matrikelnummer_2, #V_nachname_2, #V_vorname_2, #V_studiengang_2),
STUDENTEN(#V_matrikelnummer_2, #V_nachname_1, #V_vorname_1, #V_studiengang_1)
->
#V_vorname_1 = #V_vorname_2,
#V_studiengang_1 = #V_studiengang_2,
#V_nachname_1 = #V_nachname_2

The CHASE failed because of a violated egd.
An empty instance will be returned.
The two terms in question are Maria and Maxi.
-Output (Instance):
```

B.2. Anfragen

B.2.1. Q1

```
Der QueryHead enthaelt nicht (ausschliesslich) alle ausgezeichneten Variablen
aus dem QueryBody oder umgekehrt...
Der QueryHead wurde automatisch angepasst (siehe Input-Query)!

Die Anfrage genuegt nicht den Einschraenkungen der Tableaus...
Der CHASE wird vortgesetzt. Ein korrektes Ergebnis wird nicht garantiert!

-Input (Query):
STUDENTEN(#E_matrikelnummer_3, #V_nachname_3, #V_vorname_3, #V_studiengang_3),
NOTEN(#V_modulnummer_1, #V_matrikelnummer_1, #E_semester_1, #E_note_1),
STUDENTEN(#V_matrikelnummer_2, #V_nachname_2, #V_vorname_2, #V_studiengang_2),
TEILNEHMER(#V_modulnummer_2, #V_matrikelnummer_2),
TEILNEHMER(#V_modulnummer_3, #V_matrikelnummer_3)
->
queryHead(#V_nachname_3, #V_vorname_3, #V_studiengang_3, #V_modulnummer_1,
#V_matrikelnummer_1, #V_matrikelnummer_2, #V_nachname_2, #V_vorname_2,
#V_studiengang_2, #V_modulnummer_2, #V_modulnummer_3, #V_matrikelnummer_3)

-Constraints:
TGD:
STUDENTEN(#V_matrikelnummer_1, #V_nachname_1, #V_vorname_1, #V_studiengang_1),
TEILNEHMER(#V_modulnummer_1, #V_matrikelnummer_1)
->
NOTEN(#V_modulnummer_1, #V_matrikelnummer_1, #E_semester_1, #E_note_1)

-Output (Query):
STUDENTEN(#E_matrikelnummer_3, #V_nachname_3, #V_vorname_3, #V_studiengang_3),
NOTEN(#V_modulnummer_1, #V_matrikelnummer_1, #E_semester_1, #E_note_1),
STUDENTEN(#V_matrikelnummer_2, #V_nachname_2, #V_vorname_2, #V_studiengang_2),
NOTEN(#V_modulnummer_2, #V_matrikelnummer_2, #E_semester_2, #E_note_2),
TEILNEHMER(#V_modulnummer_2, #V_matrikelnummer_2),
TEILNEHMER(#V_modulnummer_3, #V_matrikelnummer_3)
->
queryHead(#V_nachname_3, #V_vorname_3, #V_studiengang_3, #V_modulnummer_1,
#V_matrikelnummer_1, #V_matrikelnummer_2, #V_nachname_2, #V_vorname_2,
#V_studiengang_2, #V_modulnummer_2, #V_modulnummer_3, #V_matrikelnummer_3)
```

B.2.2. Q2

```
Der QueryHead enthaelt nicht (ausschliesslich) alle ausgezeichneten Variablen
aus dem QueryBody oder umgekehrt...
Der QueryHead wurde automatisch angepasst (siehe Input-Query)!

Die Anfrage genuegt nicht den Einschraenkungen der Tableaus...
Der CHASE wird vortgesetzt. Ein korrektes Ergebnis wird nicht garantiert!

-Input (Query):
STUDENTEN(#V_matrikelnummer_3, #V_nachname_2, #E_vorname_2, #E_studiengang_2),
STUDENTEN(#V_matrikelnummer_3, #V_nachname_2, #V_vorname_3, #E_studiengang_3),
TEILNEHMER(#V_modulnummer_2, #V_matrikelnummer_2),
TEILNEHMER(#V_modulnummer_3, #V_matrikelnummer_3)
->
queryHead(#V_matrikelnummer_3, #V_nachname_2, #V_vorname_3, #V_modulnummer_2,
#V_matrikelnummer_2, #V_modulnummer_3)

-Constraints:
EGD:
STUDENTEN(#V_matrikelnummer_2, #V_nachname_2, #V_vorname_2, #V_studiengang_2),
STUDENTEN(#V_matrikelnummer_2, #V_nachname_1, #V_vorname_1, #V_studiengang_1)
->
#V_vorname_1 = #V_vorname_2,
#V_studiengang_1 = #V_studiengang_2,
#V_nachname_1 = #V_nachname_2

TGD:
STUDENTEN(#V_matrikelnummer_1, #V_nachname_1, #V_vorname_1, #V_studiengang_1),
TEILNEHMER(#V_modulnummer_1, #V_matrikelnummer_1)
->
NOTEN(#V_modulnummer_1, #V_matrikelnummer_1, #E_semester_1, #E_note_1)

-Output (Query):
STUDENTEN(#V_matrikelnummer_3, #V_nachname_2, #V_vorname_3, #E_studiengang_2),
NOTEN(#V_modulnummer_3, #V_matrikelnummer_3, #E_semester_1, #E_note_1),
TEILNEHMER(#V_modulnummer_2, #V_matrikelnummer_2),
TEILNEHMER(#V_modulnummer_3, #V_matrikelnummer_3)
->
queryHead(#V_matrikelnummer_3, #V_nachname_2, #V_vorname_3, #V_modulnummer_3,
#V_modulnummer_2, #V_matrikelnummer_2)
```

B.2.3. Q3

–Input (Query):
STUDENTEN(#V_matrikelnummer_1, #E_nachname_2, #E_vorname_2, ITTI),
STUDENTEN(#V_matrikelnummer_1, #E_nachname_3, Maxi, #V_studiengang_3),
NOTEN(#V_modulnummer_1, #V_matrikelnummer_1, #E_bla_31, #E_blub_34),
TEILNEHMER(#V_modulnummer_1, #V_matrikelnummer_1)
–>
queryHead(#V_matrikelnummer_1, #V_studiengang_3, #V_modulnummer_1)

–Constraints:
S–T TGD:
STUDENTEN(#V_matrikelnummer_1, #V_nachname_1, #V_vorname_1, #V_studiengang_1),
TEILNEHMER(#V_modulnummer_1, #V_matrikelnummer_1)
–>
NOTEN(#V_modulnummer_1, #V_matrikelnummer_1, #E_semester_1, #E_note_1)

EGD:
STUDENTEN(#V_matrikelnummer_2, #V_nachname_2, #V_vorname_2, #V_studiengang_2),
STUDENTEN(#V_matrikelnummer_2, #V_nachname_1, #V_vorname_1, #V_studiengang_1)
–>
#V_vorname_1 = #V_vorname_2,
#V_studiengang_1 = #V_studiengang_2,
#V_nachname_1 = #V_nachname_2

–Output (Query):
STUDENTEN(#V_matrikelnummer_1, #E_nachname_2, Maxi, ITTI),
NOTEN(#V_modulnummer_1, #V_matrikelnummer_1, #E_bla_31, #E_blub_34),
TEILNEHMER(#V_modulnummer_1, #V_matrikelnummer_1)
–>
queryHead(#V_matrikelnummer_1, #V_modulnummer_1)

B.3. Terminierung

B.3.1. T1

–**Input** (Instance):
Ang(4711, Informatik),
Abt(Informatik, #N_ManagerId_1, Meike)

–**Constraints**:
TGD:
Abt(#V_AbtId_1, #V_ManagerId_1, #V_ManagerName_1)
–>
Ang(#V_ManagerId_1, #E_AbtId_1)

TGD:
Ang(#V_AngId_1, #V_AbtId_1)
–>
Abt(#V_AbtId_1, #E_ManagerId_1, #E_ManagerName_1)

B.3.2. T2

–**Input** (Instance):
Ang(4711, Informatik),
Abt(Informatik, #N_ManagerId_1, Meike)

–**Constraints**:
TGD:
Abt(#V_AbtId_1, #V_ManagerId_1, #V_ManagerName_1)
–>
Ang(#V_ManagerId_1, #V_AbtId_1)

TGD:
Ang(#V_AngId_1, #V_AbtId_1)
–>
Abt(#V_AbtId_1, #E_ManagerId_1, #E_ManagerName_1)

–**Output** (Instance):
Ang(4711, Informatik),
Ang(#N_ManagerId_1, Informatik),
Abt(Informatik, #N_ManagerId_1, Meike)

B.4. AQuV

–**Input** (Query):
R(#V_x_1, #E_w_1, #E_y_1),
T(#E_z_1, #E_u_1),
S(#E_y_1, #E_z_1)
–>
queryHead(#V_x_1)

–**Constraints**:
S–T TGD:
R(#V_x_1, #V_w_1, #V_y_1)
–>
VR(#V_x_1, #V_y_1)

S–T TGD:
S(#V_y_1, #V_z_1)
–>
VS(#V_y_1, #V_z_1)

S–T TGD:
S(#V_y_1, #V_z_1),
R(#V_x_1, #V_w_1, #V_y_1)
–>
VRS(#V_x_1, #V_z_1)

S–T TGD:
T(#V_z_1, #V_u_1)
–>
VT(#V_z_1, #V_u_1)

–**Output** (Query):
VS(#E_y_1, #E_z_1),
VR(#V_x_1, #E_y_1),
R(#V_x_1, #E_w_1, #E_y_1),
T(#E_z_1, #E_u_1),
VRS(#V_x_1, #E_z_1),
S(#E_y_1, #E_z_1),
VT(#E_z_1, #E_u_1)
–>
queryHead(#V_x_1)

B.5. Benchmark-Anfragen

B.5.1. A1

–**Input** (Query):
R1(#V_a1_1, #E_a2_1),
R2(#E_a2_1, #E_a3_1)
–>
queryHead(#V_a1_1)

–**Constraints**:
TGD:
R1(#V_a1_1, #V_a2_1)
–>
R2(#V_a2_1, #E_a3_1)

–**Output** (Query):
R1(#V_a1_1, #E_a2_1),
R2(#E_a2_1, #E_a3_1)
–>
queryHead(#V_a1_1)

B.5.2. A2

–**Input** (Query):
R(#E_a1_1, #V_a2_1, #V_a3_1),
R(#V_a1_1, #V_a2_1, #E_a3_1)
–>
queryHead(#V_a2_1, #V_a3_1, #V_a1_1)

–**Constraints**:
EGD:
R(#V_a1_2, #V_a2_1, #V_a3_2),
R(#V_a1_1, #V_a2_1, #V_a3_1)
–>
#V_a3_1 = #V_a3_2

–**Output** (Query):
R(#E_a1_1, #V_a2_1, #V_a3_1),
R(#V_a1_1, #V_a2_1, #V_a3_1)
–>
queryHead(#V_a2_1, #V_a3_1, #V_a1_1)

B.5.3. A3

–**Input** (Query):
R(#E_a1_1, #V_a2_1, #V_a3_1),
R(#V_a1_1, #V_a2_1, #E_a3_1)
–>
queryHead(#V_a2_1, #V_a3_1, #V_a1_1)

–**Constraints**:
TGD:
R(#V_a1_2, #V_a2_1, #V_a3_2),
R(#V_a1_1, #V_a2_1, #V_a3_1)
–>
R(#V_a1_1, #V_a2_1, #V_a3_2)

–**Output** (Query):
R(#E_a1_1, #V_a2_1, #V_a3_1),
R(#V_a1_1, #V_a2_1, #V_a3_1),
R(#V_a1_1, #V_a2_1, #E_a3_1)
–>
queryHead(#V_a2_1, #V_a3_1, #V_a1_1)

B.5.4. B1

–Input (Query):
R1(#V_a_1, #V_b_1),
R2(#V_a_1, #E_c_1),
R4(#V_a_1, #E_e_1),
R5(#V_a_1, konstante10),
R3(#V_a_1, #E_d_1)
–>
queryHead(#V_a_1, #V_b_1)

–Constraints:
TGD:
R5(#V_a_1, #V_f_1)
–>
R4(#V_a_1, #E_e_1)

TGD:
R4(#V_a_1, #V_e_1)
–>
R3(#V_a_1, #E_d_1)

TGD:
R3(#V_a_1, #V_d_1)
–>
R2(#V_a_1, #E_c_1)

TGD:
R2(#V_a_1, #V_c_1)
–>
R1(#V_a_1, #E_b_1)

–Output (Query):
R1(#V_a_1, #V_b_1),
R2(#V_a_1, #E_c_1),
R4(#V_a_1, #E_e_1),
R5(#V_a_1, konstante10),
R3(#V_a_1, #E_d_1)
–>
queryHead(#V_a_1, #V_b_1)

B.5.5. B2

–Input (Query):
R4(#E_d_1, #E_e_1),
R1(#V_a_1, #V_b_1),
R3(#V_a_1, #E_d_1),
R2(#V_a_1, #E_c_1),
R5(#E_d_1, konstante10)
–>
queryHead(#V_a_1, #V_b_1)

–Constraints:
TGD:
R5(#V_d_1, #V_f_1)
–>
R4(#V_d_1, #E_e_1)

TGD:
R4(#V_d_1, #V_e_1)
–>
R3(#E_a_1, #V_d_1)

TGD:
R3(#V_a_1, #V_d_1)
–>
R2(#V_a_1, #E_c_1)

TGD:
R2(#V_a_1, #V_c_1)
–>
R1(#V_a_1, #E_b_1)

–Output (Query):
R4(#E_d_1, #E_e_1),
R1(#V_a_1, #V_b_1),
R3(#V_a_1, #E_d_1),
R2(#V_a_1, #E_c_1),
R5(#E_d_1, konstante10)
–>
queryHead(#V_a_1, #V_b_1)

B.5.6. B3

–Input (Query):
R2(#V_d_1, #E_e_1, konstante5),
R2(#V_d_1, #V_e_1, #E_f_1)
–>
queryHead(#V_d_1, #V_e_1)

–Constraints:
TGD:
R2(#V_d_1, #V_e_1, #V_f_1)
–>
R1(#V_d_1, #V_e_1, #E_c_1)

EGD:
R1(#V_a1_1, #V_a2_1, #V_a3_1),
R1(#V_a1_1, #V_a2_2, #V_a3_2)
–>
#V_a2_1 = #V_a2_2

–Output (Query):
R2(#V_d_1, #V_e_1, #E_f_1),
R1(#V_d_1, #V_e_1, #E_c_2),
R2(#V_d_1, #V_e_1, konstante5),
R1(#V_d_1, #V_e_1, #E_c_1)
–>
queryHead(#V_d_1, #V_e_1)

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommen 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 2.3.2020 _____