Universität Rostock

Traditio et Innovatio

Masterarbeit

Erweiterung des CHASE-Werkzeugs ChaTEAU um ein Terminierungskriterium

eingereicht von: Andreas Oliver Görres

eingereicht am: 03.03.2020

Gutachter: Andreas Heuer
Meike Klettke

Zusammenfassung

Der Chase stellt einen grundlegenden Algorithmus der Datenbanktheorie dar. Durch ihn können insbesondere Integritätsbedingungen in Datenbankinstanzen eingearbeitet werden. Während der Chase in vielen praktischen Anwendungsfällen konfluent ist und terminiert, ist dies für allgemeine Integritätsbedingungen nicht der Fall. Obwohl das Problem der Chase-Terminierung im Allgemeinen unentscheidbar ist, werden in der Literatur Klassen von Integritätsbedingungen beschrieben, für die der Chase auf jeder Datenbankinstanz terminiert. In der vorliegenden Arbeit werden diese Terminierungsklassen auf Basis einer konkreten Beispieldatenbank dargestellt und verglichen. Als Ergebnis dieser Evaluation werden mehrere Terminierungstests ausgewählt, in Form eines Software-Tools implementiert und in das Chase-Werkzeug ChaTEAU integriert. Nutzer wählen eine beliebige Kombinationen von Terminierungstests, die unabhängig vom eigentlichen Chase-Vorgang die Terminierungseigenschaften des Chase überprüfen. Wenn keines der Testergebnisse die Terminierung sicherstellt, besteht zumindest eine hohe Wahrscheinlichkeit, dass der Chase nicht terminieren wird. Dem Nutzer steht jedoch frei, diese Einschätzung zu ignorieren und den Chase dennoch zu starten. Während der Terminierungstester die Chase-Terminierung in allen gegebenen Testfällen korrekt bestimmt, weist eine Laufzeitanalyse auf die schlechte Skalierbarkeit einzelner Testmethoden hin. Da also kein Kriterium für jeden Anwendungsfall geeignet ist, enthält die vorliegende Arbeit Empfehlungen, die den Nutzer bei der Wahl des passenden Terminierungstests unterstützen sollen. Insbesondere sollte zunächst ein Test mit polynomialer Laufzeit angewandt werden, zum Beispiel der Test auf Safety, und erst anschließend – abhängig von den Ergebnissen dieser Untersuchung – ein Test exponentieller Laufzeit, zum Beispiel der Test auf Azyklizität, durchgeführt werden.

Abstract

The Chase is a basic algorithm of database theory. In particular, it can be used to incorporate integrity constraints into database instances. While the Chase is confluent and terminates in most cases, this is not guaranteed for general integrity constraints. Chase termination has been shown to be an undecidable problem. Nonetheless, several criteria for integrity constraints are known which guarantee Chase terminating on any database instance. This thesis compares termination criteria found in literature using examples from a concrete database. Based on our analysis, we choose the most suitable termination tests and integrate them into the Chase tool ChaTEAU. Users may select an arbitrary combination of termination tests. In case none of the test results guarantees safe termination of the Chase, there is a high chance the algorithm will not terminate. However, the user is free to ignore this prediction and to start the Chase anyway. While the termination tester was able to predict Chase termination correctly in all given test cases, the runtime analysis suggests that not all of the test criteria are scalable. Since none of the test criteria is suitable for every use case, we provide recommendations to support the user's decision for an appropriate termination test. In particular, users should perform a test with polynomial time complexity (e.g. Safety) first, and only afterwards – based on the results of the previous test – they should execute a test with exponential time complexity (e.g. Acyclicity).

Inhaltsverzeichnis

1. Einleitung

Der Chase stellt eine grundlegende Technik der Datenbanktheorie dar, die in einer Vielzahl von Einsatzgebieten Anwendung findet. Hierzu zählen unter anderem:

- Integration heterogener Datenbanken
- Anfragetransformation mit Beschränkung auf bestimmte Sichten (AQuV)
- Anfrageoptimierung unter Integritätsbedingungen
- Provenancemanagement von Daten

Am Lehrstuhl für Datenbanken und Informationssysteme der Universität Rostock wurde das Chase-Werkzeug ChaTEAU entwickelt, das bislang in den Anwendungsfällen Datenbankintegration und Anfrageoptimierung eingesetzt werden kann, in Zukunft jedoch auf weitere Anwendungsgebiete erweitert werden soll.

Terminierung und Konfluenz des Chase sind Grundlage einer interessanten Problemstellung: Während die Einarbeitung einfacher Abhängigkeiten – zum Beispiel funktionaler Abhängigkeiten – in polynomialer Zeit zu einem eindeutigen Ergebnis führt, ist für allgemeinere Abhängigkeiten, wie TGDs (Tuple Generating Dependencies) und EGDs (Equality Generating Dependencies), die Terminierung des Algorithmus nicht entscheidbar. Dies betrifft sowohl die Terminierung auf einer bestimmten Instanz, als auch die Terminierung auf allen möglichen Instanzen [GO18]. Darüber hinaus ist nicht auszuschließen, dass der Chase nur manchmal nicht terminiert.

Eines der einfachsten Terminierungskriterien stellt die Schwache Azyklizität dar. Für dieses Kriterium (wie auch für die meisten anderen Terminierungsbedingungen) werden eingebettete TGDs überprüft, die sich selbst zyklisch aufrufen. Wenn die im letzten Zyklus erzeugten Nullwerte nicht in die neu generierten Tupel (die neue Nullwerte enthalten können) übertragen werden, ist ein Terminieren des Chase garantiert. Obwohl der Algorithmus eine wichtige Klasse von terminierenden Bedingungen erkennt, wird insbesondere das gegenseitige Aufrufen von TGDs nur unzureichend untersucht. Das Kriterium der Stratifizierung behandelt diesen Sachverhalt näher und trifft eine Vorauswahl der TGD-Kombinationen, deren Schwache Azyklizität gewährleistet sein muss. Interessanterweise zeigt das Kriterium der Stratifizierung nicht, dass alle möglichen Abfolgen von Chase-Schritten terminieren, sondern lediglich, dass mindestens eine Folge von Chase-Schritten zu einem Ergebnis kommt. Korrigierte Stratifizierung (c-Stratifizierung) verwendet den naiven Oblivious Chase anstelle des Standard-Chase und garantiert die Terminierung aller möglichen Folgen von Chase-Schritten. Obwohl Stratifizierung selbst nur ein Vorschritt des Testes auf Schwache Azyklizität darstellt, stellen Stratifizierung und Schwache Azyklizität die Grundlagen für zwei eigene Klassen von Terminierungskriterien dar [GO18]. Zu den Erweiterungen der Schwachen Azyklizität zählt der Test auf „sichere“ Bedingungen und die Superschwache Azyklizität, während zu den auf Stratifizierung basierenden Kriterien die Induktive Restriktion und deren Erweiterung T[k] zählen.

Das Kriterium SwA-Str stellt eine Erweiterung von sowohl Superschwacher Azyklizität als auch von c-Stratifizierung dar, ist jedoch nicht mit den anderen auf Stratifizierung beruhenden Bedingungen vergleichbar. Lokale Stratifizierung wiederum stellt eine Erweiterung aller erwähnten Bedingungen dar.

„Constraint-Rewriting“ bietet eine Möglichkeit, die Anzahl der Bedingungsmengen zu erhöhen, die von einem der genannten Kriterien als terminierend erkannt werden. Hierfür wird nicht die ursprüngliche Menge an TGDs überprüft, sondern TGDs, die aus diesen durch „Adornment“ gewonnen werden. Adornment (bzw. Adornment++, eine Erweiterung des Adornment-Algorithmus) kann nicht nur zusammen mit anderen Terminierungskriterien verwendet werden, sondern ist auch für sich bereits Grundlage des Azyklizitätskriteriums, welche alle anderen zuvor genannten Terminierungsbedingungen erweitert [GST11].

In bisherigen Abschluss- und Projektarbeiten der Universität Rostock wurden die Chase-Werkzeuge Llunatic, Graal, PDQ, ChaseFun und ProvCB untersucht. Aus diesen Analysen geht hervor, dass zumindest Llunatic einen Test auf Schwache Azyklizität durchführt. Ob die anderen genannten Programme einen Test auf Terminierung durchführen, ist zum gegenwärtigen Zeitpunkt noch unklar.

Im Gegensatz zu den genannten Werkzeugen überprüft ChaseTEQ explizit die meisten der oben genannten Terminierungskriterien [Spe11]. ChaseTEQ entstand in der Arbeitsgruppe der Entwickler des Kriteriums der Lokalen Stratifizierung und des Adornment-Algorithmus. Weitere Funktionen von ChaseTEQ betreffen die Reparatur unvollständiger Datenbanken und das Formulieren von Anfragen an diese Datenbanken.

1.1. Aufgabenstellung

Ziel der zu erstellenden Arbeit ist es, die Überprüfung eines Terminierungskriteriums in das bereits vorliegende Chase-Programm ChaTEAU einzubauen. In einem ersten Schritt setzt dies eine Überprüfung der bekannten Terminierungskriterien und ihrer Implementierung in vergleichbaren Chase-Werkzeugen voraus. Hierbei wird ein Fokus auf die Arbeiten von Greco et al. gelegt, der einige der umfassendsten Terminierungskriterien (Lokale Stratifizierung, SwA-Str, Azyklizität ...) definiert hat [GST11].

Die Berücksichtigung von EGDs soll ebenfalls untersucht werden, allerdings ist die praktische Umsetzung hierfür nicht Kern der Arbeit. Algorithmen wie die Umwandlung von EGDs in TGDs können relativ einfach nachträglich in ChaTEAU eingebunden werden, bieten jedoch keine letztendlich befriedigende Lösung des Problems.

Neben EGDs und TGDs kann ChaTEAU auch st-TGDs verarbeiten. Ob dies eine Anpassung des Algorithmus verlangt oder ob das Softwaremodul auf TGDs beschränkt werden muss, verlangt einer weiteren Untersuchung – die bisher zitierten Arbeiten berücksichtigen Terminierungskriterien von st-TGDs allerdings nicht.

1.2. Beispieldatensatz und Notation

Die Untersuchung der Kriterien soll anhand eines laufenden Beispiels erfolgen, das auf der schon in früheren Projektarbeiten verwendeten Studentendatenbank beruht. Diese besteht mindestens aus den Tabellen:

STUDENTEN St($Matrikelnummer, Nachname, Vorname, Studiengang, Institut$)
FÄCHER Fä($Studiengang, Institut, FSR\_Sprecher$)
NOTEN No($Matrikelnummer, Modulnummer, Note$)
NOTEN_INFORMATIK Ni($Nachname, Modulnummer, Note, Institut$).

Darüber hinaus ist es für einige Beispiele wünschenswert, Relationen zu besitzen, in deren Tupeln derselbe Wert mehrfach auftreten kann:

MODULE Mo($Modulnummer, Voraussetzungsmodul, Äquivalenzmodul$)
KOMPLEXPRÜFUNGEN Ko($Matrikelnummer, Modulnummer1, Modulnummer2, Modulnummer3, Modulnummer4$).

Relationennamen und Variablen, die diese Studentendatenbank betreffen, werden in der vorliegenden Arbeit jeweils durch zwei Buchstaben dargestellt. Für Relationennamen sind dies die soeben genannten Abkürzungen (z.B. No für Noten), während Variablennamen durch Abkürzung des Attributnamens entstehen, in welchem sie zuerst auftauchen. Für Relationennamen wird ein Großbuchstabe und ein Kleinbuchstabe verwendet, während Variablen entweder durch zwei Großbuchstaben (existenzquantifizierte Variablen, z.B. NO) oder zwei Kleinbuchstaben (allquantifizierte Variablen, z.B. mo) dargestellt werden. Darüber hinaus sind Variablennamen im Gegensatz zu Relationennamen kursiv gesetzt. Ein *Atom* der Notenrelation mit einer existenzquantifizierten Variablen im Attribut Note und zwei allquantifizierten Variablen in den Attributen Matrikelnummer und Modulnummer könnte also wie folgt notiert werden: No(ma, mo, NO).

Des Weiteren können sowohl Relationennamen als auch Variablen mit einer Zahl ergänzt werden. Auf diese Weise können zwei ansonsten gleichnamige Variablen unterschieden werden. Für bestimmte Variablen stammt die Zahl auch aus dem Attributnamen (z.B. die Variable $mo1$ des Attributs $Modulnummer1$). Ein Relationenname mit tiefgestellter Zahl bezeichnet hingegen eine *Position* in dieser Relation (z.B. Ko_1). Zahlen in Positionsnamen sind tiefgestellt, während Zahlen in Variablennamen (wie auch in Attributnamen) nicht tiefgestellt sind. Markierte Nullwerte hingegen (z.B. η_1) sind durch einen tiefgestellten Index gekennzeichnet.

Obwohl wir in Beispielen wenn möglich auf diese Studentendatenbank Bezug nehmen, wird an einigen Stellen – insbesondere in Definitionen – eine abstrakte Datenbank verwendet, deren Relationen mit einzelnen Großbuchstaben, wie R, S und T, bezeichnet werden (wobei die Anzahl der Attribute dieser Relationen nicht allgemein festgelegt ist) und deren Variablennamen (von den Relationennamen unabhängig) aus Klein- oder Großbuchstaben, wie x, y oder Z, bestehen.

Eine *Konjunktion von Atomen* kann im Folgenden auch als einzelnes Atom mit dem Relationennamen ϕ, ψ, λ oder ρ dargestellt sein. Eine derartige Konjunktion von Atomen hängt von einem Vektor aus Variablen ab, welche ähnlich einer einzelnen Variablen durch einen der Buchstaben x, y oder z (zum Teil mit ergänzender Zahl) gekennzeichnet sind. Vektoren von Variablen sind fettgesetzt. Durch Verwendung mehrerer derartiger Vektoren ist eine nähere Charakterisierung der Variablen möglich. $\exists \mathbf{y} : \psi(\mathbf{x}, \mathbf{y})$ steht also zum Beispiel für eine Konjunktion von Atomen ψ, deren Variablen allquantifiziert ($\mathbf{x}$) oder existenzquantifiziert sind ($\mathbf{y}$), allerdings muss nicht jedes Atom alle Variablen aus $\mathbf{x}$ oder $\mathbf{y}$ enthalten. Konjunktionen von Gleichheitsatomen (z.B. $\mathbf{x}_i = \mathbf{x}_j$ für die Konjunktion $x1 = x3$, $x2 = x4$) sind nur durch die fettgesetzten Variablennamen von einzelnen Gleichheitsatomen unterscheidbar. Die Teilvektoren $\mathbf{x}_i$ und $\mathbf{x}_j$ aus dem Kopf einer EGD bestehen jeweils aus Komponenten des Vektors $\mathbf{x}$ aus dem EGD-Körper (z.B. $\mathbf{x} = (x1, x2, x3, x4)$, $\mathbf{x}_i = (x1, x2)$, $\mathbf{x}_j = (x3, x4)$). Zum Teil werden auch Variablen einzelner Atome zu Vektoren zusammengefasst. Mengen von Variablen werden im Gegensatz zu Vektoren nicht fettgesetzt, sondern großgeschrieben. Der Zusammenhang zwischen Großschreibung von Variablensymbolen und der Existenzquantifizierung der Variablen besteht also nur für einzelne Variablen und nicht für Vektoren oder Mengen von Variablen.

1.3. Aufbau der Arbeit

Im Folgenden (Kapitel 2) werden zunächst allgemeine Begriffe des relationalen Datenbankmodells definiert. Auf Basis dieses grundlegenden Vokabulars wird anschließend ein spezifischer Algorithmus der Datenbankforschung – der *Chase* – näher betrachtet. Von besonderer Bedeutung ist hierbei die Terminierung des Chase.

Kapitel 3 untersucht den aktuellen Stand der Forschung. Das zuvor angesprochene Problem der Chase-Terminierung wird erneut aufgegriffen, indem anhand einer Literaturrecherche bekannte Terminierungskriterien verglichen werden. Tatsächlich ist die vorliegende Arbeit nicht der erste Ansatz, einen Terminierungstester in ein Chase-Werkzeug einzubinden. Im folgenden Abschnitt 3.2 stellen wir bestehende Chase-Tools kurz vor und untersuchen anhand eines einfachen Anwendungsfalles, wie sie mit einem nicht terminierenden Chase umgehen.

Auf Basis der in den Vorkapiteln gemachten theoretischen Überlegungen und Anregungen durch bereits implementierte Chase-Werkzeuge fällen wir anschließend die Entscheidung, welche Kriterien für ChaTEAUs Terminierungstester geeignet sind. Diese Entscheidung bildet die Einleitung von Kapitel 4, das sich mit Konzeption und Implementierung der Testsoftware beschäftigt. Hierbei werden mehrere grundlegende Algorithmen des Programmes beschrieben und alternative Workflows für Testabläufe diskutiert. Es folgt eine Evaluation des implementierten Terminierungstesters anhand mehrerer Anwendungsfälle. Hierbei wird insbesondere der Skalierbarkeit einzelner Testverfahren besondere Aufmerksamkeit gewidmet. In Kapitel 6 ziehen wir ein Fazit und fassen abschließend zusammen, welche Ziele im Rahmen der vorliegenden Arbeit umgesetzt werden konnten und welche Ansätze für eine Weiterentwicklung des Terminierungstesters bestehen.

2. Grundlagen

Im folgenden Kapitel stellen wir zunächst einige grundlegende Konzepte des relationalen Datenbankmodells vor (Abschnitt 2.1) und betrachten anschließend den Chase, einen wichtigen Algorithmus der Datenbankforschung (Abschnitt 2.2). Wenn in der Literatur unterschiedliche Schreibweisen und Definitionen vorhanden sind, folgen wir konsequent [GMS12] (oder einer anderen Publikation derselben Arbeitsgruppe) und passen gegebenenfalls die Notation an die in der Einleitung (Abschnitt 1.2) gegebene Schreibweise an.

2.1. Relationales Datenbankmodell

Im weiteren Verlauf der Arbeit werden wir auf ein allgemeines Datenbankvokabular zurückgreifen, dessen Begriffe wir hier definieren wollen. Von zentraler Bedeutung für den Chase sind hierbei insbesondere Integritätsbedingungen, prädikatenlogische Formeln, die Beziehungen zwischen Relationen definieren.

2.1.1. Schemata und Instanzen

Gegeben seien folgende paarweise disjunkte Mengen:

- eine abzählbar unendliche Menge von *Konstanten* (KONST), auch als Datenbankdomäne bezeichnet,
- eine abzählbar unendliche Menge von *Attributen* (A), wobei jedes Attribut $A_i \in$ A mit einer Menge Konstanten (der Attributdomäne $\mathrm{dom}(A_i)$) assoziiert ist,
- eine abzählbar unendliche Menge von *Relationennamen* (R).

Jeder Relationenname $R_i \in$ R ist mit einem Tupel von Attributen $(A_1, ..., A_n)$ assoziiert. Die Anzahl der Attribute $A_1, ..., A_n$ von R_i wird als *Stelligkeit* von R_i (Stelligkeit(R_i)) bezeichnet.

$R_i(A_1, ..., A_n)$ (bzw. $R_i(U_i)$) stellt ein *Relationenschema* dar. Eine *Relation* r über einem solchen Relationenschema ist eine endliche Untermenge $\mathrm{dom}(A_1) \times ... \times \mathrm{dom}(A_n)$. Wir bezeichnen r auch als Relation über R.

Eine *Position* P_m steht für ein Tupel (P, A_m), wobei P einen Relationennamen aus R darstellt und A_m ein Attribut des Relationenschemas von P ist ($1 \leq m \leq$ Stelligkeit(P)). Diese Definition der Position setzt voraus, dass die Attribute jedes Relationenschemas durchnummeriert sind. Wenn in Beispielen der vorliegenden Arbeit Positionen verwendet werden, ist stets die in Anhang A.4 eingeführte Nummerierung gemeint. Wie in der Einleitung bereits angegeben, werden in der vorliegenden Arbeit Zahlen in Positionsnamen – im Gegensatz zu Zahlen in Variablennamen – tiefgestellt.

Jedes Element $t \in R$ ist ein *Tupel*, wobei $t[A_i]$ für die Komponente A_i von t steht. In gleicher Weise bezeichnet $t[X](X \subseteq \{A_1, ..., A_n\})$ die Einengung (d.h. die Projektion) von t auf X.

Ein *Datenbankschema* stellt eine nichtleere, endliche Menge $\mathbf{R} = \{R_1(U_1), ..., R_m(U_m)\}$ von relationalen Schemata dar. Eine *Datenbankinstanz* D über einem derartigen Datenbankschema ist eine endliche Menge

von Relationen $\{r_1, ..., r_m\}$, wobei jedes $r_i \in r_1, ..., r_m$ für eine Relation über $R_i(U_i)$ steht. Tupel einer Relation über dem Schema $R_i(A_1, ..., A_n)$ werden mit $R_i(t)$ bezeichnet.

2.1.2. Integritätsbedingungen

Gegeben seien zusätzlich zu den im vorherigen Unterkapitel geforderten Mengen:

- eine unendliche Menge markierter Nullwerte NULL (die im Folgenden mit $\eta_i, i \in \mathbb{N}^+$ bezeichnet werden),
- eine unendliche Menge von Variablen VAR.

Definition 2.1. (Atom, nach [GST11])
Ein **relationales Atom** hat die Form $R(t_1, ..., t_n)$, wobei R ein Relationenname (bzw. ein „relationales Prädikat“) ist, $t_1, ..., t_n$ Terme sind, die zur Menge VAR $\cup$ NULL $\cup$ KONST gehören, und $n =$ Stelligkeit(R). Ein **Gleichheitsatom** hat hingegen die Form $t_1 = t_2$, wobei die Terme t_1 und t_2 Elemente der Menge VAR $\cup$ NULL $\cup$ KONST sind.

Eine *eingebettete Abhängigkeit* ist ein prädikatenlogischer Ausdruck erster Ordnung der Form

$$\forall \mathbf{x} \forall \mathbf{y} \phi(\mathbf{x}, \mathbf{y}) \rightarrow \exists \mathbf{z} : \psi(\mathbf{x}, \mathbf{z}),$$

wobei $\mathbf{x}, \mathbf{y}, \mathbf{z}$ Tupel von Variablen sind und $\phi(\mathbf{x}, \mathbf{y})$ und $\psi(\mathbf{x}, \mathbf{z})$ Konjunktionen von Relations- und Gleichheitsatomen darstellen.

$\phi(\mathbf{x}, \mathbf{y})$ und $\psi(\mathbf{x}, \mathbf{z})$ werden als *Körper* (oder linke Seite) und *Kopf* (oder rechte Seite) der Abhängigkeit bezeichnet.

Von nun an wird implizit angenommen, dass alle nicht-quantifizierten Variablen allquantifiziert sind, weshalb wir auf die Darstellung des Allquantors verzichten. Des Weiteren schreiben wir (wie bereits in der Einleitung erwähnt) die Namen allquantifizierter Variablen in Kleinbuchstaben und die Namen existenzquantifizierter Variablen in Großbuchstaben.

Im Allgemeinen gehen wir davon aus, dass Gleichheitsatome nur im Kopf von Abhängigkeiten vorkommen und existenzquantifizierte Variablen nicht in Gleichheitsatomen auftauchen. In Definition 3.15 wird allerdings eine Ausnahme der ersten Annahme gemacht. In der vorliegenden Arbeit werden vor allem folgende Gruppen von Abhängigkeiten behandelt:

- *volle Abhängigkeiten* enthalten keine existenzquantifizierten Variablen $\mathbf{z}$,
- *TGDs* (Tuple Generating Dependencies) weisen keine Gleichheitsatome auf,
- *EGDs* (Equality Generating Dependencies) sind voll und weisen einen Kopf auf, der nur aus einem einzelnen Gleichheitsatom besteht.

Funktionelle Abhängigkeiten stellen einen einfachen Spezialfall der EGDs dar: Sei $R(U)$ ein Relationenschema. Eine FD über $R(U)$ ist ein Ausdruck der Form $X \rightarrow Y$, wobei $X, Y \subseteq U$ gilt. Eine Relation r über $R(U)$ erfüllt $X \rightarrow Y$ genau dann, wenn für alle Paare von Tupeln $t_1, t_2 \in r$ gilt: Wenn $t_1[X] = t_2[X]$, so gilt $t_1[Y] = t_2[Y]$.

Als prädikatenlogischer Ausdruck kann $X \rightarrow Y$ wie folgt notiert werden:

$$\mathrm{R}(\mathbf{x}, \mathbf{y1}, \mathbf{z1}), \mathrm{R}(\mathbf{x}, \mathbf{y2}, \mathbf{z2}) \rightarrow \mathbf{y1} = \mathbf{y2}.$$

Hierbei sind $\mathbf{x}, \mathbf{y1}, \mathbf{z1}, \mathbf{y2}, \mathbf{z2}$ Tupel von Variablen, wobei $\mathbf{x}$ der Attributmenge X, die Tupel $\mathbf{y1}$ und $\mathbf{y2}$ jeweils der Attributmenge Y und die Tupel $\mathbf{z1}$ und $\mathbf{z2}$ jeweils dem Komplement der genannten Attributmengen $Z = U - XY$ entspricht.

Join Dependencies (JDs) stellen einen Spezialfall voller TGDs dar: Sei $R(U)$ ein Relationenschema. Eine Join Dependency über $R(U)$ ist ein Ausdruck der Form $\bowtie \{X_1, ..., X_n\}$, wobei $X_i \subseteq U$ für $1 \leq i \leq n$ und $\cup_{i=1}^{n} X_i = U$ gilt. Eine Relation r über $R(U)$ erfüllt $\bowtie \{X_1, ..., X_n\}$ genau dann, wenn

$$r = \bowtie_{i=1}^{n} r[X_i].$$

Als prädikatenlogischer Ausdruck kann $\bowtie \{XY, YZ, XZ\}$ wie folgt notiert werden:

$$\mathrm{R}(\mathbf{x1}, \mathbf{y1}, \mathbf{z2}), \mathrm{R}(\mathbf{x2}, \mathbf{y1}, \mathbf{z1}), \mathrm{R}(\mathbf{x1}, \mathbf{y2}, \mathbf{z1}) \rightarrow \mathrm{R}(\mathbf{x1}, \mathbf{y1}, \mathbf{z1}).$$

Hierbei sind $\mathbf{x1}, \mathbf{y1}, \mathbf{z1}, \mathbf{x2}, \mathbf{y2}, \mathbf{z2}$ Tupel von Variablen, wobei $\mathbf{x1}, \mathbf{x2}$ der Attributmenge X, $\mathbf{y1}, \mathbf{y2}$ der Attributmenge Y und $\mathbf{z1}, \mathbf{z2}$ der Attributmengen Z entspricht.

Definition 2.2. (Lösung, [GMS12])
Sei D eine Datenbankinstanz und sei Σ eine Menge von Abhängigkeiten über dem Datenbankschema von D. Eine Lösung (D, Σ) ist eine Instanz J, für die gilt $D \subseteq J$ und $J \models \Sigma$. Die Menge aller Lösungen für (D, Σ) wird mit $\mathrm{Sol}(D, \Sigma)$ bezeichnet.

Bisher haben wir die grundlegenden Begriffe des relationalen Datenbankmodells definiert. Dies war nötig, um das Verhalten des Chase-Algorithmus verstehen zu können, mit dem wir uns nun beschäftigen wollen.

2.2. Chase

Im Folgenden soll der Chase [MMS79], ein Algorithmus mit breitem Anwendungsspektrum in der Datenbankforschung, näher betrachtet werden. Wir werden uns hier auf Varianten des Chase beschränken, die für die behandelte Problemstellung auch tatsächlich relevant sind – dies ist zum einen der *Standard-Chase*, der in ChaTEAU implementiert ist, und zum anderen Varianten des *Oblivious Chase*, die für Chase-Terminierungskriterien von Bedeutung sind. Tatsächlich beziehen sich nämlich die meisten der in späteren Kapiteln behandelten Kriterien (außer z.B. die *Stratifikation*) nicht auf den Standard-Chase, sondern auf den *Skolem-Oblivious Chase*.

2.2.1. Algorithmus

Der Chase ist ein Fixpunktalgorithmus, der *Parameter* in ein *Objekt* integriert. Wichtig hierbei ist lediglich, dass sich sowohl Parameter als auch Objekt durch prädikatenlogische Ausdrücke darstellen lassen. Beispielsweise können durch den Chase Integritätsbedingungen in Datenbankinstanzen eingebaut werden (weitere Beispiele folgen in Unterkapitel 2.2.2).

Durch den Körper der Integritätsbedingung wird ein „Muster“ definiert. Wenn sich dieses „Muster“ in der Datenbankinstanz wiederfindet (d.h. wenn ein *Homomorphismus* vom Körper der Abhängigkeit zu

einem Tupel der Datenbankinstanz existiert), so liegt ein *Trigger* vor. Ist der Kopf der Abhängigkeit nicht erfüllt, so ist der Trigger *aktiv*. Für TGDs bedeutet dies, dass das geforderte Tupel noch nicht in der Datenbankinstanz existiert. Für EGDs ist der Trigger aktiv, wenn die Terme, deren Gleichheit gefordert wird, ungleich sind.

Als Konsequenz auf den aktiven Trigger einer TGD generiert der Chase ein neues Tupel, das mitunter neue markierte Nullwerte enthält (in den Positionen der existenzquantifizierten Variablen). Der aktive Trigger einer EGD hingegen führt zum Austausch eines Nullwerts gegen eine Konstante oder einen anderen Nullwert (mit niedrigerem Index). Wenn ein solcher Austausch nicht möglich ist (weil durch die EGD zwei Konstanten gleichgesetzt werden), scheitert der Chase.

Definition 2.3. (Homomorphismus, [GMS12])
Seien K und J zwei Instanzen über dem selben Datenbank-Schema mit Werten aus KONST $\cup$ NULL. Ein **Homomorphismus** $h : K \to J$ ist eine Abbildung von KONST$(K) \cup$ NULL(K) nach KONST$(J) \cup$ NULL(J), sodass gilt:

1. $h(c) = c$ für jedes c in KONST(K).
2. Für alle Fakten $R_i(t)$ von K gilt, dass $R_i(h(t))$ ein Faktum von J ist (wobei für $t = (a_1, ..., a_s)$ gilt, dass $h(t) = (h(a_1), ..., h(a_s))$ ist).

Ein Homomorphismus h von einer Konjunktion relationaler Atome $\lambda(\mathbf{x})$ auf eine Datenbankinstanz J ist eine Abbildung der Variablen $x \in \mathbf{x}$ nach KONST$(J) \cup$ NULL(J), sodass für jedes Atom R$(x_1, ..., x_n)$ aus $\lambda(\mathbf{x})$ das Faktum R$(h(x_1), ..., h(x_n))$ in J vorkommt.

Homomorphismen spielen eine wesentliche Rolle bei der Bestimmung von *Triggern*. Liegt ein Trigger für ein bestimmtes Tupel der Datenbank vor, so löst die Anwendung einer Integritätsbedingung durch den Chase aus – der Chase wird also „getriggert“.

Definition 2.4. (Trigger/aktiver Trigger, [BKM+17])
Sei I eine Instanz und sei τ eine TGD $\forall \mathbf{x} : \phi(\mathbf{x}) \to \exists \mathbf{z} : \psi(\mathbf{x}, \mathbf{z})$ oder EGD $\forall \mathbf{x} : \phi(\mathbf{x}) \to \mathbf{x}_i = \mathbf{x}_j$. Ein **Trigger** für τ in I ist ein Homomorphismus h von $\phi(\mathbf{x})$ nach I. Ein **aktiver Trigger** für τ in I ist ein Trigger h für τ in I, für den folgendes gilt:

- Ist τ eine TGD, existiert keine Erweiterung von h, die zu einem Homomorphismus von $\psi(\mathbf{x}, \mathbf{y})$ nach I führt.
- Ist τ hingegen eine EGD, so gilt $h(\mathbf{x}_i) \neq h(\mathbf{x}_j)$.

Beispiel 2.1. Unterschied zwischen aktivem und inaktivem Trigger
Wenden wir die TGD

$$r_1 : \text{No}(ma, mo, no) \to \exists NA, VO, ST, I : \text{St}(ma, NA, VO, ST, I)$$

auf die in Tabelle 2.1 gegebene Datenbankinstanz I_1 an, so lassen sich zwei Homomorphismen zu Tupeln dieser Instanz bilden: $h_1 = (ma \mapsto 18055, mo \mapsto 1789, no \mapsto 1.0)$ und $h_2 = (ma \mapsto 18051, mo \mapsto 1789, no \mapsto 1.3)$. Diese beiden Homomorphismen sind also Trigger von r_1. Allerdings lässt sich h_1 zu folgendem Homomorphismus in I_1 erweitern:

$$(ma \mapsto 18055, mo \mapsto 1789, no \mapsto 1.0, NA \mapsto \text{Mueller}, VO \mapsto \text{Max}, ST \mapsto \text{Elektrotechnik}, I \mapsto \text{IOF}).$$

Eine entsprechende Erweiterung von h_2 ist nicht möglich, da keine Tupel in I_1 den Wert 18051 im Attribut *Matrikelnummer* aufweist. Folglich sind also h_1 und h_2 Trigger, aber lediglich h_2 ist aktiv. □

Nachdem wir nun die wesentlichen Begriffe definiert haben, die für das Verständnis des Chase notwendig sind, wollen wir den Chase selbst definieren. Beginnen wir hierfür bei einem einzelnen Chase-Schritt.

Definition 2.5. (Standard-Chase-Schritt, nach [FKMP05])
Sei K eine Datenbankinstanz und sei r eine TGD $\phi(\mathbf{x}, \mathbf{y}) \rightarrow \exists \mathbf{z} : \psi(\mathbf{x}, \mathbf{z})$. Sei weiterhin h ein Homomorphismus von $\psi(\mathbf{x}, \mathbf{z})$ nach K, sodass keine Erweiterung von h zu einem Homomorphismus h' existiert, der $\phi(\mathbf{x}, \mathbf{y}) \wedge \psi(\mathbf{x}, \mathbf{z})$ auf K abbildet. Wir sagen in diesem Fall, dass r auf K unter dem Homomorphismus h **angewandt** werden kann. Sei K' die Vereinigung von K mit der Menge der Fakten, die in folgenden zwei Schritten erzeugt werden:

1. Erweitere h nach h', sodass jeder Variablen in $\mathbf{z}$ ein neuer markierter Nullwert zugewiesen wird,
2. $K' = K \cup \{h'(\psi(\mathbf{x}, \mathbf{z}))\}$ unter h'. Mit $K \xrightarrow{r,h} K'$ drücken wir aus, dass die Anwendung von r auf K unter h die Datenbankinstanz K' generiert.

Sei r die EGD $\phi(\mathbf{x}) \rightarrow \mathbf{x}_1 = \mathbf{x}_2$. Sei weiterhin h ein Homomorphismus von $\phi(\mathbf{x})$ nach K, sodass $h(\mathbf{x}_1) \neq h(\mathbf{x}_2)$. In diesem Fall sagen wir, dass r auf K unter h angewandt werden kann. Darüber hinaus unterscheiden wir folgende Fälle:

1. $h(\mathbf{x}_1) \in \text{KONST} \wedge h(\mathbf{x}_2) \in \text{KONST} \Rightarrow K \xrightarrow{r,h} \perp$ („Scheiternder Chase“),[1]
2. $h(\mathbf{x}_1) \in \text{KONST} \wedge h(\mathbf{x}_2) \in \text{NULL} \Rightarrow$ ersetze $h(\mathbf{x}_2)$ überall durch $h(\mathbf{x}_1)$,
3. $h(\mathbf{x}_1) \in \text{NULL} \wedge h(\mathbf{x}_2) \in \text{NULL} \Rightarrow$ ersetze $h(\mathbf{x}_2)$ überall durch $h(\mathbf{x}_1)$.

Scheitert der Chase nicht, so drücken wir mit $K \xrightarrow{r,h} K'$ (dem **Chase-Schritt**) aus, dass r auf K unter h angewandt wurde, um K' zu erzeugen.

Der Chase-Algorithmus besteht aus einer Folge der so definierten Chase-Schritte. Für die vorliegende Arbeit ist hierbei wesentlich, dass diese Folge auch eine unendliche Länge haben kann.

Definition 2.6. (Standard-Chase, nach [FKMP05])
Sei Σ eine Menge von TGDs und EGDs, des Weiteren sei K eine Datenbankinstanz. Eine **Chase-Sequenz** auf K mit Σ ist eine endliche oder unendliche Folge von Chase-Schritten $K_i \xrightarrow{r_i,h_i} K_{i+1}$ mit $i \in \mathbb{N}, K_0 = K, r_i \in \Sigma$.

Ein endlicher Chase auf K mit Σ ist eine endliche Folge $K_i \xrightarrow{r_i,h_i} K_{i+1}$ mit $0 \leq i < m$, wobei wir zwei Fälle unterscheiden:

1. $K_m = \perp$,
2. Es existiert weder ein $r_m \in \Sigma$ noch ein Homomorphismus h_m, sodass r_m auf K_m unter h_m angewandt werden könnte.

K_m stellt bildet das Ergebnis des endlichen Chase, wobei 1) als **scheiternder endlicher Chase** und 2) als **erfolgreicher endlicher Chase** bezeichnet wird.

Es folgt eine Darstellung des Chase als Pseudocode (Algorithm 1, nach [BKM+17]). Wir gehen hier davon aus, dass der Chase eine Menge Integritätsbedingungen Σ in eine Datenbankinstanz I einbaut. Varianten des Chase, die hiervon abweichende Eingangsparameter annehmen, werden in Abschnitt 2.2.2 näher erläutert.

[1] Mit $h(\mathbf{x}_1) \in \text{KONST} \wedge h(\mathbf{x}_2) \in \text{KONST}$ wird hier ausgedrückt, dass eine Komponente des Vektors $h(\mathbf{x}_1)$ und die zugehörige Komponente des Vektors $h(\mathbf{x}_2)$ Konstanten sind. Für Fall 2 und 3 können $\mathbf{x}_1$ und $\mathbf{x}_2$ auch vertauscht werden.

Algorithm 1 STANDARD-CHASE(Σ, I)

```
1:  ΔI := I
2:  while ΔI ≠ ∅ do
3:    N := ∅, m := ∅
4:    for each τ ∈ Σ with body λ(x⃗) do
5:      for each  trigger h for τ in I such that h(λ(x⃗)) ∩ ΔI ≠ ∅ do
6:        if h is an active trigger for τ in μ(N ∪ I) then
7:          if τ = ∀x⃗λ(x⃗) → ∃x⃗ρ(x⃗, y⃗) is a TGD  then
8:            h' := h ∪ {y⃗ ↦ v⃗} where v⃗ ⊆ NULL is fresh
9:            N := N ∪ h'(ρ(x⃗, y⃗))
10:         else if τ = ∀x⃗λ(x⃗) → x_i = x_j is an EGD  then
11:           if {h(x_i) ≠ h(x_j)} ⊆ KONST then
12:             fail
13:           end if
14:           ω := {max(h(x_i), h(xj)) → min(h(x_i), h(x_j))}
15:           μ := μ ∘ (ω ∘ μ)
16:         end if
17:       end if
18:     end for
19:   end for
20:   ΔI := μ(N ∪ I) \ I
21:   I := μ(I) ∪ ΔI
22: end while
```

Beispiel 2.2. Durchführung des Standard-Chase

Im Folgenden sollen die wesentlichen Zeilen des obigen Pseudocodes näher erläutert werden. Betrachten wir hierfür die TGD r_1, die wir bereits in Beispiel 2.1 untersucht hatten:

$$r_1 : \mathrm{No}(ma, mo, no) \rightarrow \exists NA, VO, ST, SE, I : \mathrm{St}(ma, NA, VO, ST, I).$$

Gefordert wird hier lediglich, dass die Notentabelle ein Tupel enthält. Dies ist für die in Tabelle 2.1 dargestellt Datenbankinstanz der Fall – es existieren (wie im vorherigen Beispiel gezeigt) zwei Trigger (Zeile 5), von denen nur einer aktiv ist (Zeile 6): Wenden wir r_1 auf das Tupel No(18055, 1789, 1.0) an, so würde ein Tupel erzeugt werden (St(18055, η_1, η_2, η_3 , η_4)), für das ein Homomorphismus zum bereits in der Datenbank vorhandenen Tupel St(18055, Mueller, Max, Elektrotechnik, IOF) existiert – dieser Trigger ist also nicht aktiv. Ergänzen wir den verbliebenen (aktiven) Trigger

$$h_2' = (ma \mapsto 18051, mo \mapsto 1789, no \mapsto 1.3, NA \mapsto \eta_1, VO \mapsto \eta_2, ST \mapsto \eta_3, SE \mapsto \eta_4, I \mapsto \eta_5)$$

mit Abbildungen auf neue, markierte Nullwerte (Zeile 8) und wenden ihn auf den Kopf von r_1 an (Zeile 9). Wir generieren (da r_1, wie in Zeile 7 überprüft, eine *Tuple Generating Dependency* ist) das Tupel St(18051, η_1, η_2, η_3, η_4), das der Datenbankinstanz hinzugefügt wird (Zeile 9, gelb markiert in Tabelle 2.2). Durch das Ergänzen von h_2 um Abbildungen auf Nullwerte, die bis zur Generierung des neuen Tupels nicht in der Datenbankinstanz vorkommen, verliert r_2 vorübergehend seine Homomorphismuseigenschaften und ist folglich bis zum Einfügen des Tupels kein Trigger mehr.

Fordern wir nun – um Notengerechtigkeit zu garantieren – dass alle Studenten eines Moduls die gleiche Note erhalten müssen. Hierfür führen wir folgende EGD ein:

$$r_2 : \mathrm{No}(ma1, mo, no1), \mathrm{No}(ma, mo, no2) \rightarrow no1 = no2.$$

Die Vorgehensweise ist bis Zeile 6 ähnlich zum gerade beschriebenen Beispiel: Erneut generieren wir Homomorphismen: $h_3 = (ma1 \mapsto 18055, mo \mapsto 1789, no1 \mapsto 1.0, ma2 \mapsto 18051, no2 \mapsto 1.3)$, $h_4 = (ma1 \mapsto$

$18055, mo \mapsto 1789, no1 \mapsto 1.0, ma2 \mapsto 18055, no2 \mapsto 1.0)$ und $h_5 = (ma1 \mapsto 18051, mo \mapsto 1789, no1 \mapsto 1.3, ma2 \mapsto 18051, no2 \mapsto 1.3)$. Von diesen ist jedoch nur einer (h_3) aktiv. Da für h_4 nicht nur $1.0 \neq 1.3$, sondern auch $\{1.0, 1.3\} \subseteq$ KONST gilt (Zeile 11), scheitert der Chase (Zeile 12).

Noten

Matrikelnummer	Modulnummer	Note
18055	1789	1.0
18051	1789	1.3

Studenten

Matrikelnummer	Nachname	Vorname	Studiengang	Institut
18055	Mueller	Max	Elektrotechnik	IOF

Tabelle 2.1.: Ursprüngliche Datenbankinstanz.

Studenten

Matrikelnummer	Nachname	Vorname	Studiengang	Institut
18055	Mueller	Max	Elektrotechnik	IOF
18051	η_1	η_2	η_3	η_4

Tabelle 2.2.: Studententabelle nach Anwendung des Standard-Chase. Neu eingefügte Tupel sind gelb markiert.

□

Ein Beispiel für einen erfolgreichen Chase auf einer EGD findet sich im nachfolgenden Unterkapitel 2.2.2, in dem wir uns mit Varianten des zuvor dargestellte Standard-Chase beschäftigen. Besonderen Fokus legen wir hierbei auf die Terminierungseigenschaften der Chase-Varianten.

2.2.2. Variationen des Chase

Wie zuvor erwähnt, baut der Chase-Algorithmus Parameter ($\tau \in \Sigma$ im Pseudocode) in ein Objekt (I) ein. Konkret handelt es sich hierbei unter anderem um um:

- Integritätsbedingungen (Parameter) und Datenbankschemata (Objekt)
- Integritätsbedingungen (Parameter) und (mitunter unvollständige) Datenbankinstanzen (Objekt)
- Integritätsbedingungen (Parameter) und Anfragen (Objekt)
- Sichten (Parameter) und Anfragen (Objekt)
- Operationen (Parameter) und Anfragen (Objekt)

Prinzipiell ist es für den Chase (und die Terminierung des Chase) gleichgültig, welche dieser Varianten praktisch angewandt wird. Wir werden uns im Folgenden (wie bereits in Beispiel 2.2) auf das Einbauen von Integritätsbedingungen in Datenbankinstanzen beschränken. Bedeutsamer für die vorliegende Arbeit sind aber Variationen des Algorithmus selbst. Neben dem im vorherigen Unterkapitel 2.2.1 dargestellten *Standard-Chase* (bzw. Restricted Chase) gibt es Formen des Chase, die von diesem abweichende Terminierungseigenschaften aufweisen.

Von den verschiedenen Varianten des Oblivious Chase werden wir hier nur zwei näher darstellen: Den *Naiven Oblivious Chase* (Unrestricted Chase) und den *Skolem-Oblivious Chase* (Semirestricted Chase). Beiden Formen des Chase ist gemein, dass ein Testen des aktiven Trigger unterbleibt. Im zuvor gezeigten Pseudocode wird infolgedessen auf die rot markierte Zeile 6 verzichtet. Bei Vorliegen eines Triggers wird das neue Tupel also ohne die Überprüfung möglicher Erweiterungen des Triggers generiert. für jede existenzquantifizierte Variable des Kopfes der TGD einen Nullwert enthält. wenn der Trigger In praktischen Implementierungen des Chase geschieht dies aus Effizienzgründen, da das Testen des aktiven Triggers komplex und zeitaufwendig sein kann. In der vorliegenden Arbeit wird der naive Chase jedoch aus zwei anderen Gründen benötigt:

- Unabhängigkeit von der Reihenfolge der Chase-Schritte (siehe Beispiel 3.3),
- Möglichkeit, eine minimale Datenbankinstanz zu verwenden (siehe Unterabschnitt 3.1.2).

Hierbei ist anzumerken, dass dies keine generellen Vorteile des Oblivious Chase sind – es handelt sich vielmehr um Berührungspunkte der vorliegenden Arbeit mit zwei Variationen des Chase, die (noch) nicht in ChaTEAU implementiert sind. Bei Verwendung allgemeiner TGDs[2] weist der Standard-Chase die ungünstige Eigenschaft auf, weder konfluent, noch terminierend zu sein. Auch die Terminierung des naiven Chase ist nicht garantiert (tatsächlich terminiert der Standard-Chase sogar in Fällen, in denen der Oblivious Chase nicht terminiert), allerdings ist die Terminierung des Oblivious Chase nicht reihenfolgeabhängig [GMS12]. Bei der Implementierung eines Tests auf Terminierung in ein Chase-Tool (d.h. der Aufgabenstellung dieser Arbeit) ist es nicht von Interesse, ob es eine terminierende Folge von Chase-Schritten gibt, da das Tool nicht notwendigerweise diese Folge wählt.

Für das Terminierungskriterium der *Stratifikation* ist in diesem Sinne von Interesse, dass wir das Triggerverhalten von zwei TGDs r_1 und r_2 unabhängig von einer zwischenzeitlich getriggerten dritten TGD r_3 untersuchen können, wir können also stets davon ausgehen, dass dass die Reihenfolge der Trigger r_1, r_2, r_3 ist. Sie kann also vernachlässigt werden.

Obwohl wir in der vorliegenden Arbeit nicht tiefer darauf eingehen werden, kann es sinnvoll sein, den Chase-Algorithmus testweise auf einer „einfachen“ Datenbankinstanz (für das semi-dynamische Kriterium „kritische Instanz“ genannt) anzuwenden. Eine derartige Instanz enthält (von Konstanten abgesehen) in allen Attributen denselben Eintrag ($*$ für das semi-dynamische Kriterium). Terminierung ist für diese Instanz also nicht von zufälligen „Mustern“ der Datenbankeinträge abhängig. Der Adornment-Algorithmus wendet denselben Gedankengang an, indem Variablen, die vorhandene Werte der Datenbankinstanz tragen, mit dem selben Wert „b“ adornt werden, und nur Variablen, die Nullwerte tragen, mit unterscheidbaren f_i-Adornments versehen werden.

Eine solche Datenbankinstanz kann – wenn wir auf Konstanten verzichten – niemals einen aktiven Trigger generieren, da alle generierten Tupel in jeder Position eben jenen bereits in der Zielrelation vorhanden Eintrag aufweisen.

Wie zuvor angemerkt terminiert der Oblivious Chase in einigen Situationen nicht, in denen der Standard-Chase terminieren würde. Reziprok terminiert der Standard-Chase jedoch immer, wenn ein Oblivious Chase terminieren würde, d.h. die durch die beiden Chase-Arten definierten Terminierungsklassen von TGD-Mengen sind echt ineinander enthalten.

Der Naive Oblivious Chase kann jedoch durch *Skolemisierung* erweitert werden, sodass seine Terminierungseigenschaften dem Standard-Chase ähnlicher werden, ohne dass er die beiden zuvor genannten

[2] EGDs betrachten wir an dieser Stelle nicht.

positiven Eigenschaften des Oblivious Chase verliert. Skolemisierung bedeutet hier, dass existenzquantifizierte Variablen $\mathbf{z}$ durch die Skolemfunktion $f^r_{\mathbf{z}}(\mathbf{x})$ substituiert werden, wobei r die Integritätsbedingung ist und $\mathbf{x}$ für den Vektor der in Kopf und Körper dieser Integritätsbedingung vorkommenden Variablen steht. Nach Instanziierung der Skolemfunktion wird sie durch einen markierten Nullwert ersetzt.

Beispiel 2.3. Vergleich der drei vorgestellten Chase-Arten (terminierender Chase)
Vergleichen wir nun die drei vorgestellten Formen des Chase. Hierfür betrachten wir zunächst Datenbankinstanz und TGD (r_1) aus Beispiel 2.2:

$r_1 : \mathrm{No}(ma, mo, no) \rightarrow \exists NA, VO, ST, SE, I : \mathrm{St}(ma, NA, VO, ST, SE, I).$

Wie zuvor gezeigt, generiert der Standard-Chase ein zusätzliches Tupel in der Studententabelle. Da die beiden Varianten des Oblivious Chase nicht überprüfen, ob Trigger aktiv sind, erzeugen sie ein zusätzliches Tupel in der Studententabelle, da sie auch das Tupel $\mathrm{No}(18055, 1789, 1.0)$ der Notentabelle berücksichtigen. Der Skolem-Oblivious Chase substituiert die existenzquantifizierten Variablen zunächst durch Skolemfunktionen (und erzeugt hierdurch Tabelle 2.4 b), die in einem zweiten Schritt durch markierte Nullwerte ersetzt werden (Tabelle 2.4 c). Der Naive Oblivious Chase erzeugt hingegen sofort für jede existenzquantifizierte Variable einen markierten Nullwert, kommt aber zum gleichen Ergebnis wie der Skolem-Oblivious Chase.

Noten

Matrikelnummer	Modulnummer	Note
18055	1789	1.0
18051	1789	1.3

Studenten

Matrikelnummer	Nachname	Vorname	Studiengang	Institut
18055	Mueller	Max	Elektrotechnik	IOF

Tabelle 2.3.: Ursprüngliche Datenbankinstanz

Führen wir nun ein Cleaning der Datenbank durch. Wir nehmen an, dass über der Studententabelle die funktionellen Abhängigkeiten Matrikelnummer $\rightarrow$ Nachname und Matrikelnummer $\rightarrow$ Vorname bestehen. Wir schließen also nicht aus, dass Studenten mehrere Studiengänge besuchen und für verschiedene Institute eingetragen sind, aber zumindest der Name eines (durch seine Matrikelnummer identifizierten) Studenten sollte eindeutig sein. Dieser Sachverhalt entspricht den folgenden EGDs:

$r_4 : \mathrm{St}(ma, na1, vo1, st1, se1, in1), \mathrm{St}(ma, na2, vo2, st2, se2, in2) \rightarrow na1 = na2$
$r_5 : \mathrm{St}(ma, na1, vo1, st1, se1, in1), \mathrm{St}(ma, na2, vo2, st2, se2, in2) \rightarrow vo1 = vo2.$

Im Gegensatz zu Beispiel 2.2 erreichen wir Zeile 14 des Pseudocodes und bilden das Minimum von η_1 und der Konstanten Mueller sowie von η_2 und der Konstanten Max. Während wir beim Vergleich zweier markierter Nullwerte hier den Index der Nullwerte vergleichen würden, ist das Minimum einer Konstanten und eines Nullwertes stets die Konstante. Wir substituieren also in der gesamten Datenbankinstanz η_1 durch Mueller und η_2 durch Max (Zeile 20) und erzeugen die Studententabelle aus Tabelle 2.5:

□

a)

Matrikelnummer	Nachname	Vorname	Studiengang	Institut
18055	Mueller	Max	Elektrotechnik	IOF
18051	η_1	η_2	η_3	η_4

b)

Matrikelnummer	Nachname	Vorname	Studiengang	Institut
18055	Mueller	Max	Elektrotechnik	IOF
18055	$f^{r_1}_{NA}(18055)$	$f^{r_1}_{VO}(18055)$	$f^{r_1}_{ST}(18055)$	$f^{r_1}_{IN}(18055)$
18055	$f^{r_1}_{NA}(18051)$	$f^{r_1}_{VO}(18051)$	$f^{r_1}_{ST}(18051)$	$f^{r_1}_{IN}(18051)$

c)

Matrikelnummer	Nachname	Vorname	Studiengang	Institut
18055	Mueller	Max	Elektrotechnik	IOF
18055	η_1	η_2	η_3	η_4
18051	η_5	η_6	η_7	η_8

Tabelle 2.4.: Standard-Chase und (Naiver bzw. Skolem-) Oblivious Chase erzeugen bei Anwendung von r_1 auf die Datenbankinstanz von Tabelle 2.3 jeweils unterschiedliche Instanzen der Studententabelle. Durch den Chase generierte Tupel sind gelb markiert.
a) Studententabelle nach Anwendung des Standard-Chase.
b) Zwischenzustand der Studententabelle nach Anwendung des Skolem-Oblivious Chase.
c) Studententabelle nach Anwendung des Naiven Oblivious Chase oder – nach Ersetzen der Skolemfunktionen durch Nullwerte – des Skolem-Oblivious Chase.

Matrikelnummer	Nachname	Vorname	Studiengang	Institut
18055	Mueller	Max	Elektrotechnik	IOF
18055	Mueller	Max	η_3	η_4
18051	η_5	η_6	η_7	η_8

Tabelle 2.5.: Studententabelle nach Anwendung der EGDs r_4 und r_5 (mit einer beliebigen Variante des Chase). Durch den Chase generierte Tupel sind gelb markiert.

Beispiel 2.4. Vergleich der drei vorgestellten Chase-Arten (Naiver Oblivious Chase terminiert nicht)
Im vorherigen Beispiel 2.3 terminierten alle drei Varianten des Chase. Darüber hinaus kamen Naiver Oblivious Chase und Skolem-Oblivious Chase letztendlich zu identischen Ergebnissen. Im folgenden Beispiel terminieren hingegen Standard-Chase und Skolem-Oblivious Chase (mit unterschiedlichen Ergebnisinstanzen), während der Naive Oblivious Chase nicht terminiert. Betrachten wir hierfür folgende TGD:

$r_4 : \mathrm{St}(ma, na_1, vo_1, st_1, in_1) \rightarrow \exists : \mathrm{St}(ma, NA_2, VO_2, ST_2, IN_2)$

Der hier beschriebene Sachverhalt – wenn ein Eintrag für einen Studenten in der Studententabelle steht, so ist ein Tupel mit seiner Matrikelnummer in der Studententabelle vorhanden – ist eine Tautologie. Das Erzeugen neuer Tupel ist also nicht nötig, um r_4 zu erfüllen. Folglich existiert auch kein aktiver Trigger für den Standard-Chase – der Chase terminiert also (Tabelle 2.6 a). Der Skolem-Oblivious Chase hingegen erzeugt – wie in Tabelle 2.6 a) zu sehen ist – zumindest ein neues Tupel. Wir gehen hier von der ursprünglichen Datenbankinstanz aus Beispiel 2.3 (Tabelle 2.3) aus, auf einer leeren Studententabelle würden selbstverständlich alle drei Chase-Varianten terminieren, ohne neue Tupel zu generieren.

Erneute Anwendung des Skolem-Oblivious Chase würde das bereits in der Datenbank vorhandene Tupel erzeugen, da auch die Skolemfunktionen dieses neuen Tupels vom Wert 18055 abhängen (und da eine Relation eine Menge von Tupeln darstellt, wird das zusätzliche Tupel nicht erzeugt). Nach Ersetzen der Skolemfunktionen durch Nullwerte entsteht Tabelle 2.6 b). Der Skolem-Oblivious Chase terminiert also ebenfalls.

Der Naive Oblivious Chase hingegen terminiert nicht (Tabelle 2.6 c). Jedes erzeugte Tupel aus Nullwerten liefert einen neuen Trigger und generiert ein Tupel aus neuen Nullwerten, da der aktive Trigger nicht überprüft wird.

a)

Matrikelnummer	Nachname	Vorname	Studiengang	Institut
18055	Mueller	Max	Elektrotechnik	IOF
18055	$f^{r_1}_{NA}(18055)$	$f^{r_1}_{VO}(18055)$	$f^{r_1}_{ST}(18055)$	$f^{r_1}_{IN}(18055)$

b)

Matrikelnummer	Nachname	Vorname	Studiengang	Institut
18055	Mueller	Max	Elektrotechnik	IOF
18055	η_1	η_2	η_3	η_4

c)

Matrikelnummer	Nachname	Vorname	Studiengang	Institut
18055	Mueller	Max	Elektrotechnik	IOF
18055	η_1	η_2	η_3	η_4
18055	η_5	η_6	η_7	η_8
...	...	...	...	...

Tabelle 2.6.: a) Zwischenzustand der Studententabelle nach Anwendung des Skolem-Oblivious Chase.
b) Studententabelle nach Ersetzen der Skolemfunktionen aus a) durch Nullwerte.
c) Studententabelle bei Anwendung des Naiven Oblivious Chase (da der Algorithmus im vorliegenden Beispiel nicht terminiert, kann der Zustand der Tabelle „nach“ Anwendung des Chase nicht angegeben werden). Durch den Chase generierte Tupel sind gelb markiert.

□

Im letzten Beispiel konnten wir beobachten, dass der Naive Oblivious Chase nicht immer terminiert. Aber auch das Terminieren des Standard-Chase ist keinesfalls sichergestellt. Im folgenden Kapitel werden

wir Kriterien kennenlernen, die das Terminieren des Chase garantieren – und zwar unabhängig von der verwendeten Datenbankinstanz.

3. Stand der Forschung und Technik

Im folgenden Kapitel soll der aktuelle Stand der Forschung und Technik zum Thema Chase-Terminierung beschrieben werden. Stand der Forschung bezieht sich auf die theoretische Definition von Terminierungskriterien (Abschnitt 3.1), während Stand der Technik für deren praktische Umsetzung in existierenden Chase-Werkzeugen steht (Abschnitt 3.2). Die Darstellung beruht im Wesentlichen auf der Analyse aktueller Forschungsliteratur, wobei die Notation entsprechend angepasst wurde. Vorgefundene Beispielen wurden transformiert und auf die von uns verwendete Studentendatenbank übertragen. Bei der Evaluation der Chase-Werkzeuge werden wir darüber hinaus auch auf eigene praktische Tests zurückgreifen.

3.1. Terminierungskriterien

Die Terminierung des Chase ist nicht entscheidbar. Dies betrifft sowohl die Terminierung des Chase auf einer bestimmten Datenbankinstanz als auch seine Terminierung auf einer beliebigen Instanz. Wir können jedoch Klassen von Abhängigkeitsmengen definieren, für die der Chase sicher terminiert. Für bestimmte Datenbankinstanzen sind diese Terminierungsklassen durch semi-dynamische Kriterien beschrieben (siehe Unterabschnitt 3.1.2), während statische Terminierungskriterien Terminierungsklassen auf allgemeinen Datenbankinstanzen definieren (erläutert im folgenden Unterabschnitt 3.1.1).

3.1.1. Statische Kriterien

Das einfachste – und zuerst beschriebene – Terminierungskriterium ist die „Azyklizität“ (nicht zu verwechseln mit dem gleichnamigen Terminierungskriterium des Adornment-Algorithmus). Wenn kein Zyklus aus TGDs mit paarweise gleichen Relationsnamen in Kopf und Körper existiert, so terminiert der Chase garantiert. Dies stellt allerdings eine starke Einschränkung dar. Ein weiteres grundlegendes Terminierungskriterium betrifft das Vorkommen von existenzquantifizierten Variablen: Da das zyklische Erzeugen neuer markierter Nullwerte ein charakteristisches (und sogar notwendiges) Merkmal des nicht terminierenden Chase darstellt, ist die Terminierung des Chase garantiert, wenn alle TGDs voll sind – Zyklen voller TGDs sind also unerheblich für die Terminierung des Chase. Allerdings ist auch dieses Kriterium noch sehr einschränkend. Im Folgenden sollen Kriterien vorgestellt werden, welche die Terminierung des Chase in Gegenwart existenzquantifizierter Variablen erkennen können.

Wir gehen von einer beliebigen, jedoch endlichen Datenbankinstanz aus. Angenommen, wir haben einen (an dieser Stelle noch vage definierten) Zyklus sich gegenseitig aufrufender voller TGDs, so können diese nur eine endliche Anzahl sich von einander unterscheidender Tupel generieren (da sie auf eine endliche Menge von Werten aus der Ausgangsdatenbankinstanz beschränkt sind). Letztendlich terminiert der Chase also, da keine der TGDs einen aktiven Trigger aufweist.

Eingebettete TGDs hingegen können neue (Null-)Werte erzeugen, die nicht in der Ausgangsdatenbank standen. Allerdings erzeugt nicht jede existenzquantifizierte Variable automatisch einen neuen Nullwert: Wenn die betreffende Relation bereits ein Tupel enthält, das sich nur an den Stellen der potentiellen Nullwerte vom potentiell neu generierten Tupel unterscheidet, liegt kein aktiver Trigger vor und es wird kein neues Tupel erzeugt. Wir müssen also mindestens einen Wert in das Tupel übertragen, der noch nicht in der Relation steht, und da wir von jeder möglichen Ausgangsdatenbankinstanz ausgehen können, muss dieser Wert ein durch eine TGD erzeugter Nullwert sein. Der Nullwert kann trotzdem bereits in der Relation stehen – wir können sogar im Körper der TGD voraussetzen, dass sich der Nullwert bereits an der entsprechenden Stelle der Relation befindet, siehe Beispieldatei `Beispiel17_Implication.txt`.

Im Folgenden betrachten wir verschiedene Terminierungskriterien. Wir starten dabei mit der Schwachen Azyklizität.

Schwache Azyklizität

Das Kriterium der *Schwachen Azyklizität* wurde in [FKMP03] entwickelt, parallel hierzu entstand das verwandte Kriteriums „Bedingungen mit Stratifizierten Zeugen“ in [DT03]. Schwache Azyklizität überprüft für jede eingebettete TGD eines Zyklus, ob ein im bisherigen Zyklusverlauf erzeugter Nullwert in den Kopf der TGD übertragen wird. Hierbei müssen existenzquantifizierte Variable und übertragener Nullwert nicht notwendigerweise im selben Atom der rechten Seite stehen. Sind TGDs *schwach azyklisch*, so ist die Terminierung des Skolem-Oblivious Chase – und somit auch des Standard-Chase – garantiert, nicht jedoch die Terminierung des Naiven Oblivious Chase. Die Terminierung dieser Chase-Variante wird durch das Terminierungskriterium der Reichen Azyklizität sichergestellt.

Definition 3.1. (Schwache Azyklizität, [GMS12])
Sei Σ eine Menge von TGDs über einem Datenbankschema $\mathbf{R}$. pos(Σ) gibt eine Menge von **Positionen** R_i zurück, wobei R ein relationales Prädikat ist und ein R-Atom in Σ vorkommt. i kennzeichnet hier ein Attribut von R ($i \in \mathbb{N}^+, 1 \leq i \leq$ Stelligkeit(R)). **Schwache Azyklizität** (WA) basiert auf der Konstruktion eines gerichteten Graphen dep(Σ) = (pos(Σ), E) – dem **Abhängigkeitsgraphen** – wobei E wie folgt definiert ist: Für jede TGD $\phi(\mathbf{x}, \mathbf{z}) \rightarrow \exists \mathbf{y} : \psi(\mathbf{x}, \mathbf{y}) \in \Sigma$ und

1. für jedes $x \in \mathbf{x}$, das in Position R_i in ϕ und Position S_j in ψ vorkommt: ziehe eine Kante von R_i nach S_j (wenn noch keine derartige Kante existiert),
2. für jedes $x \in \mathbf{x}$, das in ϕ in Position R_i vorkommt und für jedes $y \in \mathbf{y}$, das in ψ in Position T_k vorkommt: ziehe eine besondere Kante[1] von R_i nach T_k (wenn noch keine derartige Kante existiert).

Σ ist schwach azyklisch, wenn dep(Σ) keinen Zyklus aufweist, der durch eine besondere Kante führt.

Das Kriterium der **Reichen Azyklizität** [HS07], das die Terminierung des Naiven Oblivious Chase garantiert, basiert ebenfalls auf einem Abhängigkeitsgraphen E, der wie folgt definiert ist: Für jede TGD $\phi(\mathbf{x}, \mathbf{z}) \rightarrow \exists \mathbf{y} : \psi(\mathbf{x}, \mathbf{y}) \in \Sigma$ und

1. für jedes $x \in \mathbf{x}$, das in Position R_i in ϕ und Position S_j in ψ vorkommt: ziehe eine Kante von R_i nach S_j (wenn noch keine derartige Kante existiert),
2. für jedes $x \in \mathbf{x}$ und jedes $z \in \mathbf{z}$, das in ϕ in Position R_i vorkommt und für jedes $y \in \mathbf{y}$, das in ψ in Position T_k vorkommt: ziehe eine spezielle Kante von R_i nach T_k (wenn noch keine derartige Kante existiert).

[1] Derartige Kanten werden in [GO18] generierende Kanten genannt, was ihre Funktion – die Generierung neuer Nullwerte und somit neuer Tupel – adäquat beschreibt.

Σ ist reich azyklisch, wenn dep(Σ) keinen Zyklus aufweist, der durch eine spezielle Kante führt.

Im Folgenden wollen wir den Unterschied zwischen Schwacher und Reicher Azyklizität anhand eines Beispiels illustrieren. Die gewählten Integritätsbedingungen sind schwach azyklisch, aber nicht reich azyklisch.

Beispiel 3.1.

- Fall: Schwach azyklische TGDs
- Chase: terminiert

Betrachten wir die folgenden beiden TGDs:
$r_1 : \text{St}(ma, na, vo, st, in) \rightarrow \exists FS : \text{Fä}(st, in, FS)$
$r_2 : \text{Fä}(st, in, fs) \rightarrow \exists NA, VO : \text{St}(fs, NA, VO, st, in)$.

Die erste TGD erzeugt zwar ein Tupel mit einem potentiellen Nullwert in der Fächer-Relation, da jedoch keiner der anderen Werte dieses Tupels ein durch r_1 oder r_2 erzeugter Nullwert sein kann (also auf den endlichen bereits in der Datenbankinstanz vorkommenden Wertebereich beschränkt ist), wird die TGD bei erneutem (zyklischen) Feuern die existenzquantifizierte Variable FS auf den im vorherigen Durchlauf erzeugten Nullwert abbilden. In diesem Fall liegt kein aktiver Trigger mehr vor, der Chase terminiert also.

Der zugehörige Abhängigkeitsgraph (Abbildung 3.1a) wird folgendermaßen konstruiert. Für jedes Attribut jeder Relation erstellen wir einen Knoten (die Position), für das vierte Attribut der Studentenrelation (Studiengang) also beispielsweise St_4. Da die Variable st nicht nur im Körper, sondern auch im Kopf von r_1 vorkommt (dort nämlich im ersten Attribut der Fächerrelation), ziehen wir eine normale Kante zwischen den beteiligten Knoten (also zwischen St_4 und Fä_1). Da in r_1 außerdem die existenzquantifizierte Variable FS vorkommt (und zwar in der dritten Position der Fächerrelation), ziehen wir zusätzliche eine besondere Kante zwischen St_4 und Fä_3. Die übrigen Kanten werden auf die gleiche Art erstellt. Da kein Zyklus beobachtbar ist, der durch eine besondere Kante geht, sind die TGDs schwach azyklisch, und die zuvor argumentativ begründete Terminierung des Chase konnte bestätigt werden. Wenden wir das Kriterium der Reichen Azyklizität an, ersetzen wir im soeben konstruierten Abhängigkeitsgraphen besondere durch spezielle Kanten und ergänzen ihn um spezielle Kanten, die von St_1, St_2 und St_3 nach Fä_3 führen (Abbildung 3.1b). Hierdurch entstehen mehrere Zyklen durch spezielle Kanten, beispielsweise von St_2 nach Fä_3 und wieder zurück zu St_2. Die TGDs sind also nicht reich azyklisch, und der Naive Oblivious Chase terminiert im Gegensatz zum Standard-Chase wahrscheinlich nicht. □

Beispiel 3.2.

- Fall: TGDs, die nicht schwach azyklisch sind
- Chase: terminiert

Um zu veranschaulichen, dass das Kriterium der Schwachen Azyklizität nicht ausreicht, um in allen Fällen die Terminierung des Chase korrekt vorherzusagen, betrachten wir die folgenden beiden TGDs:
$r_1 : \text{No}(ma, mo, no), \text{St}(ma, na, vo, st, in) \rightarrow \exists FS : \text{Fä}(st, in, FS)$
$r_2 : \text{Fä}(st, in, fs) \rightarrow \exists NA, VO, IN : \text{St}(fs, NA, VO, st, IN)$.

Wir erzeugen ein Tupel in der Fächer-Relation, welches sowohl einen potentiellen neuen Nullwert im Attribut Fachschaftsratsprecher enthält, und außerdem – wenn die zweite TGD ausgelöst wird – einen durch diese erzeugten Nullwert im Institut. Auf diese Weise können also beliebig viele Tupel in der Fächer-Relation entstehen, obwohl der bereits in der Datenbankinstanz vorhandene Wertebereich endlich ist. Im Abhängigkeitsgraph manifestiert sich dies in einem Zyklus aus besonderen Kanten zwischen Fa_3 und St_5

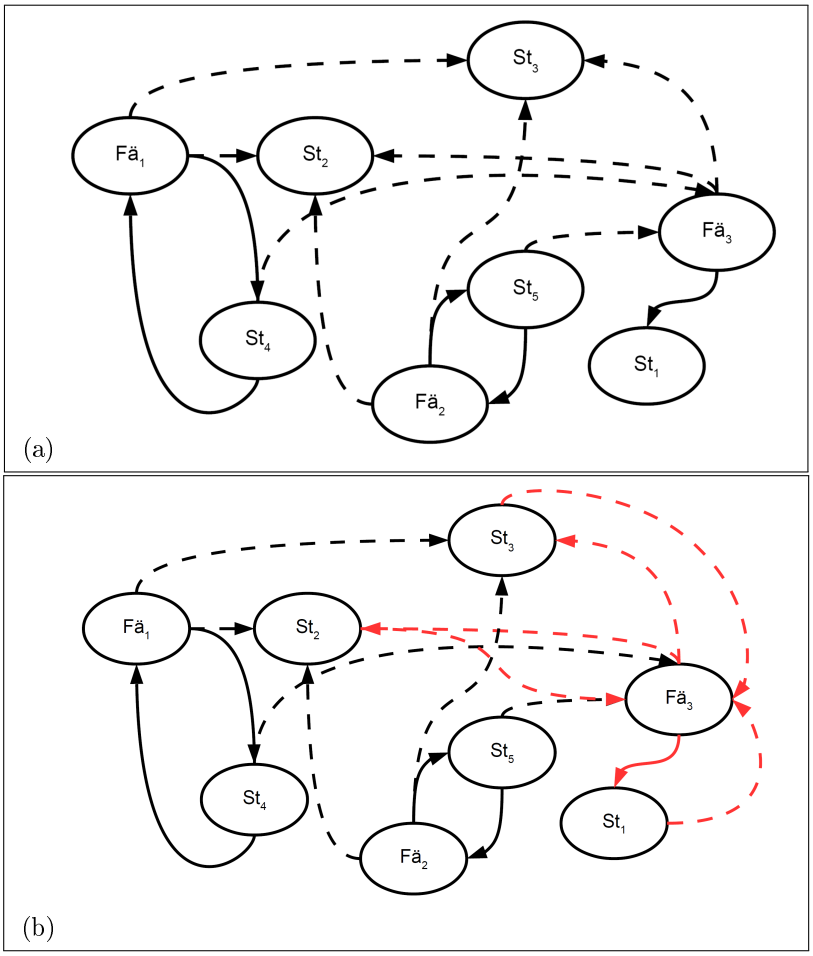


Abbildung 3.1.: Abhängigkeitsgraph der Schwachen Azyklizität für Beispiel 3.1 (a). Kein Zyklus mit besonderen Kanten (gestrichelt dargestellt) ist erkennbar, weshalb die Terminierung des Standard-Chase garantiert ist. Der Naive Oblivious Chase hingegen terminiert wahrscheinlich nicht, da Zyklen (rot dargestellt) durch spezielle Kanten (ebenfalls gestrichelt dargestellt) des Abhängigkeitsgraphen der Reichen Azyklizität existieren (b).

(siehe Abbildung 3.2). Tatsächlich wird r_1 jedoch nicht zyklisch durch r_2 ausgelöst, da r_1 zusätzlich ein Tupel in der Noten-Relation voraussetzt, das die selbe Matrikelnummer wie das zyklisch erzeugte Tupel der Studenten-Relation enthält. Wenn wir den Wert dieses Attributs jedoch zurück verfolgen, stellen wir fest, dass es einen Nullwert tragen muss – es handelt sich nämlich um den durch r_1 im vorherigen Zyklus mit einem Nullwert belegten Fachschaftsratsprecher. Dieser Nullwert kann (da er neu ist) nicht in der Ausgangsdatenbank stehen , und es gibt auch keine TGD, die den Nullwert in die Notentabelle überführen konnte. Der Chase hält also an, da wir das gegenseitige Feuern von TGDs nicht ausreichend berücksichtigt haben. Dies geschieht durch das Kriterium der *(c-)Stratifizierung.* □

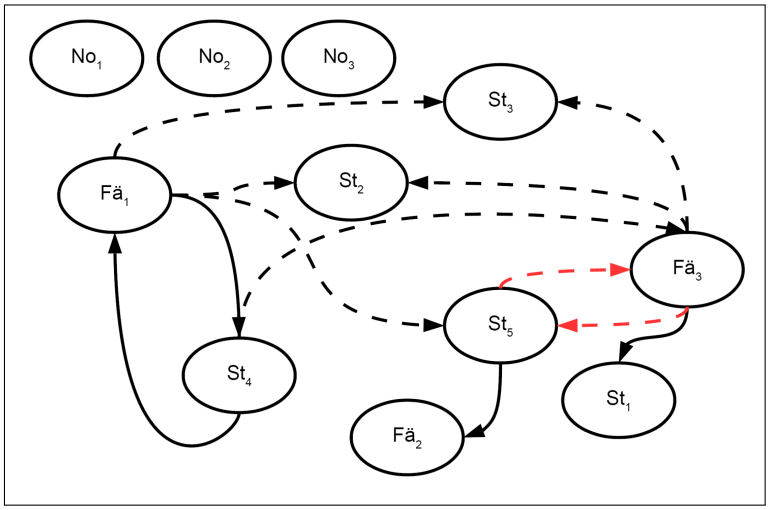


Abbildung 3.2.: Abhängigkeitsgraph für Beispiel 3.2. Der Zyklus (rot markiert) aus besonderen Kanten (gestrichelte Pfeile) zwischen Fa_3 und St_5 zeigt an, dass der Chase möglicherweise nicht terminiert – tatsächlich terminiert er jedoch aufgrund der Einbeziehung der Notenrelation, die im Abhängigkeitsgraph unberücksichtigt geblieben ist.

Stratifizierung

Stratifizierung wurde ursprünglich in [DNR08] definiert. Die Gesamtmenge der Integritätsbedingungen wird in Teilmengen unterteilt. Wird das Kriterium der Schwachen Azyklizität angewandt, so können Zyklen durch besondere Kanten ignoriert werden, die mehrere dieser Teilmengen überspannen. Es muss also nicht die Gesamtmenge der Regeln schwach azyklisch sein, sondern nur jede einzelne Teilmenge.

Definition 3.2. (Präzedenzrelation, [GMS12])
Gegeben seien eine Menge von Bedingungen Σ und zwei Bedingungen r_1, r_2 aus Σ. Wir definieren $r_1 < r_2$ wenn es eine relationale Datenbankinstanz K und zwei Homomorphismen h_1, h_2 gibt, sodass gilt:

1. $K \xrightarrow{r_1,h_1} J$,
2. $J \not\models h_2(r_2)$,
3. $K \models h_2(r_2)$.

Die so definierte Präzendenzrelation erlaubt es uns, einen *Chase-Graphen* zu erstellten. Im Gegensatz zum zuvor definierten Abhängigkeitsgraphen lassen sich aus Zyklen im Chase-Graphen noch keine Aussagen über die Terminierung des Chase treffen. Diese ergeben sich erst, indem TGDs jedes Zyklus auf Schwache Azyklizität getestet werden.

Definition 3.3. (Stratifizierung, [GMS12])
Der **Chase-Graph** $G(\Sigma) = (\Sigma, E)$ einer Menge von Bedingungen Σ enthält eine gerichtete Kante (r_1, r_2) zwischen zwei Bedingungen r_1 und r_2 genau dann, wenn $r_1 < r_2$ gilt. Wir sagen, Σ sei **stratifiziert** genau dann, wenn die Bedingungen jedes Zyklus von G(Σ) schwach azyklisch sind.

Beispiel 3.3.

- Fall: Nicht c-stratifizierte (aber stratifizierte) TGDs
- Chase: terminiert nur für eine bestimmte Folge von Chase-Schritten

Während das Kriterium der Stratifizierung für Beispiel 3.2 die sichere Terminierung des Chase korrekt bestimmt hätte, zeigt sich an folgenden drei TGDs ein wesentliches Problem des Kriteriums:
$r_1 : \text{No}(ma1, mo1, no1) \rightarrow \exists VO1, \ddot{A}Q1 : \text{Mo}(mo1, VO1, \ddot{A}Q1)$
$r_2 : \text{No}(ma1, mo1, no1) \rightarrow \text{Mo}(mo1, mo1, mo1)$
$r_3 : \text{Mo}(mo1, vo1, \ddot{a}q1), \text{Mo}(mo1, mo1, mo1) \rightarrow \exists MA1, NO1 : \text{No}(MA1, vo1, NO1)$.

Für das Testen der Stratifizierung betrachten wir, ob das Feuern einer TGD das Feuern einer zweiten TGD auslöst – und zwar nur unter Berücksichtigung dieser beiden TGDs. TGD1 kann selbstverständlich Tupel erzeugen, die TGD2 nicht benötigt, gleichzeitig kann TGD2 jedoch auch Tupel voraussetzen, die nicht von TGD1 erzeugt wurden. Stratifizierung gilt schließlich für jede Datenbankinstanz, wir können also voraussetzen, dass zusätzliche benötigte Tupel bereits in der Datenbank stehen – nur im Fall von neu erzeugten Nullwerten können wir sicher sein, dass diese nicht in der Ausgangsinstanz vorkamen.

Im vorliegenden Fall löst r_1 r_3 aus, wenn wir annehmen, dass die Datenbank zum Beispiel das Tupel Mo(1419, 1419, 1419) enthält. Dies ist prinzipiell auch möglich, da die erste Position von Mo in r_1 nicht mit einem Nullwert belegt wird (der Nullwert würde erst nach Feuern von r_3 in die Relation Noten gelangen – Stratifizierung erkennt diesen Sachverhalt aber nicht, da nur die Beziehung $r_1 < r_3$ überprüft wird).

Tatsächlich darf das Tupel Mo(1419, 1419, 1419) aber nicht in der Ausgangsdatenbankinstanz vorkommen, da ansonsten r_1 keinen aktiven Trigger besitzt – die existenzquantifizierten Variablen würden auf 1419 abgebildet, statt neue Nullwerte zu generieren. Der Chase-Graph der Stratifizierung enthält also keine Kante zwischen r_1 und r_3 (Abbildung 3.3). Berücksichtigt man r_2, wird jedoch klar, dass Mo(1419, 1419, 1419) zwar nicht in der ursprünglichen Datenbankinstanz stand, jedoch nach Feuern von r_1 durch r_2 in diese eingefügt wurde. Die Reihenfolge (r_2, r_3) der Anwendung von Chase-Schritten terminiert also sicher (r_2 und r_3 sind volle TGDs[2]), während die Abfolge (r_1, r_2, r_3, r_1, r_2, r_3, r_1, ...) nicht terminiert. Durch Anwendung des Naiven Oblivious Chase umgehen wir obiges Problem – wir überprüfen also nicht, ob r_1 tatsächlich einen aktiven Trigger besitzt, da dieser Test nicht auf Basis von zwei TGDs alleine stattfinden kann (Abbildung 3.4). □

Für *c-Stratifizierung* (ursprünglich entwickelt von [MSL09a]) modifizieren wir obige Definition der Präzedenzrelation, indem anstelle des Standard-Chase der Naive Oblivious Chase verwendet wird.

[2] r_3 enthält zwar zwei existenzquantifizierte Variablen, verhält sich hier aber wie eine volle TGD – die Positionen No_1 und No_3 werden für das Beispiel nicht benötigt und enthalten folglich auch in keiner TGD in Körper und Kopf allquantifizierte Variablen

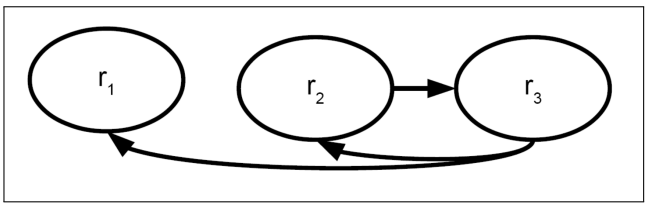


Abbildung 3.3.: Chase-Graph für Beispiel 3.3 unter Verwendung der Stratifizierung. Der Zyklus zwischen r_2 und r_3 wird zwar erkannt, ist jedoch ohne Auswirkungen. Der Zyklus zwischen r_1 und r_3 wird hingegen nicht erkannt.

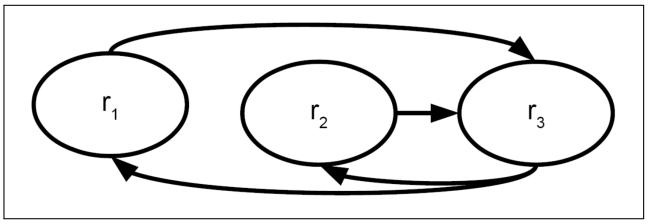


Abbildung 3.4.: Chase-Graph für Beispiel 3.3 unter Verwendung der c-Stratifizierung. Der Zyklus zwischen r_1 und r_3 wird erkannt und führt tatsächlich zum Nicht-Terminieren des Chase.

Definition 3.4. (Präzedenzrelation der c-Stratifizierung, [GMS12])
Es gilt $r_1 <_c r_2$ genau dann, wenn es zwei Datenbankinstanzen K und J sowie die Homomorphismen h_1 und h_2 gibt, sodass gilt:

1. $K \xrightarrow{*,r_1,h_1} J$,
2. $J \not\models h_2(r_2)$,
3. $K \models h_2(r_2)$

Obwohl wir den Naiven Oblivious Chase bei der Durchführung des Terminierungstests verwenden, treffen wir keine Aussagen über die Terminierung dieser Chase-Variante – schließlich überprüfen wir die gefundenen Zyklen mit dem Kriterium der Schwachen Azyklizität, nicht mit dem der Reichen Azyklizität. Beide Formen der Stratifizierung erweitern das Kriterium der Schwachen Azyklizität, indem sie die Menge der überprüften Integritätsbedingungen einschränken. Im Folgenden werden wir eine Möglichkeit kennenlernen, Schwache Azyklizität durch Einschränkung der Menge verwendeter Positionen zu erweitern.

Safety

Die Safety-Bedingung (entwickelt in [MSL09a]) basiert auf der Idee der affected Positionen. Affected Positionen zeichnen sich durch das potentielle Auftreten von Nullwerten aus.

Definition 3.5. (Affectedness, [GMS12])
Eine Position R_i ist **affected**, wenn es eine Bedingung r:$\phi(\mathbf{x}, \mathbf{z}) \rightarrow \exists \mathbf{y} : \psi(\mathbf{x}, \mathbf{y}) \in \Sigma$ gibt und es entweder

1. eine Variable $y \in \mathbf{y}$ gibt, die in Position $R_i \in \psi$ auftritt, oder
2. es eine Variable $x \in \mathbf{x}$ gibt, die sowohl in Position R_i in ψ als auch ausschließlich in affected Positionen in ϕ vorkommt. Die Menge der affected Positionen von Σ wird als aff(Σ) bezeichnet.

Definition 3.6. (Sichere Menge von TGDs, [GMS12])
Sei Σ eine Menge von TGDs, dann bezeichnet prop(Σ) = (aff(Σ), E) den **Propagationsgraphen** von Σ der wie folgt definiert ist. Für jede TGD $\phi(\mathbf{x}, \mathbf{z}) \rightarrow \exists \mathbf{y} \psi(\mathbf{x}, \mathbf{y})$ gilt, wenn eine Variable $x \in \mathbf{x}$ in ϕ in Position R_i vorkommt:

1. wenn x ausschließlich in affected Positionen in ϕ vorkommt, füge E für jedes Vorkommen von x in Position S_j in ψ die Kante $R_i \to S_j$ hinzu,

2. wenn x ausschließlich in affected Positionen in ϕ vorkommt, füge E für jedes $y \in \mathbf{y}$ und für jedes Vorkommen von y in Position S_j von ψ die besondere Kante $R_i \to S_j$ hinzu.

Eine Menge von Bedingungen wird als **sicher** (safe) angesehen, wenn der zugehörige Propagationsgraph keine Zyklen aufweist, die durch besondere Kanten gehen.

Beispiel 3.4.

- Fall: TGDs, die weder schwach azyklisch noch c-stratifiziert, jedoch safe sind
- Chase: terminiert

Wenn der Körper einer Integritätsbedingung mehrere Atome mit gleichem Relationssymbol enthält, reicht das Kriterium der Schwachen Azyklizität häufig nicht aus, um das Terminieren des Chase korrekt vorherzusagen. In einem solchen Fall müsste stattdessen das Safety-Kriterium angewandt werden, wie anhand der drei folgenden TGDs gezeigen werden soll:

$r_1 : \mathrm{No}(ma1, mo1, no1) \to \exists VO1, \ddot{A}Q1 : \mathrm{Mo}(mo1, VO1, \ddot{A}Q1)$
$r_2 : \mathrm{No}(ma1, mo1, no1) \to \mathrm{Mo}(mo1, mo1, mo1)$
$r_3 : \mathrm{Mo}(mo1, vo1, \ddot{a}q1), \mathrm{Mo}(mo1, mo1, mo1) \to \exists MA1, NO1 : \mathrm{No}(MA1, mo1, NO1).$

Der Test auf WA weist einen Zyklus $\mathrm{No}_2 \to \mathrm{Mo}_2 \to \mathrm{No}_2$ nach, der durch eine besondere Kante geht (siehe Abbildung 3.5). Dennoch ist die obige Menge von TGDs „harmlos“ – im Gegensatz zum vorherigen Beispiel wird in r_3 diesmal keine Variable übertragen, die einen Nullwert tragen kann. WA führt in diesem Fall zu einem fehlerhaften Ergebnis, da der Körper von r_3 zwei Positionen Mo_2 aufweist – eine kann einen Nullwert tragen, die andere jedoch nicht, und nur letztere überträgt ihren Wert in den Kopf der TGD. Mo_2 und Mo_3 sind affected durch r_1. Damit dies jedoch zu weiteren Positionen führen kann, die affected sind, muss es Variablen geben, die im Körper einer TGD auf Mo_2 und Mo_3 beschränkt sind. Diese gibt es auch tatsächlich – nämlich $vo1$ und $\ddot{a}q1$ in r_3. Allerdings kommen diese Variablen nicht auf der rechten Seite von r_3 vor. $mo1$ kommt auf der rechten Seite der TGD vor, ist auf der linken Seite der TGD jedoch sowohl in affected als auch in nicht affected Positionen vorhanden. No_2 ist also nicht affected – folglich gibt es keine besondere Kante $\mathrm{No}_2 \to \mathrm{Mo}_2$ (die Kante $\mathrm{Mo}_2 \to \mathrm{No}_2$ durch r_3 bleibt jedoch bestehen, siehe Abbildung 3.6). □

Mit Safety und c-Stratifizierung haben wir zwei Erweiterungen der Schwachen Azyklizität kennengelernt, die jedoch nicht miteinander vergleichbar sind – es gibt also für beide Kriterien TGD-Mengen, für welche die Chase-Terminierung ausschließlich von diesem Kriterium erkannt wird. Im Folgenden wollen wir eine Terminierungsklasse betrachten, welche eine Obermenge der zuvor definierten Terminierungsklassen (außer Stratifizierung) darstellt.

Safe Restriction

Ursprünglich vorgestellt in [MSL09b], verbindet das Kriterium der Safe Restriction Konzepte der c-Stratifizierung und der Safety.

Definition 3.7. (Safe Restriction, nach [GMS12])
Für ein Menge von Positionen P und die TGD r bezeichnet $\mathrm{aff}(r, P)$ die Menge der Positionen P_i aus dem Kopf von r, sodass

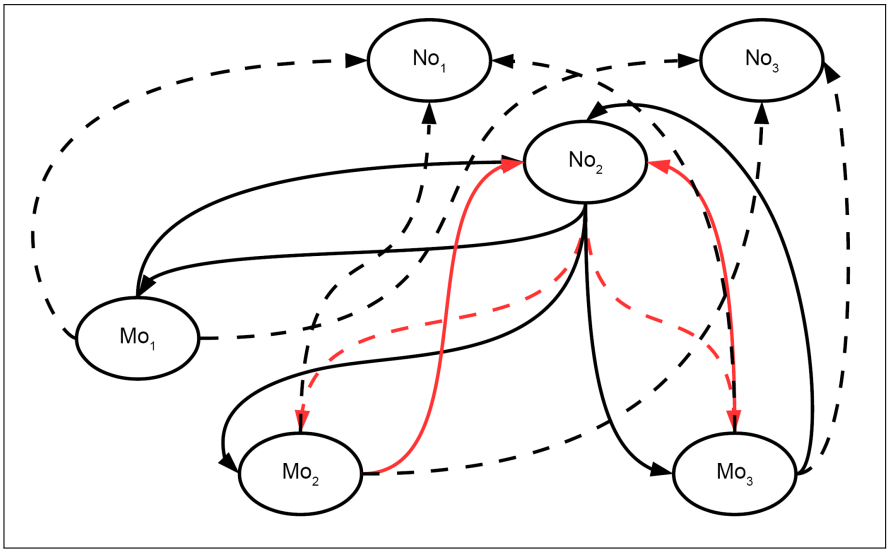


Abbildung 3.5.: Abhängigkeitsgraph für Beispiel 3.4. Die Zyklen zwischen No_2 und Mo_2 sowie zwischen No_2 und Mo_3 (rot markiert) führen jeweils durch eine besondere Kante (der gestrichelte Pfeil von No_2 nach Mo_2 und der gestrichelte Pfeil von No_2 nach Mo_3), die TGDs sind also nicht schwach azyklisch und terminieren möglicherweise nicht.

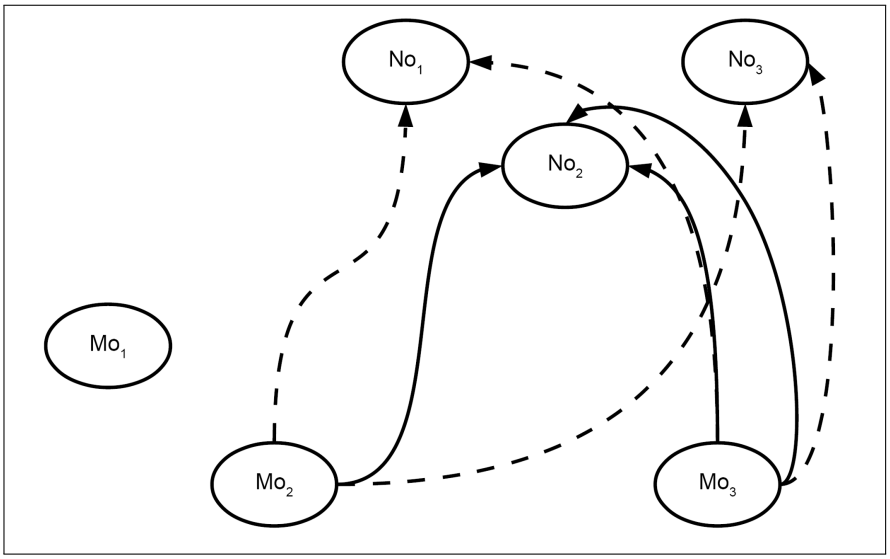


Abbildung 3.6.: Propagationsgraph für Beispiel 3.4. Da nur Mo_2 und Mo_3 affected sind, wurden aus dem Abhängigkeitsgraph in Abb. 3.5 alle Kanten entfernt, die nicht von Mo_2 oder Mo_3 ausgehen. Folglich sind keine Zyklen im Graph mehr vorhanden und die TGDs sind safe.

1. jede allquantifizierte Variable x aus P_i, die im Körper von r ausschließlich in Positionen aus P vorkommt, oder
2. P_i mindestens eine existenzquantifizierte Variable enthält.

Für jedes Paar von TGDs r_1, r_2 aus Σ und $P \subseteq pos(\Sigma), r_1 <_P r_2$ genau dann, wenn:

1. $r_1 <_c r_2$ (siehe Definition c-Stratifizierung),
2. es einen Nullwert gibt, der vom Körper zum Kopf von $h_2(r_2)$ so weitergegeben wird, dass er in K ausschließlich in Positionen von P erscheint (wobei h_2 und K wie in der Definition der c-Stratifizierung zu verstehen sind).

Ein **2-Restriktionssystem** ist ein Paar $(\mathrm{G}'(\Sigma), P)$, wobei $\mathrm{G}'(\Sigma) = (\Sigma, E)$ einen gerichteten Graph darstellt und $P \subseteq \mathrm{pos}(\Sigma)$, wobei gilt:

1. $\forall (r_1, r_2) \in E$: wenn r_1 eine TGD ist, dann ist $\mathrm{aff}(r_1, P) \cap \mathrm{pos}(\Sigma) \subseteq P$, gleichwohl gilt, dass wenn r_2 eine TGD ist, $\mathrm{aff}(r_2, P) \cap \mathrm{pos}(\Sigma) \subseteq P$ gilt, und
2. $r_1 <_P r_2 \Rightarrow (r_1, r_2) \in E$.

Σ ist **safely restricted** genau dann, wenn es ein Restriktionssystem $(\mathrm{G}'(\Sigma), P)$ für Σ gibt, sodass jede stark verbundene Komponente aus $\mathrm{G}'(\Sigma)$ safe ist. Ein 2-Restriktionssystem ist minimal, wenn es aus $((\Sigma, \emptyset), \emptyset)$ durch wiederholte Anwendung von Regel 1 und 2 entstanden ist (bis beide Regeln bezüglich aller Bedingungen gelten) und P nur um die Positionen erweitert wurde, die für die Befriedigung von Regel 1 benötigt werden. Σ ist safely restricted genau dann, wenn jede stark verbundene Komponente des Restriktionssystem $\mathrm{G}'(\Sigma)$ safe ist, wobei $(G'(\Sigma), P)$ das minimale 2-Restriktionssystem von Σ darstellt.

Diese Definition lässt also durchaus zu, dass Σ nicht nur TGDs (sondern z.B. auch EGDs) enthält, über die Zugehörigkeit von (r_1, r_2) zu E wird in diesem Fall (abgesehen von der Forderung der Minimalität) keine Aussage getroffen.

Beispiel 3.5.

- Fall: TGDs, die weder safe, noch c-stratifiziert, jedoch safely restricted sind
- Chase: terminiert

Indem das Kriterium der Safe Restriction Ideen der c-Stratifizierung und der Safety kombiniert, erkennt es das Terminieren des Chase auch in Situationen, in denen keines der beiden Kriterien für sich alleine eine korrekte Vorhersage geliefert hätte, wie die folgenden beiden TGDs zeigen werden:
$r_1 : \mathrm{No}(ma1, mo1, no1), \mathrm{Mo}(mo1, vo1, \ddot{a}q1) \rightarrow \exists VO2 : \mathrm{Mo}(\ddot{a}q1, VO2, mo1)$
$r_2 : \mathrm{No}(ma1, mo1, no1), \mathrm{Mo}(mo1, vo1, \ddot{a}q1) \rightarrow \exists VO2, MO2 : \mathrm{Mo}(\ddot{a}q1, VO2, MO2), \mathrm{Mo}(MO2, VO3, mo1)$.

Offensichtlich hält der Chase auf den obigen TGDs an. Beide TGDs benötigen ein Tupel der Noten-Tabelle, welches die selbe Modulnummer enthält, wie das ebenfalls geforderte Tupel der Modul-Tabelle. Da keine der beiden TGDs einen Eintrag in die Noten-Tabelle macht, kann die so geforderte Modulnummer also kein Nullwert sein. TGD r_2 führt jedoch Nullwerte sowohl für das Äquivalenzmodul, als auch für die Modulnummer ein, darüber hinaus überführen beide TGDs den Wert des Äquivalenzmoduls in das Attribut des Moduls – sobald hierdurch die Modulnummer jedes erzeugten Tupels mit einem neuen Nullwert belegt wird, hält der Chase also an.

Da Position No_2 nicht affected ist, führt das Einfügen von Nullwerten in Position Mo_1 nicht zu weiteren Tupeln mit Nullwerten. Für $P = \{\mathrm{Mo}_1, \mathrm{Mo}_3\}$ gilt $r_1 \not<_P r_1, r_1 \not<_P r_2, r_2 <_P r_1$ und $r_2 \not<_P r_2$ (siehe Abbildung 3.7). Daher ist $\mathrm{G}(\Sigma) = (\{r_1, r_2\}, \{(r_2, r_1)\})$. $\{r_1, r_2\}$ ist folglich safely restricted. □

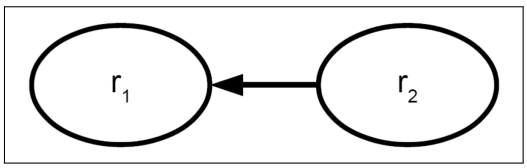


Abbildung 3.7.: Graph für Beispiel 3.5. Da keine Zyklen nachweisbar sind, existieren auch keine stark verbundenen Komponenten. Die TGDs sind also ohne weitere Tests auf Sicherheit safely restricted und die Terminierung des Chase ist garantiert. Inductive Restriction.

Induktive Restriktion

Induktive Restriktion (IR, entwickelt in [MSL09a]) verfeinert Safe Restriction, indem Bedingungen in einer geschickteren Weise partitioniert werden. Insbesondere wird zuerst das System $(\mathrm{G}'(\Sigma), P)$ und die Aufteilung in $\Sigma_1, .., \Sigma_n$ berechnet, wobei jedes Σ_i eine Menge von Abhängigkeiten darstellt, die eine stark verknüpfte Komponente in $\mathrm{G}'(\Sigma)$ darstellen. Anschließend wird – wenn $n = 1$ – das Safety-Kriterium verwendet, ansonsten wird das IR-Kriterium induktiv für jedes i angewandt.

Beispiel 3.6.

- Fall: TGDs, die weder safely restricted, noch safe, noch c-stratifiziert, jedoch inductive restricted sind
- Chase: terminiert

Betrachten wir folgende drei TGDs:
$r_1 : \mathrm{No}(ma1, mo1, no1), \mathrm{Mo}(mo1, vo1, \ddot{a}q1) \rightarrow \exists VO2 : \mathrm{Mo}(\ddot{a}q1, VO2, mo1)$
$r_2 : \mathrm{No}(ma1, mo1, no1), \mathrm{Mo}(mo1, vo1, \ddot{a}q1) \rightarrow \exists VO2, MO2, VO3 : \mathrm{Mo}(\ddot{a}q1, VO2, MO2),$
$\mathrm{Mo}(MO2, VO3, mo1)$
$r_3 : \rightarrow \exists MA1, MO1, NO1, VO1, \ddot{A}Q1 : \mathrm{No}(MA1, MO1, NO1), \mathrm{Mo}(MO1, VO1, \ddot{A}Q1).$

Indem wir der im vorherigen Beispiel betrachteten TGD-Menge die Bedingung r_3 hinzufügen, ändert sich offenbar nichts an der Terminierung des Chase (r_3 kann höchstens einmal angewandt werden, da alle Variablen im Kopf der TGD existenzquantifiziert sind). Das Safe Restricted Kriterium ist jedoch nicht mehr ausreichend, die Terminierung der TGDs zu erkennen. Die TGDs sind hingegen inductive restricted, da bei Aufteilung der Bedingungen in stark verbundene Komponenten die beiden Komponenten $\{r_3\}$ und $\{r_1, r_2\}$ erkannt werden, die beide safely restricted sind. Der Nachweis für $\{r_3\}$ ist trivial (da r_3 keinen Körper besitzt) und die Safe Restriction von $\{r_1, r_2\}$ ist bereits in Beispiel 2 erfolgt. □

Die Kriterien c-Stratifizierung, Safe Restriction und Induktive Restriktion partitionieren die untersuchte TGD-Menge in mehrere Untermengen, die anschließend entweder mit dem Kriterium der Schwachen Azyklizität oder mit dem Safety-Kriterium untersucht werden. Alle drei Kriterien hängen also letztendlich vom Konzept der Position ab. Das Kriterium der Superschwachen Azyklizität weicht von dieser Herangehensweise ab – an die Stelle der abstrakten Position tritt ein konkreter *Platz* in einem Atom. Auf diese Weise kann die Terminierung des Chase auf TGD-Mengen detektiert werden, für die dies mit den zuvor vorgestellten Kriterien nicht möglich ist.

Superschwache Azyklizität

Das Kriterium der Superschwachen Azyklizität (eingeführt in [Mar09]) konstruiert einen Triggergraphen $\Gamma(\Sigma) = (\Sigma, E)$, dessen Kanten die Beziehung zwischen Bedingungen beschreiben. Eine Kante $r_i \rightsquigarrow r_j$ zeigt an, dass ein Nullwert, der durch eine Bedingung r_i erzeugt wurde, direkt oder indirekt in den Körper von r_j weitergegeben wird.

Definition 3.8. (Superschwache Azyklizität, nach [GMS12])
Sei Σ eine Menge von TGDs und sei sk(Σ) die skolemisierte Form von Σ, d.h. jede existenzquantifizierte Variable y aus dem Kopf einer TGD wird durch die Skolemfunktion $f_y^r(x)$ ersetzt, wobei x eine Menge von Variablen ist, die sowohl im Körper, als auch im Kopf von r vorkommen. Ein Platz ist ein Paar (a, i), wobei a ein Atom von sk(Σ) darstellt und $0 \leq i \leq$ Stelligkeit(a) gilt. Gegeben sei eine TGD r und eine existenzquantifizierte Variable y im Kopf von r. Out(r, y) steht für die Menge der Plätze (Output-Plätze) im Kopf von sk(r), in denen ein Term der Form $f_y^r(x)$ vorkommt. Sei r eine TGD und sei x eine allquantifizierte Variable in r. In(r, x) steht für die Menge der Plätze (Input-Plätze) im Körper von r, in denen x vorkommt. Gegeben sei eine Menge von Variablen V. Eine **Substitution** θ von V ist eine Funktion, die jede Variable v aus V auf einem endlichen Term $\theta(v)$ abbildet, der aus Konstanten oder Funktionssymbolen besteht. Zwei Plätze (a, i) und (a', i) sind **unifizierbar** – dargestellt als $(a, i) \sim (a', i)$ – genau dann, wenn es zwei Substitutionen θ und θ' gibt, sodass $\theta(a) = \theta'(a')$ gilt. Gegeben seien zwei Mengen von Plätzen Q und Q'. Es gilt $Q \sqsubseteq Q'$ genau dann, wenn für alle q aus Q ein q' aus Q' existiert, für das gilt: $q \sim q'$.
Für jede Menge von Plätzen Q steht Move(Σ, Q) für die kleinste Menge von Plätzen Q', für die gilt

1. $Q \subseteq Q'$,
2. $\forall B_r \to H_r \in \text{sk}(\Sigma), \forall x : \Pi_x(B_r) \sqsubseteq Q' \Rightarrow \Pi_x(H_r) \subseteq Q'$, wobei $\Pi_x(B_r)$ und $\Pi_x(H_r)$ für die Mengen der Plätze in B_r und H_r stehen, in denen x vorkommt.

Gegeben sei eine Menge TGDs Σ und zwei TGDs r_1, r_2 aus Σ. r_1 **triggert** r_2 in Σ ($r_1 \rightsquigarrow r_2$) genau dann, wenn eine existenzquantifizierte Variable y im Kopf von r_1 existiert und eine allquantifizierte Variable x in Körper und Kopf von r_2 vorkommt, sodass In$(r_2, x) \sqsubseteq$ Move$(\Sigma,$ Out$(r_1, y))$. Eine Menge von Bedingungen ist **superschwach azyklisch** genau dann, wenn der Graph $\Gamma(\Sigma) = (\Sigma, \{(r_1, r_2) | r_1 \rightsquigarrow r_2\})$ – der **Triggergraph** – azyklisch ist.

Beispiel 3.7.

- Fall: TGDs, die nicht safe, jedoch c-stratifiziert und superschwach azyklisch sind
- Chase: terminiert

Um das Kriterium der Superschwachen Azyklizität zu veranschaulichen, betrachten wir folgende zwei TGDs. Die Plätze der TGDs sind in der Zeile unter der jeweiligen TGD gekennzeichnet.

$$r_1 : \text{No}(ma1, mo1, no1) \to \exists MO2, \ddot{A}Q1 : \text{Mo}(MO2, mo1, \ddot{A}Q1)$$
$$\quad\; p_1 \quad p_2 \quad p_3 \qquad\qquad\qquad\qquad\qquad\qquad p_4 \quad p_5 \quad p_6$$
$$r_2 : \text{Mo}(mo1, vo1, mo1) \to \exists MA1, NO1 : \text{No}(MA1, mo1, NO1)$$
$$\quad\; p_7 \quad p_8 \quad p_9 \qquad\qquad\qquad\qquad\qquad\qquad p_{10} \quad p_{11} \quad p_{12}$$

Da Move$(\Sigma,$ Out$(r_1, MO2) = \{p_4\}$, Move$(\Sigma,$ Out$(r_1, \ddot{A}Q1) = \{p_6\}$, und In$(r_2, mo1)) = \{p_7, p_9\}$ gilt, ist In$(r_2, mo1) \not\sqsubseteq$ Move$(\Sigma, Out(r_1, MO2))$, und In$(r_2, mo1) \not\sqsubseteq$ Move$(\Sigma, Out(r_1, \ddot{A}Q1))$. Im vorliegenden Beispiel muss noch getestet werden, ob r_1 durch r_2 getriggert wird, allerdings sind die existenzquantifizierten Variablen $MA1$ und $NO1$ im Kopf von r_2 ohne Konsequenz für die Durchführung des Chase: Move$(\Sigma,$ Out$(r_2, MA1) = \{p_{10}\}$, Out$(r_2, NO1) = \{p_{12}\}$, In$(r_1, mo1) = \{p_2\}$, daher ist In$(r_1, mo1) \not\sqsubseteq$ Move$(\Sigma,$ Out$(r_2, MA1)$ und In$(r_1, mo1) \not\sqsubseteq$ Move$(\Sigma,$ Out$(r_2, NO1)$. Folglich triggert r_1 nicht r_2, und r_2 triggert auch nicht r_1 (außerdem triggert r_1 sich auch nicht selbst), weshalb Σ superschwach azyklisch ist (vergleiche Abbildung 3.8). □

Superschwache Azyklizität kann zwar in einigen Fällen die Terminierung des Chase bestimmen, in denen dies mit dem Kriterium der Induktiven Restriktion nicht möglich ist, ist jedoch im Allgemeinen nicht

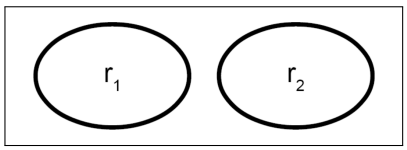


Abbildung 3.8.: Triggergraphen für Beispiel 3.7. Der Graph besitzt keine Kanten, die TGDs sind also superschwach azyklisch und der Chase terminiert garantiert.

vergleichbar mit diesem. Im Folgenden wollen wir das Kriterium der Lokalen Stratifizierung betrachten, dass mächtiger als alle zuvor genannten Terminierungskriterien (außer einfacher Stratifizierung) ist.

Lokale Stratifizierung

Das Kriterium der Lokalen Stratifizierung (vorgestellt in [GST11]) basiert auf der Idee des feuerbaren Platzes. Es sind zunächst also nicht etwa TGDs, die sich gegenseitig triggern, sondern einzelne Plätze in Kopf und Körper der TGDs. Im Gegensatz zum namensgebenden Kriterium der Stratifizierung wird das Feuern der Plätze jedoch nicht durch Kanten eines Graphen dargestellt. Stattdessen dient das Verhalten der Plätze als Grundlage eines Graphen zwischen den TGDs (vergleichbar der Superschwachen Azyklizität). Bei der Definition der Lokalen Stratifizierung verwendete Begriffe (z.B. Triggergraph) und Symbole (z.B. $<$) sind uns häufig bereits bei der Definition anderer Terminierungskriterien mit unterschiedlicher Bedeutung begegnet.

Definition 3.9. (MOVE, nach [GMS12])
Ein Platz q im Körper einer Bedingung r kann durch einen Platz q' im Kopf der Bedingung r' gefeuert werden ($q' < q$), wenn $q \sim q'$ und $r' < r$ gelten. Gegeben seien zwei Mengen von Plätzen, Q und Q'. Q' kann Q genau dann **feuern** ($Q' < Q$), wenn für alle $q \in Q$ gilt, dass ein Platz $q' \in Q'$ existiert, sodass $q' < q$ gilt. Gegeben sei eine Menge von Plätzen Q aus einer Menge von Bedingungen Σ. Wir definieren $\text{MOVE}(\Sigma, Q)$ als die kleinste Menge an Plätzen Q', sodass gilt:

1. $Q \subseteq Q'$,
2. $\forall B_r \rightarrow H_r \in \text{sk}(\Sigma), \forall x : Q' < \Pi_x(B_r) \Rightarrow \Pi_x(H_r) \subseteq Q'$, wobei $\Pi_x(B_r)$ und $\Pi_x(H_r)$ die Menge der Plätze in B_r und H_r darstellt, in denen x vorkommt, und $\text{sk}(\Sigma)$ die skolemisierte Form der Bedingungen aus Σ ist (d.h. alle existenzquantifizierten Variablen wurden durch Skolemfunktionen ersetzt).

Definition 3.10. (Lokale Stratifizierung, [GMS12])
Gegeben sei eine Menge von TGDs Σ und zwei TGDs r_1, r_2 aus Σ. Wir sagen genau dann, dass r_1 r_2 in Σ **auslöst** ($r_1 \hookrightarrow r_2$), wenn es eine existenzquantifizierte Variable y im Kopf von r_1 und eine allquantifizierte Variable x sowohl im Körper und Kopf von r_2 gibt, sodass gilt:

$$\text{MOVE}(\Sigma, \text{Out}(r_1, y)) < \text{In}(r_2, x).$$

Eine Menge von Bedingungen ist **lokal stratifiziert** (LS) genau dann, wenn der Triggergraph $\Delta(\Sigma) = \{(r_1, r_2) | r_1 \hookrightarrow r_2\}$ azyklisch ist.

Beispiel 3.8.

- Fall: TGDs, die nicht inductive restricted, jedoch lokal stratifiziert sind
- Chase: terminiert

Betrachten wir folgende drei TGDs:

$$r_1 : \mathrm{No}(ma, mo, no) \rightarrow \exists VO1, \ddot{A}Q1, MA, MO1, MO3, MO4 : \mathrm{Mo}(mo, VO1, \ddot{A}Q1),$$
$$\mathrm{Ko}(MA, MO1, \ddot{A}Q1, MO3, MO4)$$

$$r_2 : \mathrm{Mo}(mo, vo, \ddot{a}q), \mathrm{Ko}(ma, mo, \ddot{a}q, mo3, mo4) \rightarrow \exists MA, NO : \mathrm{No}(MA, \ddot{a}q, NO)$$

$$r_3 : \mathrm{Mo}(mo, vo, \ddot{a}q) \rightarrow \exists VO2 : \mathrm{Mo}(\ddot{a}q, VO2, mo)$$

Offensichtlich terminiert der Chase auf den obigen TGDs. Die TGD r_2 verlangt, dass zwei Tupel aus der Komplexprüfungstabelle und der Modultabelle in zwei Attributen übereinstimmen (d.h., dass eine Person sowohl in einem Modul, als auch in einem Modul äquivalent zum ersten Modul geprüft wurde), Einträge in die Komplexprüfungstabelle erfolgen allerdings allein durch r_1 – und hier wird das erste geprüfte Modul mit einem einzigartigen Nullwert belegt, der weder hier, noch durch eine andere TGD in die Modultabelle übertragen wird.

Betrachten wir zunächst die Superschwache Azyklizität. Hier gilt: $\mathrm{Move}(\Sigma, Out(r_1, \ddot{A}Q1)) = \{p_6, p_9, p_{21}, p_{26}, p_4, p_{28}\}$ und $\mathrm{In}(r_1, mo) = \{p_2\} \sqsubseteq \mathrm{Move}(\Sigma, Out(r_1, \ddot{A}Q1))$. Da $r_1 \rightsquigarrow r_1$ gilt, ist der Triggergraph zyklisch, daher ist Σ nicht superschwach azyklisch (siehe Abbildung 3.9). Σ ist darüber hinaus nicht IR, da $r_1 <_c r_3 <_c r_2 <_c r_1$ (siehe Abbildung 3.10) und für jedes Paar von TGDs r_i, r_j, für die $r_i <_c r_j$ gilt – also $(r_1, r_3), (r_3, r_2), (r_2, r_1)$ – ist es möglich, eine Datenbankinstanz mit Nullwerten in den Positionen $\mathrm{Mo}_1, \mathrm{Mo}_3, \mathrm{No}_2$ und Ko_3 zu konstruieren, sodass jedes Mal ein Nullwert vom Kopf von r_i in den Kopf von r_j übertragen wird, wenn r_j durch r_i gefeuert wird. Im Gegensatz hierzu gilt $r_1 \not\rightsquigarrow r_1$ (da $\mathrm{MOVE}(\Sigma, \mathrm{Out}(r_1, \ddot{A}Q)) = \{p_6, p_9, p_{26}\}$ und $\mathrm{In}(r_1, mo) = \{p_2\} \not\sqsubseteq \mathrm{MOVE}(\Sigma, \mathrm{Out}(r_1, \ddot{A}Q))$), $\Delta(\Sigma)$ ist also azyklisch und daher ist Σ lokal stratifiziert. □

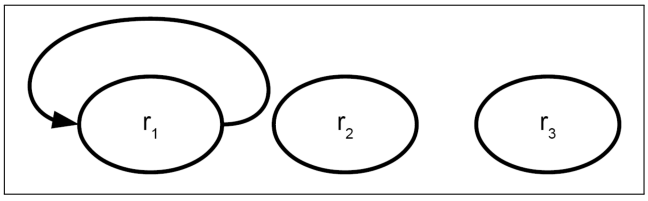


Abbildung 3.9.: Triggergraph für Beispiel 3.8. Superschwache Azyklizität ist nicht gewährleistet, da r_1 sich selbst triggert.

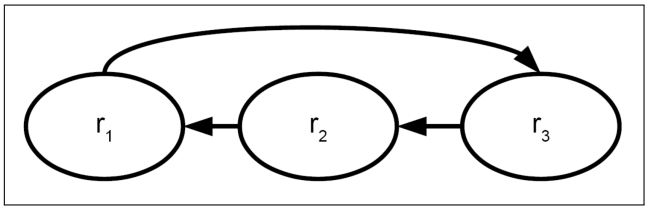


Abbildung 3.10.: Chase-Graph für Beispiel 3.8 unter Verwendung der c-Stratifizierung. Die paarweise Analyse der sich im dargestellten Zyklus aufrufenden TGDs zeigt, dass die TGDs nicht inductive restricted sind.

Während Lokale Stratifizierung ein in sich abgeschlossenes Terminierungskriterium darstellt, ist Stratifizierung nur in Kombination mit Schwacher Azyklizität anwendbar. Im Folgenden werden wir einen Terminierungstest vorstellen, der ursprünglich dafür vorgesehen war, mit anderen Kriterien kombiniert zu werden – es hat sich jedoch herausgestellt, dass dies nicht nötig ist. Constraint-Rewriting führt, ob in Kombination mit einem anderen Kriterium oder alleine angewandt, zur Terminierungsklasse der *Azyklizität*, welche eine Obermenge der Lokalen Stratifizierung darstellt.

Constraint-Rewriting

Constraint-Rewriting [GST11] kann genutzt werden, um eine Menge von TGDs in eine äquivalente Menge von Bedingungen mit gleichen Terminierungseigenschaften zu transformieren. Für diese transformierten TGDs kann die Terminierung des Chase auch dann gezeigt werden, wenn dies für die ursprüngliche Menge von Bedingungen nicht der Fall war. Ein wesentliches Konzept dieses Algorithmus ist das *Adornment*. Die in der vorliegenden Arbeit genannten Varianten des Constraint-Rewritings – Adn, Adn+ und Adn++ – erhielten ihren Namen von diesem Konzept. Wenn wir im Folgenden von Constraint-Rewriting sprechen, beziehen wir uns stets auf den Adornment++-Algorithmus (Adn++, Algorithmus 4). Ein Adornment ist ein String, der dem Relationsnamen eines Prädikats hinzugefügt wird. Das Adornment zeigt (ähnlich dem Safety-Kriterium) an, ob die zugehörige Stelle des Prädikats einen Nullwert trägt. Im Gegensatz zum Safety-Kriterium sind es also nicht die Positionen, die derart markiert werden, sondern die Plätze der einzelnen Atome. Constraint-Rewriting erzeugt darüber hinaus nicht nur eine Menge von TGDs mit dem „endgültigen“ Adornment (entsprechend der Festlegung der Safety von Positionen), sondern generiert Kopien der ursprünglichen TGDs, die alle möglichen Adornments tragen. Zu Beginn des Algorithmus erzeugen wir eine Kopie der ursprünglichen TGDs, deren Körper-Prädikate mit einem „b“ für jede Stelle des Prädikats adornt sind. Anschließend führen wir wiederholtes Kopfadornment durch.

Definition 3.11. (Kopfadornment, [GMS12])
Sei $R^{\alpha_1 \cdots \alpha_n}(x_1^{\beta_1}, ..., x_n^{\beta_n})$ ein adorntes Prädikat, wobei α_i ein Adornment des Atoms und β_i das Adornment eines Terms ist. Wenn für alle $i \in \{1, ..., \text{Stelligkeit}(R)\}$ gilt: $\alpha_i = \beta_i$, so ist das Adornment von R **konsistent**[3] . Wenn alle Atome einer Konjunktion relationaler Atome ϕ konsistent adornt sind, so ist ϕ konsistent adornt. Gegeben sei eine TGD $r : \phi(\mathbf{x}, \mathbf{z}) \rightarrow \exists \mathbf{y} : \psi(\mathbf{x}, \mathbf{y})$. Sei $\phi^{\alpha}(\mathbf{x}, \mathbf{z})$ ein konsistentes Adornment der Körperatome. SkHeadAdn$(r, \phi^{\alpha}(\mathbf{x}, \mathbf{z}))$ kennzeichnet dann den adornten Kopf von r, der wie folgt erzeugt wird:

1. jede allquantifizierte Variable hat das selbe Adornment wie im Körper,
2. existenzquantifizierte Variablen in TGDs mit leerem Körper und Konstanten sind mit b adornt,
3. jede existenzquantifizierte Variable $y \in \mathbf{y}$ wird mit f_i adornt,[4] wobei i eine natürliche Zahl ist, die auf die Skolemfunktion $\mathrm{f}_y^r(\alpha[\mathbf{x}])$ verweist, (hier steht $\alpha[\mathbf{x}]$ für den Teilstring von α , der auf $\mathbf{x}$ verweist).

Beispiel 3.9. Einfluss der Skolemisierung
Anhand folgender TGD soll gezeigt werden, welche Auswirkung die Skolemisierung der f-Adornments auf die Terminierung des Adn++-Algorithmus haben kann. Die TGD
$r_1 : \mathrm{R}(x, z) \rightarrow \exists y : \mathrm{R}(x, y)$

[3] Adornments wurden zwar zuvor für Atome definiert, man kann aber auch ein Adornment für Terme definieren, wobei ein Term das Adornment der entsprechenden Stelle des Körperatoms übernimmt. Beim Kopfadornment wird reziprok dem Kopfatom das Adornment der zugehörigen Variable zugewiesen. Inkonsistenz bedeutet für den Constraint-Rewriting-Algorithmus, dass derselben Variable im Körper einer TGD unterschiedliche Adornments zugewiesen wurden.

[4] Im Folgenden werden Adornments kursiv gesetzt, wenn eine allgemeine Gruppe von Adornments gemeint ist, z.B. der Adornment-Typ b-Adornment oder die Adornment-Variable f_i. Für letztere wird darüber hinaus Index i tiefgestellt, obwohl der Index konkreter f-Adornments aus Gründen besserer Lesbarkeit nicht tiefgestellt wird.

führt zu $\text{SkHeadAdn}(r, \text{R}^{\text{bb}}(x,z)) = \exists y : \text{R}^{\text{bf1}}(x,y)$, wobei $\text{f1} = \text{f}_y^{r_1}(b)$ gilt, anschließend erhalten wir $\text{SkHeadAdn}(r_1, \text{R}^{\text{bf1}}(x,z)) = \exists y : \text{R}^{\text{bf1}}(x,y)$. Der Adn++-Algorithmus bricht an dieser Stelle ab, da kein neues Adornement f2 generiert werden kann (welches zum Prädikat $\exists y : \text{R}^{\text{bf2}}(x,y)$ führen würde), da die Skolemfunktion von $\text{f2} = \text{f}_y^{r_1}(b)$ wäre (d.h. f1 und f2 hängen vom selben Teilstring des Adornments, der die allquantifizierten Variablen repräsentiert, in der selben TGD ab und beziehen sich auf die selbe existenzquantifizierte Variable, folglich gilt f1=f2). □

Definition 3.12. (Adornment-Substitution, [GMS12])
Durch **Adornment-Substitution** wird das Terminieren des Adornment-Algorithmus garantiert. Eine Substitution θ ist eine Menge von Paaren f_i/f_j mit $i \neq j$. Das gleiche Symbol kann nicht sowohl auf der rechten als auch der linken Seite der Substitution verwendet werden (d.h. die Menge der substituierten und die Menge der substituierenden Symbole sind disjunkt und ein Symbol f_j, das ein Symbol f_i substituiert, kann in θ nicht selbst durch ein Symbol f_k substituiert werden).

Der Algorithmus baut einen Graph E auf, welcher Abhängigkeiten zwischen adornten Prädikaten repräsentiert. Eine Kante (p_r^α, q_s^β) dieses Graphen zeigt an, dass ein Prädikat p^α aus dem Kopf einer adornten Bedingung, die von der TGD r abgeleitet ist, zur Generierung des adornten Prädikats q^β beigetragen hat, welches im Kopf einer adornten Bedingung steht, die von der TGD s abgeleitet wurde. Zweck dieses Graphen ist, zyklische Abhängigkeiten zwischen Prädikaten zu erkennen. Wenn eine neue adornte Bedingung r^α erzeugt wurde und es bereits eine Bedingung r^β gibt, für die eine Substitution θ mit $r^\alpha\theta = r^\beta$ gebildet werden kann, so wird r^α durch $\text{Körper}(r^\alpha) \rightarrow \text{Kopf}(r^\beta)$ ersetzt, um die Erzeugung einer unendlichen Menge von adornten Bedingungen zu verhindern.

Obwohl der Adornment++-Algorithmus (ebenso wie Adn+ und der ursprüngliche Algorithmus Adn, [SG10]) zunächst dazu diente, aus einer Menge von TGDs eine andere Menge von TGDs zu gewinnen, deren Terminierung mit Kriterien bestimmbar ist, welche die Terminierung der ursprünglichen Menge TGDs nicht feststellen können, stellt der Algorithmus auch für sich bereits ein Terminierungskriterium bereit – und zwar das allgemeinste der bisher vorgestellten Kriterien, die **Azyklizität**. Der Adornment++-Algorithmus gibt neben den adornten TGDs den booleschen Wert CYC (für Cyclicity) zurück. Dieser zeigt an, ob im Graphen E ein Zyklus gefunden wurde. Wenn CYC negativ ist, ist die Terminierung des Chase garantiert, ansonsten hält der Chase möglicherweise nicht an (das Problem der Unentscheidbarkeit der Chase-Terminierung wird selbstverständlich auch durch dieses Kriterium nicht gelöst). Auf den ersten Blick kann der Eindruck entstehen, dass eine erfolgreiche Substitution gleichbedeutend mit einem gefundenen Zyklus ist (und dies ist auch in allen in dieser Arbeit vorgestellten Beispielen der Fall), tatsächlich könnten zwei substituierbare adornte Versionen derselben TGD aber auch auf andere Weise entstanden sein.

Beispiel 3.10.

- Fall: Durchführung des Adornment++-Algorithmus auf einer Menge azyklischer TGDs
- Chase: terminiert

Das Terminieren des Chase auf den folgenden TGDs ist durch keines der bisher genannten Kriterien bis auf den soeben vorgestellten Constraint-Rewriting-Algorithmus nachweisbar. Tatsächlich ist der Algorithmus recht aufwendig, weshalb einige eigentlich notwendige Tests nur auszugsweise dargestellt sind (siehe Anmerkungen).

Ursprüngliche TGDs:
$r_1 : \text{No}(ma1, mo1, no1) \rightarrow \exists \ddot{A}Q : \text{Mo}(mo1, mo1, \ddot{A}Q)$
$r_2 : \text{Mo}(mo1, vo1, \ddot{a}q1) \rightarrow \exists MA, MO2, MO3, MO4 : \text{Ko}(MA, vo1, MO2, MO3, MO4)$
$r_3 : \text{Ko}(ma1, mo1, mo2, mo3, mo4), \text{Ko}(ma2, mo1, mo5, mo6, mo7),$
$\text{Ko}(ma3, mo5, mo1, mo8, mo9) \rightarrow \exists MA, NO : \text{No}(MA, mo2, NO)$

Adornte TGDs:
$r_{1a} : \text{No}^{\text{bbb}}(ma1, mo1, no1) \rightarrow \exists \ddot{A}Q : \text{Mo}^{\text{bbf1}}(mo1, mo1, \ddot{A}Q)$
$r_{2a} : \text{Mo}^{\text{bbb}}(mo1, vo1, \ddot{a}q1) \rightarrow \exists MA, MO2, MO3, MO4 : \text{Ko}^{\text{f2bf3f4f5}}(MA, vo1, MO2, MO3, MO4)$
$r_{3a} : \text{Ko}^{\text{bbbbb}}(ma1, mo1, mo2, mo3, mo4), \text{Ko}^{\text{bbbbb}}(ma2, mo1, mo5, mo6, mo7),$
$\text{Ko}^{\text{bbbbb}}(ma3, mo5, mo1, mo8, mo9) \rightarrow \exists MA, NO : \text{No}^{\text{f6bf7}}(MA, mo2, NO)$

Anmerkung: Die Tabellen der Studentendatenbank enthalten deutlich mehr Attribute, als für das Beispiel nötig wären, weshalb auch Nullwerte (bzw. f-Adornments) entstehen, die keinen Beitrag zum Algorithmus leisten – die in r_3 vorhandenen existenzquantifizierten Variablen dienen allein dem Auffüllen der Relation, sie werden von r_1 nicht verwendet und erzeugen folglich keine neuen Prädikate:
$\text{No}^{\text{f7bf7}}(ma1, mo1, no1) \rightarrow \exists \ddot{A}Q : \text{Mo}^{\text{bbf1}}(mo1, mo1, \ddot{A}Q) \Rightarrow$ kein neues Prädikat

Setzen wir nun das mit r_{2a} generierte Prädikat (bzw. das Adornment dieses Prädikats) in r_{3a} ein, um r_{3b} zu erzeugen:
$r_{3b} : \text{Ko}^{\text{f2bf3f4f5}}(ma1, mo1, mo2, mo3, mo4), \text{Ko}^{\text{bbbbb}}(ma2, mo1, mo5, mo6, mo7),$
$\text{Ko}^{\text{bbbbb}}(ma3, mo5, mo1, mo8, mo9) \rightarrow \exists MA, NO : \text{No}^{\text{f8f3f9}}(MA, mo2, NO).$

Anmerkung:
$\text{Ko}^{\text{f2bf3f4f5}}(ma1, mo1, mo2, mo3, mo4), \text{Ko}^{\text{f2bf3f4f5}}(ma2, mo1, mo5, mo6, mo7),$
$\text{Ko}^{\text{f2bf3f4f5}}(ma3, mo5, mo1, mo8, mo9) \rightarrow \exists MA, NO : \text{No}^{\text{f8f2f9}}(MA, mo2, NO)$ ist nicht konsistent (u.a. $mo1$ ist mit unterschiedlichen Adornments belegt), konsistent wäre hingegen:
$\text{Ko}^{\text{f2bf3f4f5}}(ma1, mo1, mo2, mo3, mo4), \text{Ko}^{\text{f2bf3f4f5}}(ma2, mo1, mo5, mo6, mo7),$
$\text{Ko}^{\text{f2f3bf4f5}}(ma3, mo5, mo1, mo8, mo9) \rightarrow \exists MA, NO : \text{No}^{\text{f8f2f9}}(MA, mo2, NO),$
allerdings benötigt dies ein Prädikat – $\text{Ko}^{\text{f2f3bf4f5}}(ma3, mo5, mo1, mo8, mo9)$ – das bisher nicht erzeugt wurde.

Setzen wir nun das durch r_{3b} generierte adornte Prädikat in r_{1a} ein, erhalten wir r_{1b} und, nach Einsetzen des auf diese Weise erzeugten adornten Prädikats in r_{2a}, r_{2b}:
$r_{1b} : \text{No}^{\text{f8f2f9}}(ma1, mo1, no1) \rightarrow \exists \ddot{A}Q : \text{Mo}^{\text{f3f3f10}}(mo1, mo1, \ddot{A}Q)$
$r_{2b} : \text{Mo}^{\text{f2f2f10}}(mo1, vo1, \ddot{a}q1) \rightarrow \exists MA, MO2, MO3, MO4 : \text{Ko}^{\text{f11f3f12f13f14}}(MA, vo1, MO2, MO3, MO4).$
Der Algorithmus bricht ab, ohne einen Zyklus gefunden zu haben (vergleiche Abbildung 3.11), da nur inkonsistente neue TGDs erzeugt werden können, z.B. die folgende:
$\text{Ko}^{\text{f11f3f12f13f14}}(ma1, mo1, mo2, mo3, mo4), \text{Ko}^{\text{f11f3f12f13f14}}(ma2, mo1, mo5, mo6, mo7),$
$\text{Ko}^{\text{f11f3f12f13f14}}(ma3, mo5, mo1, mo8, mo9) \rightarrow \exists MA, NO : mathrmNo^{f15f12f16}(MA, mo2, NO).$ □

Bisher haben wir Terminierungstests betrachtet, die unabhängig von der jeweiligen Datenbankinstanz angewandt werden können. Für viele Anwendungsfälle ist es jedoch ausreichend, wenn der Chase auf einer bestimmten (möglicherweise sogar leeren) Instanz terminiert. Im Folgenden werden wir beispielhaft einen Terminierungstest vorstellen, der dies berücksichtigt.

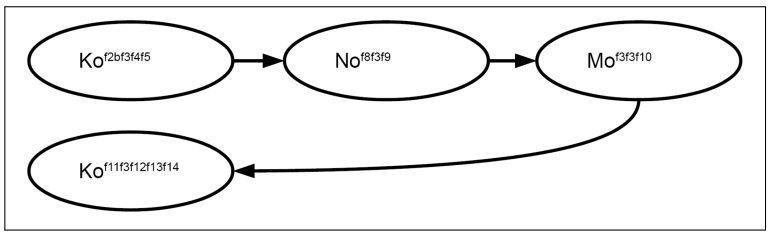


Abbildung 3.11.: Beziehung zwischen den in Beispiel 3.10 durch den Adornment++-Algorithmus generierten Prädikaten. Um Platz zu sparen, wurde auf die Darstellung der Variablen verzichtet. Kein Zyklus ist erkennbar, die TGDs sind also azyklisch und die Terminierung des Chase ist garantiert.

3.1.2. Semi-dynamische Kriterien

[MSL09a] unterscheidet zwischen einem statischen Ansatz (in der vorliegenden Arbeit durch die statischen Kriterien implementiert) und einem dynamischen Ansatz, die Chase-Terminierung zu überprüfen. Für letzteren wird nicht die Terminierung auf einer beliebigen, sondern nur auf einer bestimmten Datenbankinstanz untersucht. Hierfür wird während der Durchführung des Chase auf dieser Instanz die Einführung von Nullwerten in einem „Monitor Graph“ festgehalten und die mögliche Nicht-Terminierung anhand der Eigenschaften dieses Graphen ermittelt. Semi-dynamische Kriterien ähneln diesem Ansatz, da auch hier nur die Terminierung auf einer bestimmten Instanz untersucht und möglicherweise der Chase selbst genutzt wird, um die Terminierung des Chase festzustellen (allerdings wird letzteres an keiner Stelle ausdrücklich festgelegt, [GHK+13]).

Die Bezeichnung „semi-dynamische Kriterien“ stammt nicht von den Entwicklern der Kriterien selbst, sondern aus [CGMT16], dessen Autoren die von ihnen konzipierten Constraint-Rewriting-Algorithmen sowohl von den statischen, als auch von den semi-dynamischen Kriterien abgrenzen. Da der in der vorliegenden Arbeit beschriebene Adn++-Algorithmus über das eigentliche Constraint-Rewriting – also die Erhöhung der Mächtigkeit statischer Kriterien durch Modifikation der Abhängigkeitsbasis – hinaus geht und bereits für sich die Terminierung des Chase voraussagen kann, wurde Constraint-Rewriting hier als statisches Kriterium aufgefasst. Tatsächlich weist es jedoch starke Ähnlichkeiten mit semi-dynamischen Kriterien auf, die auf die sogenannte *kritische Instanz* angewandt werden. Man könnte nämlich Adornments auch als Werte interpretieren, die durch den Skolem-Oblivious Chase in Relationen eingefügt werden (jedes erzeugte adornte Prädikat würde hier einem Tupel der jeweiligen Relation entsprechen). b-Adornments stehen hierbei für die Einträge der kritischen Instanz (Adn++ berücksichtigt – anders als der in diesem Unterabschnitt beschriebene MSA-Algorithmus – Konstanten im Körper der TGDs nicht als gesonderte Einträge).

Es sind zwei semi-dynamische Kriterien bekannt, *Model-Faithful Acyclicity* (MFA) und *Model-Summarising Acyclicity* (MSA). Da letzterer eine Erweiterung des ersteren darstellt, wird nachfolgend nur MSA beschrieben.

Definition 3.13. (Model-Summarising Acyclicity, [GHK+13])
Für jede Abhängigkeit $r : \phi(\mathbf{x}, \mathbf{z}) \rightarrow \exists \mathbf{y} : \psi(\mathbf{x}, \mathbf{y})$ und jede Variable $y_i \in \mathbf{y}$, sei F_r^i ein neues, einstelliges Prädikat (eindeutig identifizierbar durch i und r). Seien darüber hinaus S und D neue binäre Prädikate[5], und sei C ein neues nullstelliges Prädikat.

Darüber hinaus gilt für jede Abhängigkeit $r : \phi(\mathbf{x}, \mathbf{z}) \rightarrow \exists \mathbf{y} : \psi(\mathbf{x}, \mathbf{y})$ und für jede Variable $y_i \in \mathbf{y}$: Sei c_r^i ein neues Konstantensymbol, das eindeutig durch r und y_i identifizierbar ist. Wir bezeichnen mit $\mathbf{MSA}(r)$ die folgende Abhängigkeit, wobei θ_{MSA} für eine Substitution steht, die jede Variable $y_i \in \mathbf{y}$

[5]Die beiden binären Prädikate entsprechen im Wesentlichen den besonderen Kanten der Schwachen Azyklizität.

durch c_r^i ersetzt:

$$\phi(\mathbf{x},\mathbf{z}) \rightarrow \psi(\mathbf{x},\mathbf{y})\theta_{MSA} \wedge \bigwedge_{y_i \in \mathbf{y}} \left(\mathrm{F}_r^i(y_i)\theta_{MSA} \wedge \bigwedge_{x_j \in \mathbf{x}} \mathrm{S}(x_j, y_i)\theta_{MSA} \right).$$

Für eine Menge von Abhängigkeiten Σ bezeichnet **MSA**(Σ) die kleinste Menge, die MSA(r) für jede Abhängigkeit r aus Σ, und darüber hinaus die folgenden TGDs r_1, r_2 und r_3 enthält, wobei r_3 für jedes Prädikat F_r^i ($r \in \Sigma$) instanziiert wird.

- $r_1 : \mathrm{S}(x_1, x_2) \rightarrow \mathrm{D}(x_1, x_2)$
- $r_2 : \mathrm{D}(x_1, x_2), \mathrm{S}(x_2, x_3) \rightarrow \mathrm{D}(x_1, x_3)$
- $r_3 : \mathrm{F}_r^i(x_1), D(x_1, x_2), \mathrm{F}_r^i(x_2) \rightarrow \mathrm{C}$

Die Menge Σ ist **model-summarising acyclic** (MSA) bezüglich einer Instanz I wenn $I \cup \mathrm{MSA}(\Sigma) \not\models \mathrm{C}$ gilt[6]. Darüber hinaus ist Σ **allgemein MSA**[7], wenn Σ MSA bezüglich I_Σ^* (der kritischen Instanz) ist.

Die **kritische Instanz** I_Σ^* für eine Menge von Abhängigkeiten Σ enthält alle Tupel, die aus der neuen Konstante $*$ und den Konstanten erzeugt werden können, die in den Körpern der Abhängigkeiten aus Σ vorkommen. Der Skolem-Oblivious Chase terminiert auf I_Σ^* genau dann, wenn der Skolem-Oblivious Chase mit Σ auf jeder Instanz terminiert.

Im vorangegangenen Unterabschnitt haben wir gezeigt, wie die Terminierung des Chase auf bestimmten Datenbankinstanzen untersucht werden kann. Hierbei sind wir, wie auch bei der Definition aller anderen Terminierungskriterien, nur von TGDs ausgegangen. Wichtige Integritätsbedingungen – zum Beispiel Primärschlüsselbeziehungen – lassen sich jedoch nicht als TGD darstellen, sondern als EGD. Im Folgenden demonstrieren wir eine Möglichkeit, EGDs in einen Terminierungstest zu integrieren.

3.1.3. Einbeziehung von EGDs

Bei der Formulierung der Terminierungskriterien wurde bisher nur auf TGDs geachtet. Tatsächlich ist der Einbezug von EGDs nicht für alle Kriterien ohne weiteres möglich. Manche der oben genannten Kriterien (insbesondere die Schwache Azyklizität) sind so streng, dass EGDs ignoriert werden können. Für andere Kriterien (insbesondere Adornment++) gibt es eine spezielle Variante, die EGDs einschließt (aber nicht notwendigerweise die selbe Mächtigkeit besitzt wie der ursprüngliche Algorithmus ohne Einbezug von EGDs, [CGMT16]). Letztendlich kann man jedoch für alle Kriterien existierende EGDs durch TGDs simulieren. Im folgenden Algorithmus werden Werte, die durch EGDs gleichgesetzt wurden, in einer neu definierten zweiattributigen Tabelle *Eq* gespeichert, wobei Gleichheit als Äquivalenzrelation verstanden wird (tatsächlich sind z.B. Nullwerte und Konstanten für EGDs aber nicht gleichwertig). Wenn immer eine TGD gleiche Werte voraussetzt, soll sie auch ungleiche Werte akzeptieren, solange diese in der Tabelle Eq als gleich definiert wurden.

Der Algorithmus beschäftigt sich also mit dem Fall, dass eine TGD gleiche Werte erwartet, eine andere TGD jedoch an der entsprechenden Stelle unterschiedliche Werte erzeugt (was zum Terminieren des Chase führen würde), eine EGD die ungleichen Werte aber gleichsetzt (was zum Nicht-Terminieren des Chase führt).

[6] Wie eingangs erwähnt legt die Definition nicht fest, dass der (Skolem-Oblivious) Chase für diese Ableitung verwendet werden soll. Bei der Implementierung mittels des Chase müssten selbstverständlich leichte Änderungen vorgenommen werden, da die Darstellung eines nullstelligen Prädikats als relationale Tabelle wenig sinnvoll ist.

[7] Wenn Σ allgemein MSA ist, terminiert der Chase sicher auf jeder Instanz – tatsächlich bezeichnet allgemein MSA also ein statisches Kriterium (siehe Unterabschnitt 3.1.1)

Ein anderer Fall, dass nämlich eine eingebettete TGD einen Nullwert erzeugt (der zum Nicht-Terminieren des Chase führt), eine EGD diesen Nullwert jedoch mit einer Konstante gleichsetzt und somit entfernt (was zu einem Terminieren des Chase führt), wird hingegen nicht betrachtet. Dies entspricht jedoch den Einschränkungen, die wir ohnehin für Terminierungskriterien haben – wenn die Bedingungen bestimmte Eigenschaften aufweisen, terminiert der Chase garantiert, ansonsten terminiert er möglicherweise jedoch ebenfalls.

Substitutionslose Simulation besteht aus zwei Schritten: EqAx und Singularisierung. Im ersten Schritt wird sichergestellt, dass die neue Relation Eq eine Äquivalenzrelation repräsentiert, also symmetrisch, transitiv und reflexiv ist. Wenn Terme im Kopf einer Integritäsbedingung (d.h. einer EGD) gleichgesetzt werden, so führt dies zu einem entsprechenden Eintrag in Eq (durch $\psi\downarrow_{Eq}$) und der Umwandlung der EGD in eine TGD. Im zweiten Schritt werden Gleichsetzungen im Körper von Integritätsbedingungen behandelt. Hierbei ist zwischen der Gleichsetzung von Variablen und der Gleichsetzung von Variablen mit Konstanten zu unterscheiden.

Definition 3.14. (EqAx[S], [Mar09])
Gegeben sei eine neue Relation **Eq**. Sei ψ eine Formel oder Datenbank, so bezeichnen wir mit $\psi\downarrow_{Eq}$ die Formel oder Datenbank, die durch Ersetzen jedes Atoms $(x = y)$ aus ψ durch $\mathrm{Eq}(x, y)$ entsteht. Gegeben sei ein Schema S. Wir definieren die Menge EqAx[S] als die Menge von Bedingungen, welche die folgenden TGDs enthält:

- $\mathrm{Eq}(x, y) \rightarrow \mathrm{Eq}(y, x)$,
- $\mathrm{Eq}(x, y), \mathrm{Eq}(y, z) \rightarrow \mathrm{Eq}(x, z)$,
- $\forall \mathrm{R} \in S$ der Stelligkeit n: $\mathrm{R}(x_1, ..., x_n) \rightarrow \mathrm{Eq}(x_1, x1), \ldots, \mathrm{Eq}(x_n, x_n)$.

Definition 3.15. (Singularisierung, [Mar09])
Gegeben sei eine TGD oder EGD r. Die **Singularisierung** $\mathrm{Sgr}(r)$ von r ist die Formel, die wie folgt aus r gewonnen wird:

1. für jedes Vorkommen einer Konstante c in einem Atom $\mathrm{R}(x, c, y)$ im Körper von r, erzeuge eine neue Variable z_c und ersetze $\mathrm{R}(x, c, y)$ durch $\mathrm{R}(x, z_c, y), z_c = c$,[8]
2. für alle Variablen x, die n-mal im Körper von r vorkommen (wobei $n \geq 2$ gilt), erzeuge $n - 1$ neue Variablen $x_2, \ldots, x_n$, und ergänze den Körper von r mit $x = x_2, x_2 = x_3, \ldots, x_{n-1} = x_n$ und ersetze $n - 1$ Vorkommen von x durch paarweise unterschiedliche Variablen aus $\{x_2, \ldots, x_n\}$.

Definition 3.16. (Substitutionslose Simulation, [Mar09])
Für alle Tupel oder Gleichheit erzeugende Schemaabbildungen $M = (S, T, \Sigma)$ definieren wir eine tupelerzeugende Schemaabbildung $\mathrm{Sim}^e(M) = (S', T', \mathrm{Sim}^e(\Sigma))$, wobei $S' = S \cup \{\mathrm{Eq}\}, T' = T \cup \{\mathrm{Eq}\}$ und $\mathrm{Sim}^e(\Sigma) = \{\mathrm{Sgr}(r) | r \in \Sigma\}\downarrow_{Eq} \cup \mathrm{EqAx}[S \cup T]$ gilt.

Substitutionslose Simulation (in der vorliegenden Arbeit häufig auch als EGD-Rewriting bezeichnet) wandelt also nicht nur EGDs direkt in TGDs um, sondern erzeugt darüber hinaus neue TGDs und ergänzt bereits vorhandene TGDs um zusätzliche Atome. Die Komplexität der zu untersuchenden Integritätsbedingungen wird unter Umständen also deutlich erhöht – und zwar selbst dann, wenn ausschließlich TGDs betrachtet werden. Im Folgenden untersuchen wir einige Chase-Werkzeuge und die von ihnen durchgeführten Terminierungstests. Substitutionslose Simulation wird jedoch (bisher) von keinem der Werkzeuge angewandt.

[8] In diesem Zwischenschritt haben wir tatsächlich Gleichheitsatome im Körper einer TGD oder EGD (die letztendlich selbstverständlich durch das Atom $\mathrm{Eq}(z_c, c)$ ersetzt werden).

3.2. Umgesetzte Chase-Werkzeuge

Obwohl es bislang kein Programm gibt, das sämtliche Anwendungsvarianten des Chase umsetzt, bildet der Algorithmus die Grundlage mehrerer frei verfügbarer Softwaretools. Im Folgenden wollen wir einige davon kurz vorstellen. Neben dem an der Universität Rostock entstehenden ChaTEAU sind für uns selbstverständlich besonders jene Programme von besonderem Interesse, die einen Chase-Terminierungstest implementiert haben. Das Terminierungswerkzeugs für ChaTEAU sollte in einer Weise konzipiert werden, die nicht einfach eine der hier vorgestellten Herangehensweisen kopiert, sondern über diese hinaus geht. Die vorgestellten Werkzeuge wurden (bis auf ChaseTEQ) bereits zuvor im Rahmen studentischer Arbeiten untersucht und (bis auf PDQ) in einer virtuellen Maschine installiert. ChaseTEQ wurde dieser virtuellen Maschine nachträglich hinzugefügt. Um zu überprüfen, wie die verschiedenen Chase-Tools mit TGDs umgehen, für die der Chase nicht terminiert, wurde Anwendungsfall 2 (siehe Kapitel 5.1) in die jeweiligen Programme importiert.

3.2.1. ChaTEAU

ChaTEAU entstand 2018 im Zuge der Masterarbeit von Martin Jurklies [Jur18]. Im Folgejahr wurden durch Fabian Renn grundsätzliche Erweiterungen der Ein- und Ausgabe vorgenommen und der Chase auf Anfragen implementiert [Ren19]. Die Weiterentwicklung der Kernmodule von ChaTEAU ist noch nicht abgeschlossen, so wird etwa parallel zur vorliegenden Arbeit die Einbeziehung von st-TGDs durch Jakob Zimmer realisiert.

Zentrales Anliegen der ursprünglichen Autoren war die Implementierung eines Werkzeugs, das mehrere Anwendungsvarianten des Chase in sich vereint. Varianten bezieht sich hier auf die Variabilität der Parameter (z.B. TGDs oder EGDs) und Objekte (z.B. Datenbankinstanzen oder Anfragen), auf die der Chase angewandt werden kann, nicht aber auf algorithmische Varianten wie z.B. den Naiven Oblivious Chase – in dieser Hinsicht ist ChaTEAU auf eine Chase-Variante, nämlich den Standard-Chase, beschränkt. Bei der Implementierung wurde Wert darauf gelegt, theoretische Konzepte möglichst unverfälscht umzusetzen, anstatt (wie z.B. Llunatic) neue Strategien zu entwickeln, welche die praktische Anwendung des Chase effektiver gestalten. In erster Linie handelt es sich bei ChaTEAU also um eine Plattform, auf der wissenschaftliche Konzepte (wie in der vorliegenden Arbeit die Chase-Terminierung) getestet werden können.

Die Interaktion des Nutzers mit ChaTEAU erfolgt über Dateien und die Konsole, eine graphische Benutzeroberfläche ist (bis auf einen Dateiauswahl-Dialog) nicht vorhanden. Datenbankinstanzen, Integritätsbedingungen und Anfragen werden im XML-Format definiert. Hierbei diente PDQ als Vorbild. Obwohl im Gegensatz zu Llunatic vorrangig XML-Tags und XML-Attribute genutzt werden, um TGDs, Atome und Variablen zu definieren, sind zusätzliche Informationen im Namen der Terme codiert, so beginnt der Name existenzquantifizierter Variablen beispielsweise stets mit der Zeichenfolge „#E_“. Ergebnisinstanzen werden erstens als XML-Datei, zweitens als csv-Datei und drittens textuell über die Konsole ausgegeben. Über das Log kann der Nutzer einzelne Chase-Schritte nachvollziehen.

Erst im Rahmen der vorliegenden Rahmen wird ChaTEAU über die Möglichkeit erweitert, vor Durchführung des Chase dessen Terminierung zu überprüfen. Tatsächlich wird ein nicht terminierender Chase für Chateau aber auch erst durch Erweiterungen ermöglicht, die parallel zur vorliegenden Arbeit vorgenommen werden.

3.2.2. PDQ

Literaturreview PDQ (Proof Driven Query Answering) wurde 2014 erstmals in einem Konferenzbeitrag vorgestellt [BLT14]. Konkreter Anwendungsfall von PDQ ist die Beantwortung von Anfragen an entfernte Web-basierte Datenquellen, zum Beispiel über ein Webformular oder einen Webservice. Die hierfür verwendeten Anfragepläne generiert PDQ, indem es beweist, dass die Anfrage beantwortbar ist. Aus jedem Beweis lässt sich ein Anfrageplan ableiten. Gleichzeitig werden die Kosten des Plans auf Basis einer nutzerdefinierten Kostenfunktion bestimmt.

PDQ nimmt im Wesentlichen zwei Eingaben an: Zum einen Eingabeschemata (einschließlich Angaben über Zugriffskosten) und zu erfüllende Integritätsbedingungen, zum anderen Anfragen (mit Select-, Project- oder Join-Operationen). Sowohl Schema als auch Anfragen müssen als XML-Dateien vorliegen. Die hierbei verwendete Beschreibung durch XML-Tags ist hierarchisch strukturiert und diente z.B. ChaTEAU als Vorbild. PDQ verfügt über eine graphische Benutzerschnittstelle, über welche die genannten XML-Dateien geladen werden können. Darüber hinaus ermöglicht es diese Benutzeroberfläche, den Suchraum des Planers in Echtzeit als Graph zu visualisieren. PDQ kann jedoch auch aus der Kommandozeile heraus gestartet werden.

PDQ akzeptiert als Eingabe zwar nur Integritätsbedingungen, für die der Chase terminiert (und zusätzlich TGDs der Form $\forall \vec{x} : \mathrm{G}(\vec{x}) \rightarrow \exists \vec{y} : \bigwedge_i \mathrm{H}_i(\vec{x}, \vec{y})$, wobei G und jedes H_i Atome sind), aber ein expliziter Terminierungstest scheint nicht implementiert zu sein [BLT14].

Praktischer Test Das Schema der Studenten- und Fächer-Relation wurde in Form einer XML-Datei definiert. Die beiden TGDs des Anwendungsfalles wurden in derselben Datei charakterisiert. Die bereits für ChaTEAU erstellte XML-Datei konnte an dieser Stelle aufgrund mangelnder Kompatibilität nicht verwendet werden. PDQ benötigt zusätzliche Daten. Da diese Informationen (z.B. Zugriffskosten) für den untersuchten Anwendungsfall unwesentlich sind, wurde hier auf Standardwerte aus studentischen Vorarbeiten zurückgegriffen. Aus den gleichen Arbeiten wurde eine Konfigurationsdatei (`case.properties`) übernommen. In einer dritten Datei wurde eine Anfrage an die Fächerrelation, ebenfalls im XML-Format, definiert. PDQ erlaubt zwar komplexe Anfragen, die in SQL-Anfragen transformiert werden können, wir begnügten uns jedoch mit einer Projektion auf die ersten beiden Attribute der Fächer-Relation (wie in der Untersuchung von Graal). Da wir keine Datenbankinstanz festlegen konnten (und keinen Webservice definieren wollten), war es nicht möglich, die Ergebnisse der Anfrage auszuwerten (im Gegensatz zu Graal). Stattdessen generiert PDQ – wenn der Chase terminiert – einen Anfrageplan.

Wie aus dem Literaturreview zu erwarten war, erkennt PDQ nicht im Voraus, dass der Chase nicht terminiert. Die graphische Benutzeroberfläche von PDQ bietet zwar Möglichkeiten, eine maximale Schrittanzahl sowie eine maximale Zeitdauer zu definieren, diese Optionen führen jedoch nicht zu einem vorzeitigen Abbruch der Bearbeitung. Da die Analyse des Programmes nicht endet (Abbildung 3.12), generiert PDQ keinen Anfrageplan.

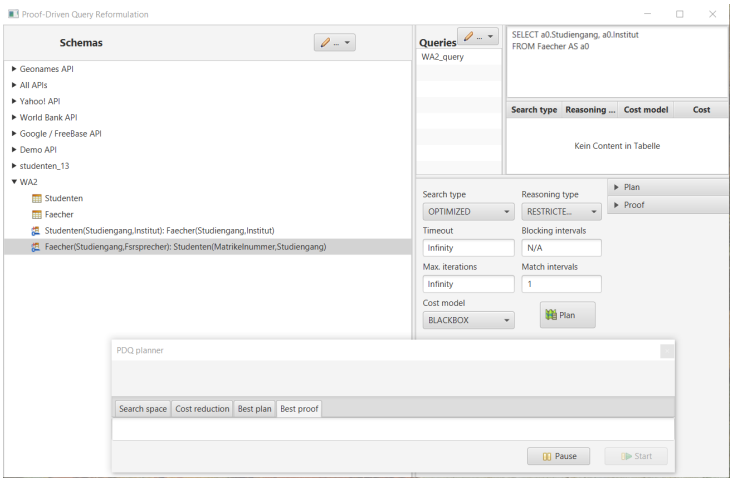


Abbildung 3.12.: Screenshot der Testanwendung von PDQ. Wie erwartet liefert das Programm kein Ergebnis für die gegebene Anfrage.

3.2.3. Llunatic

Literaturreview Llunatic wurde erstmals 2013 von [GMPS13] in einem Konferenzbeitrag vorgestellt. Im Rahmen der Doktorarbeit von Donatello Santoro wurde das Programm 2014 erweitert [San14]. Der Code des Open Source-Programmes ist frei zugänglich, aufgrund der fehlenden Kommentierung jedoch schwer verständlich.

Grundlegendes Ziel der Autoren war es, Schemaabbildungen (zum Zweck von Datenbanktransformationen und Datenaustausch) und Datenreparatur (inklusive Data Cleaning) in einer einheitlichen Weise durchzuführen [GMPS14]. Dementsprechend ist es mit Llunatic möglich, mehrere Quelldatenbanken (inklusive möglicher Masterdaten) auf eine (nicht notwendigerweise leere) Zieldatenbank abzubilden und Reparaturen der Daten vorzunehmen. Für letzteres kann auf die Masterdaten zurück gegriffen und außerdem verschiedene Arten von Abhängigkeiten (u.a. FDs, conditional FDs, Editing Rules, IDs, conditional IDs) berücksichtigt werden. Ein wesentliches Anliegen der Autoren war die Skalierbarkeit der Anwendung, da der Chase in ungünstigen Fällen exponentielle Laufzeit benötigt. Als Konsequenz dieser Zielvorgabe wurde eine parallele Variante des Chase implementiert.

Datenbankinstanzen und Integritätsbedingungen werden im XML-Format übergeben. Im Gegensatz zu PDQ oder dem hiervon beeinflussten ChaTEAU ist die Struktur der einzelnen Tupel oder Bedingungen jedoch nicht auf diese Weise charakterisiert. Es wird also z.B. nicht für jedes Atom einer Bedingung ein eigenes XML-Tag verwendet, sondern alle Bedingungen in einem einzigen CDATA-Tag definiert. Darüber hinaus erfolgt die Kontrolle von Llunatic über eine graphische Benutzeroberfläche, mit der Anwender Szenarien spezifizieren und Konfigurationen des Reparaturprozesses vornehmen können.

Obwohl die Untersuchung von Terminierungskriterien keinen Schwerpunkt von Llunatic darstellt und in den meisten Veröffentlichungen der Autoren nicht erwähnt wird, überprüft Llunatic vor Durchführung des Chase die übergebenen Integritätsbedingungen auf Schwache Azyklizität. Weitere „sophisticated termination conditions“ (mit Verweis auf [GST11]) werden in [San14] zwar als mögliche Erweiterungen erwähnt, jedoch nicht im Zusammenhang mit einer konkreten Implementierung in Llunatic. Interessanterweise kommen Begriffe wie „affected“ oder „stratum“ wiederholt im Quelltext von Llunatic vor, ohne dass jedoch ein Zusammenhang mit den Terminierungskriterien „Safety“ oder „Stratification“ bestünde (was aufgrund fehlender Kommentierung des Quelltextes jedoch leicht Verwirrung stiften kann).

Der Test der schwachen Azyklizität findet in der Klasse `CheckWeaklyAcyclicityInTGDs` (im Package `it.unibas.lunatic.model.dependency.operators`) durch die Methode `check()` statt. Die Methode ist vom Typ `void`, bei Identifizieren fehlender schwacher Azyklizität wird daher entweder eine Exception geworfen oder ein ein Eintrag im Log gemacht. Ob eine Exception geworfen wird, hängt vom Status des booleschen Attributs `stopOnNotWA` des `LunaticConfiguration`-Objekts (aus dem Modul `it.unibas.lunatic`) ab, welches beiden Methoden übergeben wird. Der Vorgabewert dieses Attributes ist `true`, wenn keine weiteren Anpassungen vorgenommen werden, verhindert also fehlende schwache Azyklizität die Durchführung des Chase mit Llunatic.

Praktischer Test Datenbankschema, Datenbankinstanz und die beiden TGDs (unter der Bezeichnung ExtTGDs) wurden als Llunatic-Szenario definiert. Nach dem Import der entsprechenden XML-Datei wird der Chase durch Betätigen der „Run“-Schaltfläche gestartet. Entsprechend unserer Erwartungen (und den Ergebnissen studentischer Vorarbeiten) gibt Llunatic die Fehlermeldung „TGDs are not weakly acyclic. Termination is not guarantee“ [sic] aus (Abbildung 3.13). Llunatic testet die TGDs also auf Schwache Azyklizität und führt den Chase gegebenenfalls nicht aus. Tatsächlich zeigt jedoch erst

Llunatic 2.0 dieses Verhalten, während Llunatic 1.0.2 keinen Test auf Schwache Azyklizität durchführt und stattdessen die Fehlermeldung „Reached iteration limit 10 with no solution...“ ausgibt und den Chase abbricht. Letztlich erkennt also auch diese Version von Llunatic das Nicht-Terminieren des Chase.

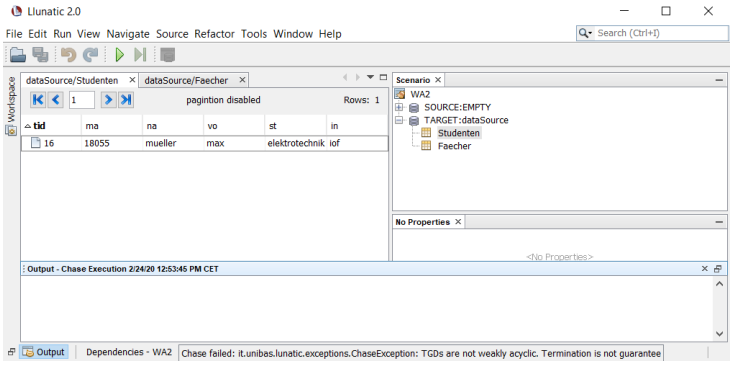


Abbildung 3.13.: Screenshot der Testanwendung von Llunatic. Llunatic erkennt im Vorfeld, dass der Chase nicht terminieren würde, und bricht mit einer Fehlermeldung (ganz unten im Bild) ab.

3.2.4. ChaseTEQ

Literaturreview ChaseTEQ wurde 2011 im Rahmen der Doktorarbeit von Francesca Spezzano [Spe11] entwickelt, das Modul des Programmes, welches den Chase-Terminierungstest durchführt, wurde darüber hinaus im selben Jahr auch einzeln vorgestellt [FGST11].

Das System stellt einen Prototyp dar, welches Anfragen an und Reparatur inkonsistenter oder unvollständiger Datenbanken erlaubt. Den drei Hauptfunktionen – Design von Abhängigkeiten, Reparatur von Datenbanken und Anfragen an Datenbanken – wurde jeweils ein eigenes Modul des Programmes gewidmet (ChaseT, ChaseE und ChaseQ).

ChaseE wendet bei der Reparatur der Datenbank den Standard-Chase, den Skolem-Oblivious Chase oder den Naiven Oblivious Chase an. Sowohl die Auswahl der Chase-Variante, als auch die Definition der anzuwendenden Integritätsbedingungen erfolgt jedoch durch die Eingabemaske von ChaseT.

TGDs werden textuell als Implikationen dargestellt (z.B. `R(x,y),S(y)->T(x,z)`), wobei jede einzelne TGD in einer eigenen Zeile stehen muss. TGDs können direkt über die Eingabemaske von ChaseT eingegeben werden, es können aber auch Textdateien mit zuvor definierten TGDs eingelesen werden. EGDs können ebenfalls definiert werden (haben aber keinen Einfluss auf das Ergebnis des Terminierungstests).

ChaseE und ChaseQ erlauben die Anbindung einer (mySQL-)Datenbank, Anfragen an diese Datenbank (durch ChaseQ) erfolgen dementsprechend unter Verwendung einer (eingeschränkten) SQL-Syntax. ChaseQ erlaubt hierbei z.B. auch den Größenvergleich von Attributen, was in ChaTEAU gegenwärtig noch nicht möglich ist.

Die durch ChaseT durchgeführten Terminierungstests schließen die Kriterien Schwache Azyklizität, Safety, Superschwache Azyklizität und c-Stratifizierung ein. Der Anwender kann Kriterien auswählen, an denen er interessiert ist. Trifft er keine derartige Auswahl, werden alle Kriterien angewandt. Die Terminierungskriterien können mit einem von zwei Constraint-Rewritingverfahren – Adn und Adn+[9] – kombiniert werden. Eine alleinige Durchführung von Constraint-Rewriting zur Ermittlung der Azyklizität ist nicht

[9] Der in der vorliegenden Arbeit beschriebene Algorithmus ist hingegen Adn++ – allerdings scheint der Unterschied zwischen Adn+ und Adn++ nicht wesentlich zu sein.

möglich. Wurde Constraint-Rewriting durchgeführt, so werden neben dem Ergebnis des Terminierungstests auch die generierten adornten TGDs angezeigt. Neben der Durchführung eines bestimmten Terminierungstests können die mit dem jeweiligen Kriterium verbundenen Graphen (z.B. der Triggergraph für das Kriterium der Superschwachen Azyklizität) visualisiert werden.

Praktischer Test Die beiden TGDs des Anwendungsbeispiels wurden als prädikatenlogische Implikation notiert und in Form einer `.txt`-Datei abgespeichert. Nach dem Import dieser Datei wurden zunächst Schwache Azyklizität und Safety der Input-Dependencies überprüft. ChaseTEQ bietet weitere statische Terminierungstests an, aus Vergleichsgründen führten wir jedoch nur jene Testverfahren durch, die wir auch in den Terminierungstester von ChaTEAU implementieren wollen. ChaseTEQ erkennt korrekterweise, dass die untersuchten TGDs weder schwach azyklisch, noch safe sind (Abbildung 3.14 a). Wenn wir jedoch nun das Adornmentverfahren Adn+ auswählen und die Schaltfläche für „Adorning“ betätigen, wird das gewünschte Ergebnis nicht in absehbarer Zeit generiert. Das in der vorliegenden Arbeit konzipierte Werkzeug benötigt für die gleiche Aufgabe nur wenige Millisekunden (Kapitel 5.1). Wenn wir den einfachen Adornment-Algorithmus Adn wählen, wird der Adornment-Algorithmus hingegen sehr schnell auf den TGDs des Anwendungsfalles durchgeführt. Werden die TGDs starkt vereinfacht (ohne ihre Terminierungseigenschaften zu verändern), wird auch Adn+ hinreichend schnell ausgeführt. ChaseTEQ macht keine Aussagen über die Azyklizität der Integritätsbedingungen. Wenn man die adornten TGDs jedoch auf Schwache Azyklizität überprüft, erhält man ein Testkriterium, das dem Test auf Azyklizität äquivalent ist [GST11]. Dieser Test wird – für die vereinfachten, abstrakten TGDs – von ChaseTEQ korrekt durchgeführt (Abbildung 3.14 b). Leider gibt ChaseTEQ lediglich die Ergebnisse der einzelnen Tests aus, nicht die möglichen Implikationen für die Terminierung des Chase.

3.2.5. Graal

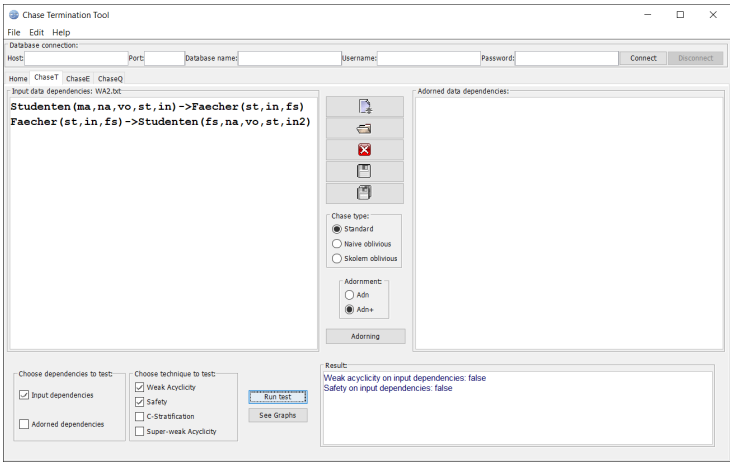
Literaturreview Graal wurde 2015 durch [BLM+15] erstmals vorgestellt. Ein Jahr später beschrieb [Roc16] die Theorie, auf welcher der Terminierungstest von Graal basiert, ohne jedoch auf die praktische Umsetzung einzugehen. Zentrales Ziel von Graal ist die *ontologische Anfragebearbeitung*. Diese erfolgt über vier Hauptkomponenten:

1. Schnittstellen für die Speicherung von Daten, nicht nur als relationale Datenbank, sondern beispielsweise auch als Graphdatenbank;
2. Anwendung von Regeln auf die so definierten Daten;
3. Query Rewriting;
4. weitere Werkzeuge wie Kiabora (siehe unten) und ein Parser für Datalog+.

Ein- und Ausgabe erfolgt im Datalog+-Format über externe Dateien. Eine graphische Benutzeroberfläche für Graal ist nicht verfügbar, allerdings verfügt das Web-Interface von Kiabora über eine derartige Oberfläche.

Der Terminierungstest von Graal wird durch Kiabora ausgeführt, eine Software, die auch unabhängig von Graal über ein Web-Interface nutzbar ist. Interessanterweise weisen die von Kiabora genutzten Terminierungskriterien kaum Übereinstimmungen mit den in der vorliegenden Arbeit beschriebenen Kriterien auf. Terminierende TGD-Mengen werden drei Klassen, charakterisiert durch abstrakte Eigenschaften, zugeordnet: *Finite Expansion Set* (FES), *Finite Unification Set* (FUS) und *Bounded Treewidth Set* (BTS) [BLM+15].

(a)



(b)

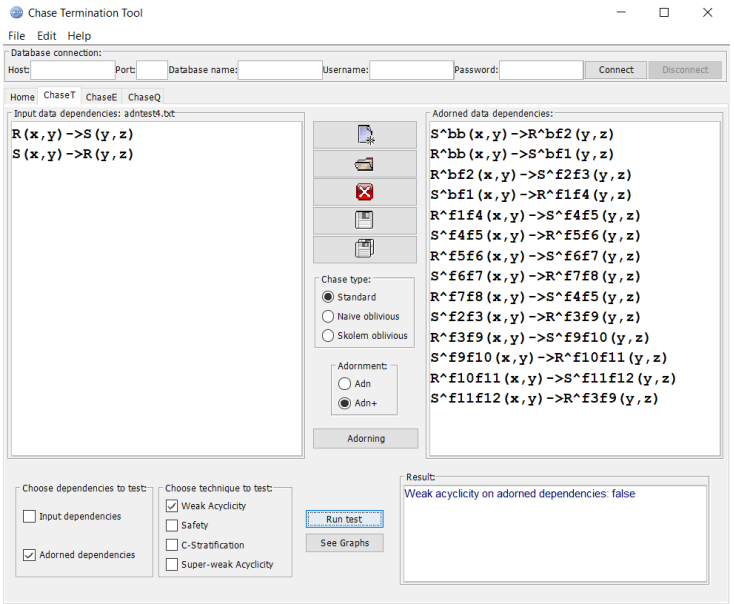


Abbildung 3.14.: Screenshot der Testanwendung von ChaseTEQ. ChaseTEQ erkennt korrekterweise, dass die untersuchten TGDs weder schwach azyklisch, noch safe sind (a). Führt man den Adornment+-Algorithmus auf einem abstrakten Beispiel mit weniger Attributen durch, wird der Algorithmus relativ schnell durchgeführt und erkannt, dass die adornten TGDs nicht schwach azyklisch (und somit die ursprünglichen TGDs azyklisch) sind (b).

Da die abstrakten Eigenschaften nicht nachweisbar sind, überprüft Kiabora stattdessen syntaktische Eigenschaften der TGDs. Durch diese syntaktischen Eigenschaften werden konkrete TGD-Klassen definiert, die einer, keiner oder mehreren der abstrakten Klassen angehören. Schwache Azyklizität (die einzige der Terminierungsklassen, die unter identischem Namen auch in der vorliegenden Arbeit beschrieben wurde) ist beispielsweise eine konkrete Klasse der abstrakten Klasse FES. Kiabora kann dreizehn der so definierten Terminierungskriterien überprüfen.

Obwohl die nähere Untersuchung dieser Terminierungskriterien – insbesondere ihre Vergleichbarkeit mit den in der vorliegenden Arbeit beschriebenen Kriterien – interessant wäre, würde dies den Rahmen der vorliegenden Arbeit sprengen. Am Beispiel Kiabora soll vielmehr gezeigt werden, dass die Klassifikation der Chase-Terminierungskriterien nach [GST11], der wir im wesentlichen folgen, nicht das einzige Klassifizierungssystem darstellt und weitere, hier nicht berücksichtigte Kriterien existieren.

Praktischer Test Eine Instanz der Studentendatenbank – bestehend aus einem Tupel der Studentenrelation – und die beiden TGDs des Anwendungsfalles wurden in Datalog+-Notation definiert und als `.dlp`-Datei gespeichert. Hierbei war es (im Gegensatz zu Llunatic) nicht notwendig, ein Datenbankschema zu definieren. Es scheint nicht nötig zu sein, die so definierten Regeln separat zu untersuchen [Gra], und auch Kiabora scheint nicht als Klasse von Graal implementiert zu sein (zumindest nicht unter diesem Namen).

Stattdessen wurde untersucht, ob es möglich ist, Anfragen an die konzipierte Datenbank zu stellen. Tatsächlich liefert die Anfrage `?(X,Y) :- Faecher(X,Y,Z)` ein Ergebnis (Abbildung 3.15). Fragen wir stattdessen nach `?(X,Y,Z) :- Faecher(X,Y,Z)`, so erhalten wir kein Ergebnis, was aber nichts mit dem Nicht-Terminieren des Chase zusammenhängt, sondern mit der Belegung von Z durch einen Nullwert.

Graal erkennt also trotz der zuvor genannten Möglichkeiten von Kiabora nicht, dass der Chase nicht terminieren würde, und führt einen Chase durch, der wider Erwarten terminiert. Offenbar ist das Verhalten von Graal nicht mit den zuvor (insbesondere in Kapitel 2.2) gemachten konzeptionellen Überlegungen in Übereinstimmung zu bringen.

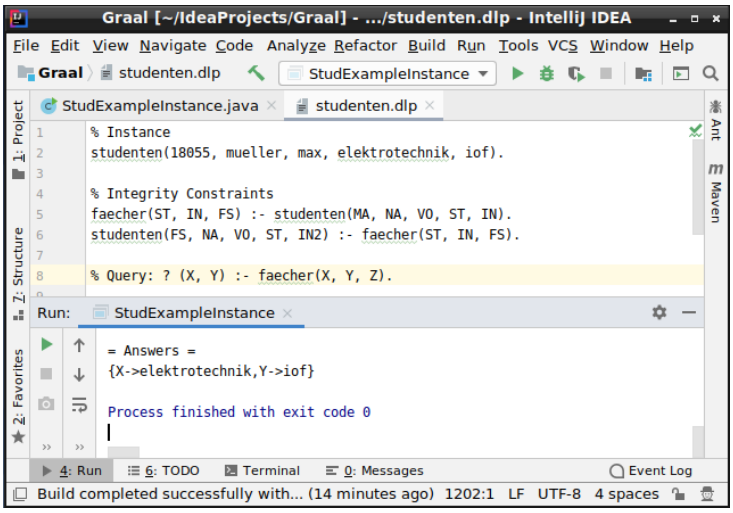


Abbildung 3.15.: Screenshot der Testanwendung von Graal. Obwohl der Chase nicht terminieren sollte, liefert Graal ein Ergebnis für die gegebene Anfrage.

Von den in diesem Unterabschnitt untersuchten Chase-Tools konnte nur Llunatic die Nicht-Terminierung des Chase anhand des von uns gewählten Anwendungsfalls korrekt (und innerhalb eines angemessenen Zeitrahmens) einschätzen. In komplexeren Beispielen – wie fast allen in Unterabschnitt 3.1.1 aufgeführten Beispielen – hätte jedoch auch Llunatic nicht das optimale Ergebnis ausgegeben (d.h. Llunatic hätte

unnötigerweise vor einem nicht terminierenden Chase gewarnt). Im folgenden Kapitel wollen wir die Konzeption und Implementierung eines Terminierungstesters beschreiben, der die Chase-Terminierung zumindest für diese Beispiele korrekt vorhersagt – selbstverständlich können auch wir das grundsätzliche Problem der Unentscheidbarkeit der Chase-Terminierung nicht lösen.

4. Konzept und Implementierung

Das folgende Kapitel beschäftigt sich mit der Konzeption und Implementierung eines Terminierungstesters für das Chase-Werkzeug ChaTEAU. Dies betrifft erstens die zugrundeliegenden Algorithmen (Abschnitt 4.2) und deren Umsetzung (Unterabschnitt 4.3.2), zweitens die benötigten Java-Klassen und deren Beziehungen zueinander (Unterabschnitt 4.3.1) und drittens den Einbau des Terminierungstesters in den Programmablauf (Abschnitt 4.4). Zunächst wollen wir jedoch eine Schlussfolgerung aus Kapitel 3 ziehen und (mindestens) eines der dort definierten Terminierungskriterien für den Terminierungstester auswählen.

4.1. Wahl eines passenden Terminierungskriteriums

In Unterabschnitt 3.1.1 wurde ein ausführlicher Überblick der verschiedenen Terminierungskriterien gegeben. Einige dieser Kriterien sind nicht miteinander vergleichbar – nur weil eine TGD-Menge superschwach azyklisch ist, so ist sie nicht zwingend inductive restricted, und wenn eine andere TGD-Menge inductive restricted ist, so muss sie nicht superschwach azyklisch sein (Abbildung 4.1). Ein Terminierungstester müsste daher beide Kriterien überprüfen, um das sichere Terminieren des Chases auf beiden TGD-Mengen erkennen zu können. Tatsächlich würde jedoch auch das unabhängige Testen beider Terminierungskriterien nicht die Terminierung des Chase auf der Vereinigung der beiden TGD-Mengen erkennen können[1]. Dieses Problem kann nur mit einigem Aufwand gelöst werden (vergleiche Theorem 2.1 Combined Rule Classes in [Roc16]). Tatsächlich können wir aber auf das Testen mehrerer Terminierungskriterien verzichten, da in [GST11] Terminierungsklassen definiert wurden, welche alle in Unterabschnitt 3.1.1 beschriebenen unvergleichbaren Terminierungsklassen einschließen. Eine derartige Klasse ist die Lokale Stratifizierung, allerdings ist die Definition des hiermit assoziierten Kriteriums komplex und abstrakt. Im Gegensatz hierzu existiert für den Test auf Azyklizität durch den Adn++-Algorithmus bereits Pseudocode in [GST11], eine Implementierung dieses Terminierungskriteriums ist also relativ unkompliziert möglich.

Während der ursprüngliche Adn++-Algorithmus fehlerhafte Ergebnisse liefern kann, wenn er auf EGDs angewandt wird, gibt es Erweiterungen des Algorithmus für allgemeine Integritätsbedingungen [CGMT16]. Geht man von der in Unterabschnitt 3.1.2 gegebenen Interpretation der Azyklizität als semi-dynamisches Kriterium aus, liegt nahe, dass sich der Adornment++-Algorithmus in ähnlicher Weise erweitern lässt wie der Chase selbst. Tatsächlich beruht die genannte Erweiterung von Adn++ für EGDs sogar auf einem als Chase-Step bezeichneten Schritt. Wenn die Chase-Funktionalität von ChaTEAU in zukünftigen Arbeiten also z.B. um zusätzliche Operatoren im TGD-Körper oder um Aggregatfunktionen (siehe Abschnitt 6.2) ergänzt wird, könnte der Adn++-Algorithmus jeweils durch einen entsprechenden Chase-Step angepasst werden. In der vorliegenden Arbeit wird jedoch nicht die erweiterte Version von Adn++ aus [CGMT16] implementiert, da diese lediglich die Existenz einer terminierenden Folge von Chase-Schritten untersucht, nicht jedoch, ob alle Folgen von Chase-Schritten (also auch die von ChaTEAU gewählte Schrittfolge) terminieren. Stattdessen soll die praktische Anwendbarkeit der Substitutionslosen Simulation, welcher EGDs

[1] Wir nehmen hier an, dass beide TGD-Mengen über disjunkten Relationenschemata definiert sind, da ansonsten die Vereinigung der TGD-Mengen zum Nicht-Terminieren des Chase führen könnte.

durch TGDs ersetzt, untersucht werden. Hierbei handelt es um einen weiteren Constraint-Rewriting-Algorithmus, von dem bisher noch nicht bekannt ist, ob er (unter Gesichtspunkten der Effizienz) mit dem Constraint-Rewriting des Adn++-Algorithmus kompatibel ist.

Da Azyklizität also nicht nur das mächtigste der vorgestellten Kriterien darstellt, sondern darüber hinaus auch erweiterbar und einfach umzusetzen ist, bildet es den Kern des in der vorliegenden Arbeit zu implementierenden Terminierungstesters für ChaTEAU. Aus Gründen der Vergleichbarkeit soll darüber hinaus ein weiteres, grundlegenderes Kriterium umgesetzt werden. Der einzige von jedem der drei Werkzeuge Llunatic, Graal und ChaseTEQ durchgeführte Terminierungstest ist der Test auf Schwache Azyklizität (siehe Abschnitt 3.2). Da der Test auf Schwache Azyklizität nahezu identisch mit dem Test auf Reiche Azyklizität ist und mit wenig Aufwand zum Test auf Safety erweitert werden kann, sollen diese Tests ebenfalls umgesetzt werden.

Wir dürfen an dieser Stelle jedoch nicht vergessen, dass das Problem der Chase-Terminierung unentscheidbar ist, es also kein optimales Kriterium geben kann. Und obwohl Azyklizität das mächtigste vorgestellte statische Terminierungskriterium ist, gibt es weitere Terminierungsklassen (z.B. von Graal verwendeten Klassen wie „Guarded“ oder „Sticky“), deren Verhältnis zur Klasse der azyklischen TGDs nicht aus der Literatur (z.B. [Roc16]) hervorgeht[2].

Die zu implementierenden Terminierungskriterien sind also:

- Reiche Azyklizität
- Schwache Azyklizität
- Safety
- Azyklizität

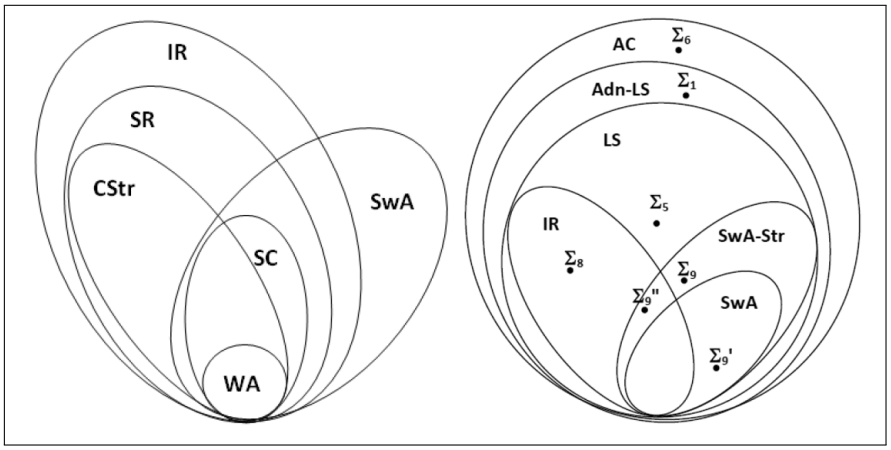


Abbildung 4.1.: Beziehung der Terminierungskriterien untereinander (aus [GST11]). Kriterien wie Superschwache Azyklizität (SwA) und Induktive Restriktion (IR) sind nicht miteinander vergleichbar (d.h. eine Testklasse ist nicht in der anderen enthalten), doch das Kriterium der Azyklizität (AC) schließt alle anderen dargestellten Testkriterien ein.

[2] Auf den ersten Blick scheint diese Unklarheit darauf zurück zu führen zu sein, dass sich jede Gruppe von Autoren nur mit jeweils einer Gruppe von Terminierungskriterien beschäftigt. Tatsächlich trifft dies jedoch nicht zu, und dieselben Autoren haben Arbeiten über den Adornment-Algorithmus [CGMT16], Guarded TGDs [CGP15] und Sticky TGDs [CP19] geschrieben.

4.2. Algorithmen

Im folgenden Abschnitt werde ich die grundlegenden Algorithmen des Terminierungstesters darstellen. Der wichtigste dieser Algorithmen – Adn++ – stammt aus [Spe11]. Ein zentraler Aspekt dieses Algorithmus stellt das Erkennen von Zyklen in einem Graph adornter Prädikate dar. Ein einfacher Algorithmus zum Erkennen derartiger Zyklen (Algorithmus 2) basiert auf dem Prinzip der Tiefensuche. Für das Testen der Schwachen Azyklizität ist hingegen nicht nur das Testen von „normalen“ Zyklen erforderlich, sondern darüber hinaus ein Test, ob der gefundene Zyklus durch eine besondere Kante geht. Für die vorliegende Arbeit wurde Algorithmus 3 als Abwandlung von Algorithmus 2 entwickelt.

Zyklen in einem gerichteten Graph finden

Algorithmus 2, der Zyklen zwischen Knoten eines gerichteten Graphen findet, basiert auf dem Prinzip der Tiefensuche. Für jeden Knoten wird die Funktion DFS (Depth First Search) aufgerufen (Zeile 4). Erreichte Knoten werden als „besucht“ markiert (d.h. sie werden der Menge „visited“ hinzugefügt, Zeile 17), anschließend wird DFS rekursiv für jeden durch eine Kante erreichbaren Knoten aufgerufen (Zeile 19). Abschließend wird der besuchte Knoten als „abgearbeitet“ markiert (d.h. er wird der Menge „finished“ hinzugefügt).

DFS kehrt unter vier verschiedenen Bedingungen zur aufrufenden Funktion zurück:

1. Ein besuchter Knoten wurde bereits als „abgearbeitet“ markiert. Dieser Zweig der Tiefensuche kann nicht zu einem Zyklus führen, da ansonsten der Algorithmus bereits abgebrochen wäre (Fall 2). DFS kehrt also mit dem booleschen Wert *false* zur aufrufenden Funktion zurück (Zeile 12).

2. Ein besuchter Knoten wurde als „besucht“, aber nicht als „abgearbeitet“ markiert. In diesem Fall hat DFS einen Zyklus gefunden und kehrt mit dem booleschen Wert *true* zur aufrufenden Funktion zurück (Zeile 15).

3. Eine Kante führt zu einem Zyklus (d.h. der rekursive Aufruf von DFS liefert den booleschen Wert *true*). Eine Untersuchung weiterer Kanten ist unnötig und der Wert *true* wird zurückgegeben (Zeile 20).

4. Alle Kanten wurden untersucht, aber keine hat zu einem vorzeitigem Abbruch des Algorithmus geführt. Es gibt also keinen vom derzeit untersuchten Knoten erreichbaren Zyklus, weshalb *false* zurück gegeben werden muss (Zeile 24).

Das Markieren von Knoten als „abgearbeitet“ führt einerseits zu einer deutlichen Effizienzsteigerung des Algorithmus (vergleiche die Ausführungen zu Algorithmus 3), andererseits erlaubt es, Zyklen (wie z.B. $N_1 \to N_2 \to N_3 \to N_4 \to N_1$) von zusammenlaufenden Strukturen (wie z.B. $N_1 \to N_2 \to N_3, N_1 \to N_4 \to N_3$) zu unterscheiden ($N_1, N_2, N_3, N_4$ sind jeweils Knoten). Tatsächlich ist die Unterscheidung beider Strukturen von entscheidender Bedeutung für den Adn++-Algorithmus (Algorithmus 4), da beide eine Konsequenz der Substitution sein können, aber nur Zyklen zum Abbrechen des gesamten Algorithmus führen.

An dieser Stelle muss angemerkt werden, dass der vorliegende Algorithmus keinesfalls alle Zyklen findet – dies ist für den Adn++-Algorithmus auch unnötig. Es gibt Kriterien – wie c-Stratifizierung – die eine derartige Variante des Algorithmus benötigen würden. Tatsächlich wäre es für diese Kriterien wünschenswert, weitere Graph-Eigenschaften (wie stark zusammenhängende Komponenten) zu bestimmen.

Da in der vorliegenden Arbeit Stratifizierung jedoch nicht implementiert wird, kann auf die Beschreibung diesbezüglicher Algorithmen verzichtet werden.

Zyklus durch besondere Kanten finden

Um zu Testen, ob ein Zyklus durch eine besondere Kante geht, wurden von mir drei Anpassungen an Algorithmus 2 vorgenommen:

1. Ein zusätzlicher rekursiver Aufruf aller durch besondere Kanten erreichbaren Knoten.
2. Ein Parameter, der anzeigt, ob der gegenwärtig untersuchte Knoten durch eine solche Kante erreicht wurde.
3. Eine Möglichkeit, besondere Zyklen (wie z.B. $N_1 \to N_2 \xrightarrow{*} N_3 \to N_4 \to N_2$) von normalen Zyklen, die über besondere Kanten erreicht wurden (wie z.B. $N_1 \xrightarrow{*} N_2 \to N_3 \to N_4 \to N_2$) zu unterscheiden.

Während Anpassung 1 durch eine weitere Schleifenkonstruktion (Zeile 26 in Algorithmus 3) direkt umsetzbar ist, bieten sich für Anpassung 2 mehrere Varianten an. DFS könnte einen zusätzlichen booleschen Parameter erhalten, der auf *true* gesetzt ist, wenn DFS einen Knoten durch eine besondere Kante erreicht, und ansonsten nicht verändert wird. Wenn DFS nun einen „besuchten“ Knoten erreicht, ist der Rückgabewert der Funktion nicht *true*, sondern der Wert dieses zusätzlichen Parameters. Allerdings könnte der so erweiterte Algorithmus die für Anpassung 3 geforderte Unterscheidung nicht treffen. Man könnte derartige Strukturen nun gänzlich ausschließen und nur noch Zyklen berücksichtigen, die mit dem ersten besuchten Knoten enden (also N_1 im gegebenen Beispiel). Alle anderen gefundenen zyklischen Strukturen würden zum Rückgabewert *false* führen. Dies ist jedoch ineffizient und macht es unmöglich, Knoten als „abgearbeitet“ zu markieren (selbst wenn alle ausgehenden Kanten untersucht wurden, konnten nur Zyklen erkannt werden, die mit dem scheinbar „abgearbeiteten“ Knoten beginnen und enden).

Der oben dargestellte Algorithmus wurde probeweise implementiert und liefert korrekte Ergebnisse, wurde aus Effizienzgründen jedoch nicht für den Terminierungstest der Schwachen Azyklizität verwendet. Der beschriebene boolesche Parameter wird stattdessen durch den Integer-Wert *indexSpecialEdge* ersetzt. Dieser zeigt nicht nur an, dass ein Ast der Tiefensuche durch eine besondere Kante verläuft, sondern auch, von welchem Knoten die letzte derartige Kante ausging (Zeile 27). Hierfür ist es natürlich notwendig, die besuchten Knoten durchzunummerieren, weshalb *visited* keine Menge darstellt, sondern eine Liste ist – die Nummer eines Knotens ist schlicht die Indexposition eines Knotens in dieser Liste. Wenn nun der Index der besonderen Kante (d.h. der Index des aufrufenden Knotens) höher ist, als der Index eines bereits besuchten Knotens (Zeile 15), so befindet sich zumindest die letzte besondere Kante im so erkannten Zyklus. Aussagen über weitere besondere Kanten werden nicht getroffen und sind nicht wesentlich für das Kriterium der Schwachen Azyklizität. Wenn hingegen der Index der besonderen Kante niedriger ist als der Index des Knotens, der den Zyklus schließt, so kann sich keine besondere Kante im Zyklus selbst befunden haben. Der Rückgabewert ist also *false* (Zeile 18), da zwar ein Zyklus gefunden wurde, jedoch keiner, der durch eine besondere Kante führt. Da *visited* den Verlauf eines Astes der Tiefensuche repräsentiert, wird die Knotenliste DFS als zusätzlicher Parameter übergeben und nicht – wie finished – als globale Variable repräsentiert. Wie auch im Algorithmus 2 werden Knoten (bis auf den Test in Zeile 11 von Algorithmus 3) nicht unnötigerweise mehrfach untersucht, wenn sie zuvor als „abgearbeitet“ markiert wurden (Zeile 31). Allerdings setzt dies voraus, dass besondere Kanten vor normalen Kanten besucht werden. Ein Knoten N_2 könnte beispielsweise sowohl durch eine besondere, als auch durch eine normale Kante erreichbar sein: $N_1 \to N_2, N_1 \xrightarrow{*} N_2$. Da diese besondere Kante die einzige besondere Kante des Zyklus $N_1 \xrightarrow{*} N_2 \to N_3 \to N_1$ darstellt, wird durch Verfolgen der Kante $N_1 \to N_2$ kein

besonderer Zyklus erkannt, jedoch N_2 als „abgearbeitet“ markiert, bevor $N_1 \rightarrow N_2$ untersucht werden kann.

Algorithm 2 ZYKLUS_ERKENNEN(E), nach [Wik]

```
1: visited := ∅
2: finished := ∅
3: for each node v ∈ E do
4:    if DFS(v) = true then
5:       return true
6:    end if
7: end for
8: return false
9:
10: DFS(v) :
11: if v ∈ finished then
12:    return false
13: end if
14: if v ∈ visited then
15:    return true
16: end if
17: visited := visited ∪ {v}
18: for each child w of v do
19:    if DFS(w) = true then
20:       return true
21:    end if
22: end for
23: finished := finished ∪ {v}
24: return false
```

Constraint-Rewriting

Im Folgenden soll der Adn++-Algorithmus des Constraint-Rewritings beschrieben werden. Der Algorithmus nimmt eine Menge von TGDs an (Σ) und gibt eine umgeschriebene Menge von TGDs sowie den booleschen Wert Cyc zurück. Im Gegensatz zu Algorithmus 2 und Algorithmus 3 ist der in diesem Unterkapitel gegebene Pseudocode (aus [Spe11]) relativ abstrakt. Hinzu kommt, dass der Algorithmus in der vorliegenden Arbeit nur zur Bestimmung des Cyc-Parameters verwendet wird, und nicht, um adornte TGDs zu generieren, die mit einem weiteren Terminierungstest untersucht werden können (beide Anwendungsfälle des Tests führen zum Terminierungskriterium der Azyklizität, haben also die gleiche Mächtigkeit [GST11]).

Algorithmus 4 lässt sich in drei Abschnitte unterteilen: Initialphase (Zeile 1 bis Zeile 8), Hauptphase (Zeile 9 bis Zeile 39) und Abschlussphase (Zeile 40 bis Zeile 49). Für die Implementierung sind hauptsächlich die ersten beiden Phasen von Bedeutung. Die Initialphase beginnt damit, dass alle TGDs durch adornte TGDs ersetzt werden. Bis auf ihre Adornments handelt es sich bei diesen neuen TGDs um Kopien der ursprünglichen TGDs. Alle Atome und Terme im Körper dieser neuen TGDs erhalten b-Adornments. Abgesehen von den Adornments existenzquantifizierter Variablen (und den entsprechenden Adornments der Atome, welche diese Variablen enthalten) sind auch alle Adornments im Kopf der TGDs b. Existenzquantifizierte Variablen erhalten durch die Funktion `SkHeadAdn` hingegen f-Adornments, d.h. Skolemfunktionen, die von den Adornments der Variablen abhängen, die in Körper und Kopf allquantifiziert sind (Zeile 6). Paarweise unterschiedliche existenzquantifizierte Variablen erhalten jeweils unterschiedliche f-Adornments. Die auf diese Weise neu erzeugten adornten Prädikate werden der Menge New_Pred hinzugefügt. Aus Effizienzgründen werden hierbei keine adornten Prädikate berücksichtigt, die bereits im Körper von TGDs vorkommen (Zeile 4 und Zeile 7).

Algorithm 3 BESONDEREN_ZYKLUS_ERKENNEN(E)

```
1: for each node v ∈ E do
2:    finished := ∅
3:    visited := empty list
4:    if DFS(v, visited, −1) = true then
5:       return true
6:    end if
7: end for
8: return false
9:
10: DFS(v, visited, indexSpecialEdge) :
11: if v ∈ finished then
12:    return false
13: end if
14: if v ∈ visited then
15:    if index of v in visited < indexSpecialEdge then
16:       return true
17:    end if
18:    return false
19: end if
20: visited := visited + v
21: for each child w with special edge v →* w do
22:    if DFS(w, visited, index of v in visited) = true then
23:       return true
24:    end if
25: end for
26: for each child w with normal edge v → w do
27:    if DFS(w, visited, indexSpecialEdge) = true then
28:       return true
29:    end if
30: end for
31: finished := finished ∪ {v}
32: return false
```

Die Hauptphase besteht im Wesentlichen aus zwei Schleifenstrukturen: Zum einen wird über jedes neue adornte Prädikat iteriert (Zeile 9), zum anderen wird für jedes dieser Prädikate über alle adornten TGDs iteriert (Zeile 13). Bei der Implementierung des Algorithmus ist zu berücksichtigen, dass neue Prädikate (Zeile 32) und neue adornte TGDs (Zeile 24, 30 und 35) im Inneren der Schleifenstrukturen entstehen, was von vielen iterierenden Programmstrukturen (z.B. dem `Iterator` aus Java) nicht erlaubt wird.

Die adornte TGD r wird nur dann berücksichtigt, wenn sie von der TGD, welche das aktuell gewählte adornte Prädikat p erzeugt hat (s), getriggert werden kann, was durch die Präzedenzrelation der Stratifikation ausgedrückt wird (Zeile 13). Dies ist eine äußerst bemerkenswerte Einschränkung, da die Präzedenzrelation der Stratifikation nicht erkennt, dass eine TGD eine andere TGD triggern kann, wenn hierfür zusätzlich eine dritte TGD benötigt wird (siehe Kriterium der c-Stratifizierung, Beispiel 3.3). Darüber hinaus könnte s mehrere adornte Prädikate mit dem Relationennamen p erzeugt haben, von denen das hier gewählte p mit keinem Atom von r kompatibel ist (z.B. aufgrund unterschiedlicher Konstanten in denselben Attributen) – der Ausdruck $s < r$ macht jedoch keine Aussagen über die Kompatibilität einzelner Atome.

Das Adornment von p wird nun auf jedes Atom von r mit dem Relationennamen von p angewandt. Hierfür wird die TGD jeweils dupliziert und das Adornment des Atoms durch das Adornment von p ersetzt (Zeile 19), allerdings bleibt das Adornment von Konstanten (bzw. die entsprechende Stelle im Adornment des Atoms) b (Zeile 17). An dieser Stelle des Algorithmus entstehen 2^n Kopien der ursprünglichen TGD (n ist die Anzahl der Atome mit dem Relationennamen von p im Körper von r), sodass alle Kombinationsmöglichkeiten von angewandtem und nicht angewandtem Adornment durchgeführt werden.

Wenn der Körper der erzeugten adornten TGD konsistent ist (d.h. Adornments von Termen und Atomen stehen nicht im Widerspruch zueinander), wird das Adornment des Kopfes der TGD bestimmt (Zeile 22). Selbst wenn die TGD nicht konsistent ist, wird sie der Menge der erzeugten adornten TGDs hinzugefügt (Zeile 35) – schließlich könnte sie später durch Anwendung eines weiteren adornten Prädikats konsistent werden. Für das Adornment des TGD-Kopfes werden die Adornments der Atome an die Adornments der Terme angeglichen, wenn diese auch im Körper der TGD vorkommen. Wenn der Term eine Konstante ist, die nur im Kopf vorkommt, so bleibt das entsprechende Adornment des Atoms b. Für existenzquantifizierte Variablen werden, wie schon zuvor in Zeile 6, f-Adornments erzeugt.

Wenn die neue TGD konsistent ist, wird überprüft, ob eine Substitution durch eine zuvor erzeugte adornte TGD möglich ist. Ist dies der Fall, so wird nicht etwa die TGD selbst der Menge adornter TGDs hinzugefügt, sondern eine neue TGD, die aus dem Körper der neuen TGD und dem Kopf der zuvor erzeugten TGD r^β besteht. Darüber hinaus werden Kanten zwischen dem verwendeten adornten Prädikat p und allen adornten Prädikaten des Kopfes von r^β gezogen. Interessanterweise betrachten wir nur p selbst. Wie zuvor bereits erwähnt werden TGD häufig nur durch eine Kombination mehrerer adornter Prädikate konsistent, doch die zuvor angewandten Prädikate müssen an dieser Stelle nicht berücksichtigt werden.

Wie bereits in der Beschreibung von Algorithmus 3 erwähnt, kann Substitution zu Zyklen im Graph der adornten Prädikate führen – es könnten aber auch z.B. „zusammenlaufende Strukturen“ entstehen. Wird an dieser Stelle ein Zyklus gefunden, so wird die Variable *Cyc* auf *true* gesetzt. Die Implementierung des Algorithmus könnte an dieser Stelle bereits abbrechen – die TGDs sind nicht azyklisch und der Chase terminiert möglicherweise nicht.

Wenn keine Substitution der neuen TGD möglich ist, wird diese der Menge neuer adornter TGDs hinzugefügt und eine Kante zwischen p und jedem adornten Prädikats des Kopfes von r gezogen. Hierbei wird

nicht kontrolliert, ob diese Prädikate des Kopfes bereits zuvor erzeugt wurden. Anschließend werden die Prädikate der Menge neuer adornter Prädikate hinzugefügt.

In der Abschlussphase von Adn++ werden zunächst alle TGDs mit nicht adorntem Kopf entfernt. Hierbei handelt es sich um nicht konsistente adornte TGDs. Anschließend werden zwei zusätzliche Mengen von TGDs eingeführt: Die Menge In enthält eine TGD für jedes Prädikat des Relationenschemas. Der Körper der TGD entspricht dem nicht adornten Prädikat, während der Kopf das jeweilige mit b adornten Prädikat ist (Zeile 42). Die Menge Out wiederum enthält eine TGD für jedes erzeugte adornte Prädikat. Das adornte Prädikat bildet den Körper der TGD, während der Kopf der TGD aus einem neuen, nicht adornten Prädikat der Stelligkeit des adornten Prädikats besteht. Der Rückgabewert des Algorithmus besteht aus einem Zweiertupel, gebildet aus den erzeugten adornten TGDs (inklusive In und Out) und dem Wert Cyc.

Algorithm 4 Adn++(Σ), [Spe11]

Base := Derived := In := Out := New_Pred := E := $\emptyset$
Cyc := false
Let Body$^b(r)$ be the conjunction obtained by adorning atoms in Body(r) with strings of b symbols
Used_Pred := $\{p_r^b | \exists r \in \Sigma$ and $\exists p^b$ in Body$^b(r)\}$
for each applicable $r \in \Sigma$ **do**
  Base := Base $\cup$ $\{$Body$^b(r) \rightarrow$ SkHeadAdn$(r,$ Body$^b(r))\}$
  New_Pred := New_Pred $\cup$ $\{p_r^\alpha | p^\alpha(t) \in$ SkHeadAdn$(r,$ Body$^b(r))\}$ $-$ Used_Pred
end for
while New_Pred $\neq \emptyset$ **do**
  Select nondeterministically $p_s^{a_1..a_n} \in$ New_Pred
  New_Pred := New_Pred $- \{p_s^{a_1..a_n}\}$
  Used_Pred := Used_Pred $\cup \{p_s^{a_1..a_n}\}$
  for each $r \in$ (Base $\cup$ Derived) so that $s <$ src(r) **do**
    for each $p^\beta(x_1, \ldots, x_n) \in$ Body(r) **do**
      B' := Body$(r) - \{p^\beta(x_1, \ldots, x_n)\} \cup \{p^{\gamma_1,\ldots,\gamma_n}(x_1, \ldots, x_n)\}$
      if $x_i \in$ KONST **then**
        $\gamma_i := b$
      else if $x_i \in$ VAR; $(i \in [1..n])$ **then**
        $\gamma_i := \alpha_i$
      end if
      if B' is coherent **then**
        Let H' := SkHeadAdn(Adn$^{-1}(r), B')$
        if $\exists r^\beta \in$ Derived and $\exists$ substitution θ so that $(B' \rightarrow H')\theta = r^\beta$ **then**
          Derived := Derived $\cup \{B' \rightarrow$ Head$(r^\beta)\}$
          E := E $\cup \{(p_s^{\alpha_1..\alpha_n}, p_{src}^\omega(r)) | p^\omega(t) \in$ Head$(r^\beta)\}$
          if r is exist. quantified and E is cyclic **then**
            Cyc := true
          end if
        else
          Derived := Derived $\cup \{B' \rightarrow H'\}$
          $E := E \cup \{(p_s^{\alpha_1..\alpha_n}, p_{src}^\omega(r)) | p^\omega(t) \in H'\}$
          New_Pred := New_Pred $\cup \{p_{src}^\omega(r) | p^\omega(t) \in H' \wedge p_{src}^\omega(r) \notin$ Used_Pred$\}$
        end if
      else
        Derived := Derived $\cup \{B' \rightarrow$ Head(Adn$^{-1}(r))\}$
      end if
    end for
  end for
end while
Delete from Derived constraints with unadorned heads
for each $p(A_1, \ldots, A_n) \in R$ **do**
  In := In $\cup \{p(x_1, \ldots, x_n) \rightarrow p^{b..b}(x_1, \ldots, x_n)\}$
  for each $p(A_1, \ldots, A_n) \in R$ **do**
    for each $p^\alpha(z_1, \ldots, z_n)$ appearing in Base $\cup$ Derived **do**
      Out := Out $\cup \{p^\alpha(x_1, \ldots, x_n) \rightarrow \hat{p}(x_1, \ldots, x_n)\}$
    end for
  end for
end for
return (Base $\cup$ Derived $\cup$ In $\cup$ Out, Cyc)

4.3. Implementierung

Während im vorangegangenen Abschnitt 4.2 drei grundlegende Algorithmen des Terminierungstesters besprochen wurden, wird sich der folgende Abschnitt mit der konkreten Implementierung in ChaTEAU beschäftigen. Zum einen werden die erstellten Klassen und die Beziehung dieser Klassen zueinander dargestellt, zum anderen wird die Implementierung des wichtigsten Algorithmus (Algorithmus 4, Adn++) im Detail beschrieben.

4.3.1. Klassen des Terminierungstesters

Die Terminierungstests lassen sich in zwei Hauptgruppen einteilen: Zum einen sind dies Varianten des Tests auf Schwache Azyklizität, zum anderen handelt es sich um den Test auf Azyklizität und die Erweiterung dieses Tests auf EGDs. Auch die meisten der neu erstellten Java-Klassen lassen sich in diese zwei Gruppen einordnen (siehe Abbildung 4.2). Darüber hinaus wurden Klassen erstellt, welche die Durchführung der Tests koordinieren und den Import von Integritätsbedingungen erleichtern. Letztendlich mussten die neu erstellten Klassen in die existierende Programmstruktur eingefügt werden, was Modifikationen der bereits bestehenden Klassen voraussetzte. Im Folgenden werden die neu erstellten Klassen kurz beschrieben. Die Beziehung dieser Klassen zueinander ist in Abbildung 4.3 als Klassendiagramm dargestellt. Utility-Klassen wie `SfS` und `Parser` weisen zwar keine direkte Assoziation zu den anderen Klassen des Klassendiagramms auf, werden aber durchaus von Klassen derselben Funktionsgruppe (siehe Abbildung 4.2) aufgerufen (z.B. durch Methoden dieser Klassen). Gleichfalls besteht keine Assoziation zwischen `Adornment` und den übrigen am Constraint-Rewriting beteiligten Klassen, obwohl die Modifikation des Adornment ein zentrales Element des Rewriting-Prozesses ist.

ConstraintRewriting

Die Klasse `ConstraintRewriting` enthält Methoden, die für die Bestimmung des Terminierungskriteriums der Azyklizität verantwortlich sind. In erster Linie handelt es sich hierbei um die Implementierung des Adn++-Algorithmus (Algorithmus 4), der im folgenden Unterabschnitt 4.3.2 ausführlicher beschrieben werden soll. Die Implementierung der Initialphase des Algorithmus – die Methode `prepareAdn()` – nimmt als Argument eine Menge von TGDs an. Während der in der Klasse `SfS` implementierte EGD-Rewriting-Algorithmus tatsächlich direkt eine Menge von TGDs erzeugt, liegt in der Regel eine Menge vor, in der sowohl TGDs als auch EGDs vorkommen (z.B. als Rückgabewert der Methode `parseFromFile()` der Klasse `Parser`). Für diesen Fall fungiert die Methode `prepareConstraintAdn()` als Adaptermethode, die EGDs aus den übergebenen TGDs entfernt.

Der Adn++-Algorithmus bestimmt die Azyklizität der übergebenen TGDs, indem ein Graph aus den erzeugten adornten Prädikaten erzeugt und auf Zyklen überprüft wird. Die hierfür benötigte Implementierung von Algorithmus 2 ist jedoch in die Klasse `GraphUtilities` ausgelagert.

GraphUtilities

Im Gegensatz zu `SfS` oder `Parser` handelt es sich bei `GraphUtilities` nicht um eine eigentliche Utility-Klasse – also um eine Menge statischer Attribute und Methoden. Stattdessen sind Instanzen der Klasse `GraphUtilities` Hüllobjekte für eine Menge von Knoten, den Graphen (Kanten sind als Relationen dieser Knoten untereinander realisiert). Darüber hinaus ist in `GraphUtilities` Algorithmus 2 in Form der Methode `sucheZyklus()` implementiert, die Zyklen in diesem Graphen findet. Dem Namen der Klasse

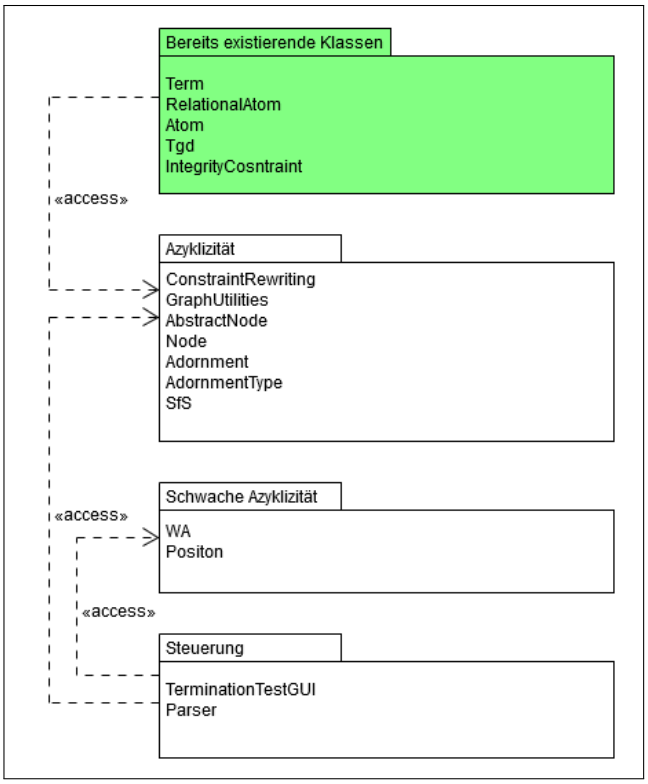


Abbildung 4.2.: Anhand ihrer Funktion lassen sich die neu erstellten und modifizierten Klassen von ChaTEAU in vier Gruppen einordnen: bereits vorhandene (modifizierte) Klassen (grün markiert), für Constraint-Rewriting verwendete Klassen, für den Test auf Schwache Azyklizität (und ähnliche Terminierungskriterien) verwendete Klassen und Klassen, die der Steuerung, Nutzerinteraktion und Ein- und Ausgabe dienen. Bei den dargestellten Gruppen handelt es sich nicht um die verwendeten Java-Pakete. Die Beziehung zwischen den Gruppen (gestrichelte Pfeile) spiegelt die in Abbildung 4.3 dargestellten Assoziationen zwischen den zugehörigen Klassen wieder.

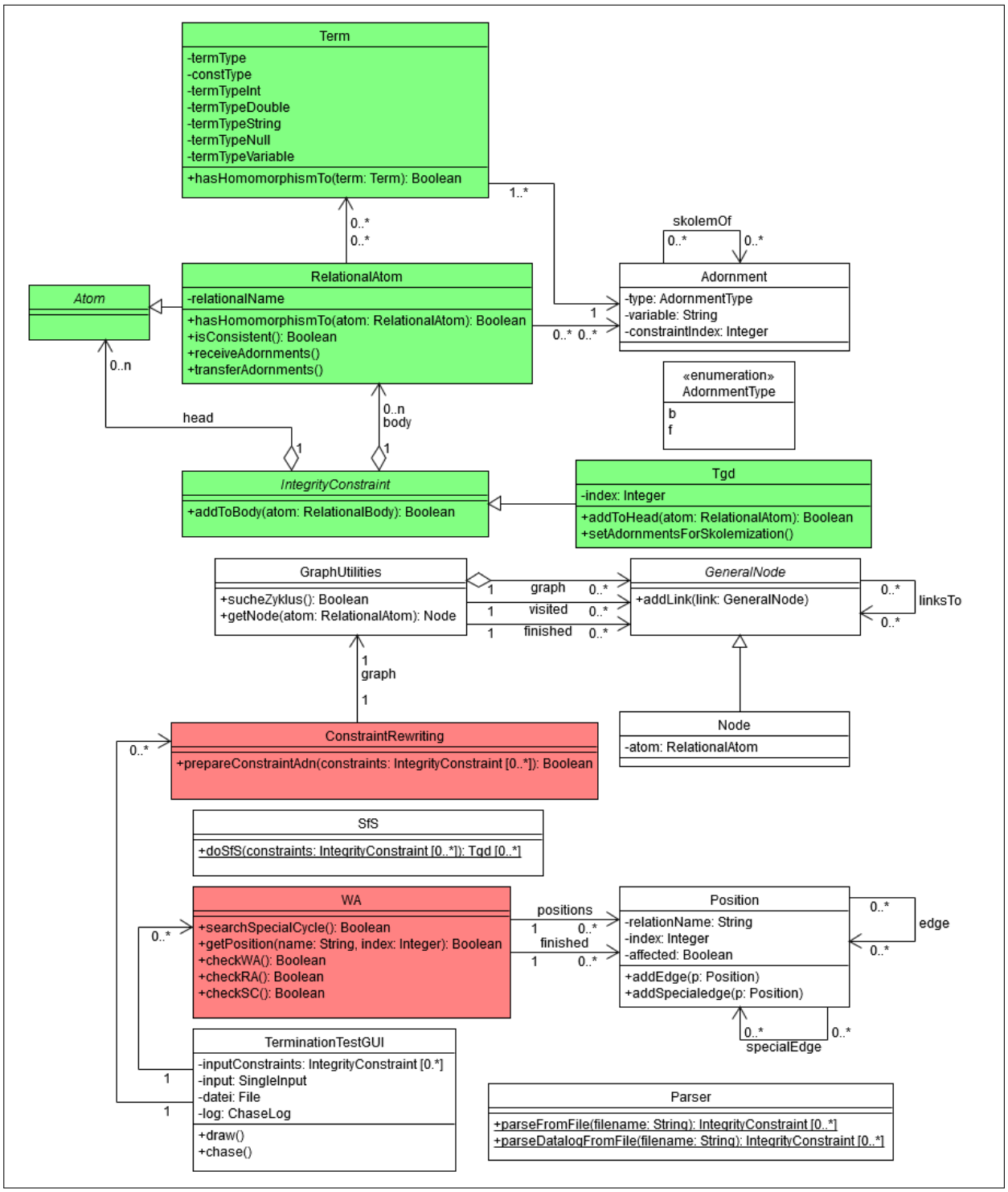


Abbildung 4.3.: Neu erstellte und modifizierte Klassen von ChaTEAU. Die neuen Klassen befinden sich im Modul `terminationTest`. Modifiziert wurden die Klassen `Term`, `Tgd` und `RelationalAtom` (wie in Abbildung 4.2 grün markiert), `IntegrityConstraint` und `Atom` sind nur aus Gründen der Vollständigkeit aufgeführt. Hilfsmethoden und reine Getter-/Setter-Methoden sowie toString- und Hash-Methoden sind nicht dargestellt. Der Adornment++-Algorithmus wird durch die Klasse `ConstraintRewriting` durchgeführt, während die Tests auf Reiche Azyklizität, Schwache Azyklizität und Safety durch Methoden der Klasse `WA` übernommen werden (beide Klassen sind rot markiert).

entsprechend wäre es möglich, weitere Graph-spezifischen Methoden zu implementieren, zum Beispiel Methoden, die alle Zyklen oder sogar alle dicht zusammenhängenden Komponenten des Graphen zurückgeben. Dies ist in der vorliegenden Arbeit nicht erfolgt, da entsprechende Terminierungskriterien – z.B. Stratifizierung – nicht implementiert wurden. Neben den Knoten des Graphen selbst werden zwei weitere Knotenmengen für die Methode `sucheZyklus()` verwendet: *visited*, die besuchten Knoten, und *finished*, die abgearbeiteten Knoten. Sowohl *visited* als auch *finished* sind globale Variablen, die vor Aufruf der Methode `sucheZyklus()` zurück gesetzt werden müssen. Der Graph wird nicht in einem Zug erstellt und anschließend auf Azyklizität getestet, stattdessen ergänzt die Methode `adnPP()` der Klasse `ConstraintRewriting` den Graphen kontinuierlich um neue Knoten und Kanten und überprüft daher regelmäßig, ob ein neuer Zyklus entstanden ist.

GeneralNode

Die Implementierung von Algorithmus 2 in `GraphUtilities` findet Zyklen in einem Graphen aus allgemeinen Knoten. Wesentlich für den Algorithmus ist allein, dass eine Instanz eines solchen `GeneralNode` (bzw. die Instanz einer Unterklasse) durch eine Assoziation (den Kanten, in der Klasse selbst als Link bezeichnet) mit beliebig vielen anderen Knoten verbunden ist.

Node

Prädikatsknoten werden in der Regel nicht explizit durch ihren Konstruktor erzeugt. Stattdessen wird die Methode `getNode()` von `GraphUtilities` aufgerufen, welche einen Prädikatsknoten zu einem übergebenen adornten Prädikat erzeugt, in den Graphen einfügt und zurückgibt, wenn der entsprechende Knoten noch nicht im Graphen vorhanden war. War der Knoten hingegen bereits Teil des Graphen, so gibt `getNode()` diesen Knoten zurück.

Adornment

Der Adn++-Algorithmus der Klasse `ConstraintRewriting` beruht im Wesentlichen auf der Manipulation von Adornments. Gemäß ihrer ursprünglichen Definition [GST11] sind Adornments Strings, die dem Relationennamen von Atomen als Index hinzugefügt werden. Jeder Eintrag dieses Strings ist entweder b oder f_i ($i \in \mathbb{N}^+$) und korrespondiert mit einem Term dieses Atoms. In der vorliegenden Arbeit wurden Adornments hingegen als eigene Objekte konzipiert, die aber auch durch einen String repräsentiert werden können. Die Klasse `ConstraintRewriting` stellt Methoden (z.B. `stringifyTGD()`) bereits, welche String-Repräsentationen adornter TGDs oder adornter Atome generiert (im Gegensatz zu den entsprechenden `toString()`-Methoden, welche Adornments ignoriert). Während Terme über ein einzelnes Adornment verfügen, besitzen relationale Atome über einen Liste von Adornments. Ein Atom ist konsistent, wenn die Adornments der Adornment-Liste mit den jeweiligen Adornments der Terme der Term-Liste übereinstimmen.

Instanzen von `Adornment` gehören entweder dem Adornment-Typ f oder b an. Zunächst besitzen alle Terme über ein b-Adornment. Während b-Adornments keine weiteren Eigenschaften aufweisen und eine String-Repräsentation von `b` haben, hängen f-Adornments von einer Liste anderer Adornments ab (`skolemOf`), nämlich den Adornments der allquantifizierten Variablen des Kopfes der erzeugenden TGD. Darüber hinaus wird ein f-Adornment durch den Index der TGD (`constraintIndex`) und der existenzquantifizierten Variablen (`variable`), die zur Generierung des Adornments geführt haben, charakterisiert. Die String-Repräsentation eines f-Adornments ist wie folgt definiert:

`f^r<constraintIndex>_<variable> (<skolemOf>)`. Im Gegensatz zur ursprünglichen Definition verwenden wir in der vorliegenden Arbeit also keinen Index, der anzeigt, in welcher Reihenfolge die f-Adornments generiert wurden. Eine derartige Indizierung der f-Adornments wäre möglicherweise sinnvoll, wenn der Adn++-Algorithmus um EGDs erweitert würde (ein Adornment f_i könnte dann ähnlich einem Nullwert η_i im Chase behandelt werden, vergleiche [CGMT16]).

WA

Die Klasse `WA` stellt Methoden bereit, die Tests für drei sehr ähnliche Terminierungskriterien implementieren: Schwache Azyklizität (WA), Reiche Azyklizität (RA) und Safety (SC). Reiche Azyklizität bezieht sich auf die Terminierung des Naiven Oblivious Chase, der allerdings nicht von ChaTEAU durchgeführt werden kann, während Tests auf Schwache Azyklizität und Safety die Terminierung des Skolem-Oblivious Chase vorhersagen, der zwar ebenfalls nicht in ChaTEAU implementiert ist, jedoch ähnliche Terminierungseigenschaften wie dieser aufweist. Reiche Azyklizität, Schwache Azyklizität und Safety stellen Terminierungsklassen dar, die ineinander enthalten sind – wenn eine Menge von TGDs also z.B. schwach azyklisch ist, so ist sie garantiert auch safe, aber nicht notwendigerweise reich azyklisch. Letztlich reicht es aus, die Zugehörigkeit einer TGD-Menge zu einer der drei Terminierungsklassen nachzuweisen, um das Terminieren des Skolem-Oblivious Chase (und somit auch des Standard-Chase) zu garantieren.

Die Methoden `checkWA()`, `checkRA()` und `checkSC()` geben jeweils einen booleschen Wert zurück, der anzeigt, ob ein Zyklus im Positionen-Graph gefunden wurde. Im Gegensatz zum Test auf Azyklizität der Klasse `ConstraintRewriting` ist der hierfür benötigte Zyklentest (Algorithmus 3) nicht in eine separate Klasse ausgelagert, sondern wird durch die Methode `findSpecialCycle()` aus `WA` implementiert. Der Kern der drei Tests – das Erzeugen von Positionen und Ziehen von Kanten durch die Methode `drawEdges()` und der Test auf Zyklen durch die Methode `findSpecialCycle()` – ist nahezu identisch. Der Test auf Reiche Azyklizität unterscheidet sich vom Test auf Schwache Azyklizität lediglich durch eine einzelne if-Anweisung der Methode `drawEdges()`, deren Ausführung vom Wert eines booleschen Methodenparameters abhängt. Für den Test auf Safety ist hingegen zusätzlich das Markieren von affected Positionen durch die Methode `markAffected()` notwendig. Im Gegensatz zum Test auf Reiche Azyklizität wird das Ziehen von Kanten, die von nicht-affected Positionen ausgehen, nicht durch eine if-Anweisung in der Methode `drawEdges()` verhindert, stattdessen werden derartige Kanten im Anschluss durch die Methode `removeEdges()` wieder entfernt. Auf diese Weise könnte man prinzipiell auf demselben Graphen zuerst Schwache Azyklizität und anschließend Safety testen – allerdings wurde dies in der vorliegenden Arbeit nicht implementiert.

Position

Instanzen der Klasse `Position` sind durch einen Relationennamen und den Index eines Attributs dieser Relation charakterisiert. Formal könnte eine Position auch durch den Relationennamen und das Attribut selbst definiert sein, wir versuchen in der vorliegenden Arbeit jedoch, Terminierungstests nur auf Basis der Integritätsbedingungen und ohne zusätzliche Schemainformationen durchzuführen. Auf diese Weise können auch Eingabeformate verarbeitet werden, die kein explizites Relationenschema – und somit auch keine Attributnamen – definieren (siehe Klasse `Parser`). Wir gehen in der vorliegenden Arbeit davon aus, dass Relationennamen eindeutig sind. Dies ist in praktischen Anwendungsfällen selbstverständlich nicht immer der Fall, so könnten Quell- und Zieldatenbank Relationen mit identischen Namen besitzen, allerdings wird dieser Fall von ChaTEAU bisher ohnehin noch nicht berücksichtigt. Wenn die adornten TGDs, die durch den Adn++-Algorithmus entstanden sind, auf Schwache Azyklizität oder

Safety untersucht werden sollen, müssten die Relationennamen der Positionen um die Adornments der entsprechenden Prädikate ergänzt werden – allerdings ist der in `ConstraintRewriting` implementierte Adn++-Algorithmus nicht für diesen Verwendungszweck vorgesehen. Abgesehen von den charakterisierenden Attributen Relationenname und Index weist `Position` große Ähnlichkeit mit der Klasse `Node` auf, da beide Klassen das Konzept eines Knotens in einem Graphen implementieren. Es wäre also folgerichtig gewesen, `Position` von `GeneralNode` erben zu lassen, allerdings hätte dies nicht zur Reduktion der Codemenge geführt, da `findSpecialCycle()` und `findeZyklus()` zwar ähnliche Methoden darstellen, sich jedoch nicht aufeinander zurückführen lassen.

SfS

`SfS` stellt eine Utility-Klasse dar, welche eine Implementierung der Substitionslosen Simulation (Substitution-free Simulation, Definition 3.16) in Form der statischen Methode `doSfS()` bereitstellt. Die Methode nimmt eine Menge an Integritätsbedingungen entgegen und gibt eine Menge von TGDs zurück, für die der Chase ähnliche Terminierungseigenschaften aufweist. Hierbei ist zu berücksichtigen, dass es hierbei nur um die Terminierung des Chase geht – die generierten TGDs sollten keinesfalls verwendet werden, um den Chase selbst durchzuführen. Tatsächlich sollten die für den Chase verwendeten Integritätsbedingungen neu erzeugt (z.B. neu aus einer XML-Datei generiert) werden, wenn zwischenzeitlich EGD-Rewriting angewendet wurde. Diese Notwendigkeit besteht nicht, wenn Constraint-Rewriting ohne EGD-Rewriting durchgeführt wurde. Darüber hinaus ist zu beachten, dass Substitionslose Simulation das Relationssymbol Eq verwendet. Eq sollte in keiner der umzuschreibenden Integritätsbedingungen vorkommen, allerdings wurde keine Methode implementiert, die dies überprüft.

Die meisten der in der vorliegenden Arbeit vorgestellten statischen Terminierungskriterien berücksichtigen, entweder explizit anhand eines Graphen oder implizit, ob eine TGD eine andere TGD auslösen (auch als „feuern“ oder „triggern“ bezeichnet) kann. Wenn eine TGD zwei verschiedene Nullwerte erzeugt, eine andere TGD aber an den entsprechenden Stelle gleiche Werte benötigt, so kann die erste TGD die zweite nicht auslösen. Auf diese Weise könnte ein ansonsten nicht terminierender Chase zum Terminieren gebracht werden. Wenn in diesem Fall eine EGD beide Nullwerte gleichsetzt, löst infolgedessen die erste TGD die zweite TGD aus, und der Chase terminiert nicht. Substitionslose Simulation behandelt ausschließlich diese Auswirkung von EGDs – EGDs könnten z.B. auch Nullwerte entfernen und so eine Endlosschleife des Chase zum Terminieren bringen, allerdings ist dieser Effekt von der Reihenfolge der Regelanwendungen abhängig und kann daher nicht auf einfache Weise gelöst werden. Es gibt eine Variante des Constraint-Rewriting [CGMT16], die sich mit diesem Problem beschäftigt, jedoch ebenfalls zu reihenfolgeabhängigen Ergebnissen kommt und in der vorliegenden Arbeit nicht implementiert wurde.

Substitionslose Simulation besteht aus zwei Schritten: EqAx und Singularisierung, welche durch die beiden statischen Methoden `doEqAx()` und `singularize()` implementiert werden. In einem vorbereitenden Schritt werden EGDs durch TGDs substituiert, wobei alle Gleichheitsatome $x = y$ durch Relationsatome $\text{Eq}(x, y)$ ersetzt werden. Die zweistellige Relation Eq repräsentiert also die Gleichheitsrelation. Da diese jedoch eine Äquivalenzrelation ist, erzeugt `doEqAx()` TGDs, die Symmetrie, Reflexivität und Transitivität von Eq sicherstellen. Während `doEqAx()` also EGDs in TGDs umwandelt und neue TGDs erzeugt, manipuliert `singularize()` die bereits vorhandenen TGDs. Zunächst werden Konstanten aus bestehenden Relationsatomen durch neue Variablen ersetzt. Diese neuen Variablen werden durch ein neues Atom der Eq-Relation mit der jeweiligen Konstanten gleichgesetzt. Anschließend werden mehrfach vorkommende Variablen durch paarweise unterschiedliche Variablen ersetzt. Diese Variablen werden ebenfalls über neue Atome der Eq-Relation gleichgesetzt. Interessanterweise erzeugt Schritt 1 von `singularize()` mehrfach vorkommende Variablen, die in Schritt 2 separiert werden müssen. Dies führt zur Generierung unnötiger

Eq-Atome, allerdings nicht zu falschen Ergebnissen. Die durch `doEqAx()` und `singularize()` erzeugten TGD-Mengen werden von `doSfS()` vereinigt und als Ergebnis der Substitutionslosen Simulation zurückgegeben.

TerminationTestGUI

Die Möglichkeiten des Nutzers von ChaTEAU, mit dem Programm zu interagieren, beschränkten sich bisher auf Wahl einer XML-Datei, in welcher die Parameter und Objekte des Chase definiert sind. Es ist möglich, den Terminierungstester (siehe pragmatische Lösung in Abbildung 4.4) ohne zusätzliche Nutzerinteraktion in ChaTEAU einzubinden. Die vorliegende Arbeit beschäftigt sich jedoch weniger mit der Durchführung des Chase durch ChaTEAU, sondern vielmehr mit den unterschiedlichen Terminierungsklassen von Intergritätsbedingungen. Aus diesem Grund soll dem Anwender (bzw. dem Tester der Software) volle Kontrolle darüber gegeben werden, welche Kombination von Terminierungskriterien untersucht werden sollen (auch wenn z.B. Reiche Azyklizität keine praktische Relevanz für ChaTEAU hat). Die hierfür konzipierte Klasse `TerminationTestGUI` besteht aus den Methode `draw()`, welche alle Elemente der GUI erzeugt und die dazugehörigen Listener definiert, und `chase()`, welche den eigentlichen Chase durchführt. Der Code für `chase()` (und einige Abschnitte von `draw()`) wurden aus der Klasse `InputTest` des Moduls `IO` übernommen, die von Fabian Renn erstellt wurde. Es war nicht Teil der Aufgabenstellung, eine graphische Benutzeroberfläche für ChaTEAU zu entwickeln. Wie der Name `TerminationTestGUI` bereits ausdrückt, handelt es sich bei der entwickelten Oberfläche auch lediglich um eine Benutzerschnittstelle des Terminierungstesters, obwohl der Chase selbstverständlich ebenfalls aus der Benutzeroberfläche heraus gestartet werden kann.

Die GUI besteht aus zwei Testfeldern, einer Menüleiste und den Schaltflächen „Termination Test“ und „CHASE“ (Abbildung 5.1). Der Menüeintrag „File“ ermöglicht es, Dateien in drei verschiedenen Eingabeformaten zu öffnen. Hierbei handelt es sich zum einen um XML-Dateien (ChaTEAU-File), in denen Integritätsbedingungen, Datenbankschemata, Datenbankinstanzen und Anfragen definiert sein können. Zum anderen handelt es sich um Textdateien, in denen Integritätsbedingungen als prädikatenlogische Implikationen (Implication-File) oder in einer Datalog-ähnlichen Form (Datalog-File) definiert vorliegen (siehe Beschreibung der Klasse `Parser`). Während nach dem Starten der GUI sowohl die Schaltfläche „Termination Test“, als auch die Schaltfläche „CHASE“ deaktiviert ist, wird „Termination Test“ nach Auswahl einer einzulesenden Datei auswählbar. „CHASE“ wird nur auswählbar, wenn ein ChaTEAU-File im XML-Format eingelesen wurde. Das obere Textfenster (mit dem Titel „Input“) zeigt nach dem Einlesen einer Datei die spezifizierten Integritätsbedingungen und – wenn vorhanden – die Datenbankinstanz an.

Der Menüpunkt „Test Options“ ermöglicht es, eine beliebige Kombination an Terminierungstest auszuwählen. Zur Wahl stehen Reiche Azyklizität, Schwache Azyklizität, Safety, Zyklizität (Constraint Rewriting durch Adn++) und eine Kombination aus EGD-Rewriting (Simulationslose Substitution) und Constraint-Rewriting. Die Reihenfolge der Tests kann nicht festgelegt werden und ist durch die Reihenfolge der Menüeinträge festgelegt. Selbstverständlich können auch einzelne Tests ausgewählt werden. Durch Auswahl der Schaltfläche „Termination Test“ werden alle ausgewählten Tests durchgeführt. Der Vorgang kann beliebig oft (z.B. mit unterschiedlichen Terminierungstests) wiederholt werden. Das Testergebnis wird im zweiten Textfenster (mit dem Titel „Output“) angezeigt. Ist kein Test ausgewählt, so erscheint stattdessen die Fehlermeldung (Please select a termination criterion in the option menu!) im Textfenster. Falls die eingelesene Datei ein ChaTEAU-File war und der Anwender aufgrund der Testergebnisse sicher ist, dass der Chase terminiert, kann dieser durch Auswahl der „CHASE“-Schaltfläche gestartet werden. Die durch den Chase generierte Ergebnisinstanz wird ebenfalls im Output-Fenster angezeigt, zuvor ausgegebene Testergebnisse werden hierdurch überschrieben. Tatsächlich steht es dem Nutzer frei, den Chase zu starten,

ohne dass die Terminierung des Algorithmus garantiert werden kann. Der Terminierungstest stellt zwar die Hauptfunktion der `TerminationTestGUI` dar, ist jedoch rein optional.

Parser

Integritätsbedingungen, Datenbankschemata, Datenbankinstanzen und Anfragen müssen als XML-Datei codiert vorliegen, damit ChaTEAU den Chase durchführen kann. Für das Testen von Integritätsbedingungen ist dies jedoch sehr aufwendig. Insbesondere ist es auf diese Weise nicht möglich, abstrakte Beispiele ohne Datenbankschema zu untersuchen. Die statische Methode `parseFromFile()` der Utility-Klasse `Parser` erlaubt es, Integritätsbedingungen ohne sonstige Zusatzinformationen textuell als prädikatenlogische Implikation zu codieren (ähnlich dem Eingabeformat des Chase-Werkzeugs ChaseTEQ). Die Codierung entspricht der in der vorliegenden Arbeit verwendeten Notation, wobei der Implikationspfeil ($\rightarrow$) durch die Zeichen $->$ ersetzt wird und Quantoren weggelassen werden können. Existenzquantifizierte Variablen sind (ähnlich der Notation von Llunatic) großgeschrieben, während allquantifizierte Variablen kleingeschrieben sind (d.h. der Parser würde auch existenzquantifizierte Variablen im Körper einer TGD akzeptieren). Um bessere Kompatibilität mit Programmen zu garantieren, die diese Konvention nicht einhalten (ChaseTEQ, Graal), werden auch kleingeschriebenen Variablen des Kopfes als existenzquantifiziert interpretiert, wenn sie nicht im Körper vorkommen.

Variablen können darüber hinaus Zahlen enthalten, aber nicht ausschließlich aus Zahlen bestehen. Der Parser erkennt sowohl TGDs als auch EGDs. Verfügt eine Integritätsbedingung über keinen Kopf, wird sie als TGD interpretiert. Abgesehen vom Kopf kann auch der Körper einer Integritätsbedingung fehlen. Obwohl Relationen ohne Attribute in praktischen Anwendungsfällen bedeutungslos sind, ist die Stelligkeit von Atomen nicht notwendigerweise größer als Null. String-Konstanten werden durch Anführungszeichen markiert, während Zahl-Konstanten ohne weitere Kennzeichnung notiert werden können. Zahl-Konstanten können Integer- oder Double-Konstanten sein. Eine derart codierte TGD sähe beispielsweise folgendermaßen aus:

$$\mathrm{R}(x, 2.5, 4) -> \mathrm{S}(x, "const", Y), \mathrm{T}(x).$$

Jede Integritätsbedingung steht in einer eigenen Zeile, Leerzeilen werden ignoriert. Darüber hinaus werden Zeilen ignoriert, die mit %, @ oder /* beginnen, diese Zeichen können also genutzt werden, um Kommentare zu kennzeichnen. Das soeben beschriebene Format wird in der graphischen Benutzerschnittstelle auch als „Implikation“ bezeichnet. Daneben kann der Parser über die Methode `parseFromDatalogFile()` auch ein als „Datalog“ bezeichnetes Datenformat interpretieren. Hierbei handelt es sich keinesfalls um eine exakte Umsetzung des Standards, der von der logischen Programmiersprache Datalog verwendet wird. Stattdessen wird statt der Zeichenfolge $->$ des Implikationsformats die Zeichenfolge $:-$ akzeptiert und die Position von Körper und Kopf der Integrationsbedingungen vertauscht. Alle anderen oben genannten Regeln, wie beispielsweise die Großschreibung existenzquantifizierter Variablen, bleiben weiterhin bestehen. Die im Implikationsformat dargestellte TGD würde im Datalogformat wie folgt notiert werden:

$$\mathrm{S}(x, "const", Y), \mathrm{T}(x) :- \mathrm{R}(x, 2.5, 4).$$

Der Punkt am Ende der Integritätsbedingung ist fakultativ. Enthält eine Zeile die Zeichenfolge $:-$ nicht, so wird sie als Kopf einer körperlosen Integritätsbedingung interpretiert. Obwohl das so beschriebene Format dem Eingabeformat von Graal sehr ähnlich ist, sind beide Formate nicht vollständig kompatibel, so sind Variablen in Graal generell großgeschrieben, während Kleinschreibung den String-Konstanten vorbehalten ist.

Der Parser ist keinesfalls als alternativer Eingabepfad für Integrationsbedingungen konzipiert, die für den Chase verwendet werden sollen. Hierfür wäre es sicherlich notwendig, die generierten Integritätsbedingun-

gen auf Konsistenz, z.B. hinsichtlich des Konstanten-Typs oder der Stelligkeit von Atomen, zu überprüfen. Stattdessen ermöglicht es der Parser, Integritätsbedingungen, die z.B. in der Literatur gefunden wurden, direkt mit dem Terminierungstester zu überprüfen, ohne ein Schema definieren zu müssen. Eine hierfür erstellter Testdatei ist beispielsweise *Beispiel17_Implication.txt*, welche eine TGD-Menge beschreibt, auf welcher der Standard-Chase im Gegensatz zum Skolem-Oblivious Chase terminiert. Die TGDs wurden direkt aus [GMS12] übernommen und weisen daher keinen Bezug zur sonst in der vorliegenden Arbeit verwendeten Studentendatenbank auf. Allerdings ist es nicht möglich, den Chase auf die so definierten TGDs anzuwenden und das Testergebnis zu überprüfen.

4.3.2. Implementierung des Adn++-Algorithmus

Da der Test auf Azyklizität der zentrale – sowohl der mächtigste, als auch der umfangreichste – Test des Terminierungstesters ist, wird die Implementierung des hieran beteiligten Adn++-Algorithmus (Algorithmus 4) im Folgenden näher erläutert. Der vollständige Programmcode des Terminierungstesters – einschließlich des hier beschrieben Codes – ist im Anhang aufgeführt (Abschnitt A.5). Die Methode `prepareAdn()` repräsentiert die Initialphase des Algorithmus, während `adnPP` die Hauptphase des Adn++-Algorithmus implementiert. Die Terminalphase des Algorithmus ist für die vorliegende Arbeit nicht von Bedeutung, da das Ziel der Methode – das Finden von Zyklen im Graph adornter Prädikate – ausschließlich in der Hauptphase erfolgt. Sollte die Methode in Kombination mit einem anderen statischen Terminierungskriterium verwendet werden (was dem Verwendungszweck von Constraint-Rewriting in z.B. ChaseTEQ entspricht), so müsste die Terminalphase des Algorithmus ergänzt werden.

Die Methode `prepareAdn()` (Listing 4.1) bereitet die zu untersuchenden TGDs auf das eigentliche Constraint-Rewriting vor, ohne zusätzliche TGDs zu erzeugen. Im Gegensatz zu Algorithmus 4 manipuliert die Java-Implementierung also die zu untersuchenden TGDs, anstatt in einem ersten Schritt jede TGD zu kopieren. Zunächst werden alle TGDs von der Methode `nameTGDs()` mit einem Index versehen. Werden später Kopien einer TGD erstellt, so kann man unterschiedlich adornte TGDs, die sich von dieser TGD ableiten, an ihrem identischen Index erkennen. Anschließend wird aus dem `Set` der TGDs eine`ArrayList` erzeugt. Dies geschieht, da es mit Java relativ einfach ist, über eine Liste zu iterieren und gleichzeitig Elemente zu dieser Liste hinzuzufügen, was für Mengen nicht der Fall ist (vergleiche die komplexe Konstruktion in Algorithmus 4 mit der einfachen for-Schleife in `adnPP()`). Anschließend werden alle Körperadornments auf b gesetzt. Terme verfügen grundsätzlich zunächst ein b-Adornment. Indem nun durch die Methode `receiveAdornment()` der Klasse RelationalAtome das Adornment der Terme in die entsprechende Position des zugehörigen Atoms kopiert wird, werden diese ebenfalls auf b gesetzt. Bevor mit den Atomen der TGD-Köpfe ebenso verfahren werden kann, müssen die Adornments der existenzquantifizierten Variablen neu bestimmt werden. Dies geschieht durch die Methode `setAdornmentsForSkolemization()` der Klasse `TGD`, welche f-Adornments erzeugt, die von den Adornments der allquantifizierten Variablen des TGD-Kopfes abhängen (an dieser Stelle also von einem Tupel aus b-Adornments).

Im vorangegangenen Schritt wurde eine Reihe neuer adornter Prädikate (d.h. relationale Atome) erzeugt. Die Prädikate des Kopfes werden der Liste `preds` hinzugefügt, wenn sie noch nicht in dieser vorhanden sind. Effizienz an dieser Stelle hat kaum einen Einfluss auf die Laufzeit des Testes, so müssen die bereits in einem TGD-Körper vorhandenen adornten Prädikate etwa nicht gesondert betrachtet werden, wie es in Algorithmus 4 geschieht. Vergleiche zwischen adornten Prädikaten geschehen grundsätzlich anhand von aus den Prädikaten erzeugten Strings (erzeugt durch die Methoden `stringifyAtomhead()` bzw. `stringifyTGD()`), da dies sich als weniger fehleranfällig erwies als ein direkter Vergleich über die `equals`-Methode. Die Liste der vorbereiteten TGDs und die Liste der neu erzeugten adornten Prädikate wird an die Hauptmethode des Algorithmus, `adnPP()`, übergeben. Der Rückgabewert dieser Methode wird zum

Rückgabewert von `prepareAdn()` und gibt Auskunft darüber, ob ein möglicher Zyklus gefunden wurde – *true* bedeutet also, dass die zu untersuchten TGDs nicht azyklisch sind und der Chase möglicherweise nicht terminieren wird.

Das eigentliche Constraint-Rewriting geschieht in der Methode `adnPP()` (Listing 4.2). In zwei verschachtelten Schleifen wird sowohl über alle erzeugten adornten Prädikate, als auch über alle adornten TGDs iteriert. Die Anzahl der vorhandenen Prädikate bzw. TGDs wird im Schleifenkopf nach jedem Schleifendurchlauf aktualisiert. Für jedes Prädikat wird durch die Methode `getNode()` ein Knoten erzeugt und in den Prädikatengraph *graph*, einer Instanz der Klasse `GraphUtilities`, eingefügt, wenn das Prädikat noch nicht vorhanden ist. Anschließend wird für jede adornte TGD überprüft, ob eine `pseudoPrecedence()` zwischen dem Prädikat und der TGD besteht. Algorithmus 4 überprüft stattdessen an dieser Stelle die Präzedenz ($<$, siehe Definition 3.2) zwischen der TGD, die das Prädikat ursprünglich erzeugt hat, und der gerade untersuchten adornten TGD. Da dieser Test nicht nur äußerst aufwendig, sondern auch negative Auswirkungen auf die Unabhängigkeit des Tests von der Reihenfolge der TGDs hat (siehe Beispiel 3.3), wird auf diesen Test an dieser Stelle verzichtet. Ein Prädikat steht dann in Pseudopräzedenz mit einer TGD, wenn im Körper der TGD ein weiteres Prädikat mit folgenden Eigenschaften vorkommt:

1. Beide Prädikate haben das selbe Relationssymbol.
2. Wenn beide Prädikate an einer Position Konstanten tragen, so sind diese Konstanten gleich.
3. Wenn das erste Prädikat an einer Position eine existenzquantifizierte Variable besitzt, so ist die entsprechende Stelle im zweiten Prädikat (d.h. das Prädikat aus dem Körper der TGD) nicht durch eine Konstante belegt.

Weitere Untersuchungen, die z.B. das Vorkommen derselben Variable in mehreren Atomen des Körpers der untersuchten TGD betreffen, können entfallen, da diese durch den Adornment-Algorithmus automatisch ablaufen. Im Gegensatz zur Definition der Präzedenzrelation beschreibt obige Definition eine Relation zwischen einem Prädikat und einer TGD, nicht die Relation zweier TGDs. Der Begriff wird im Folgenden auch für die zwei zueinander passenden Prädikate selbst verwendet. Wenn nun das gerade untersuchte adornte Prädikat mit einem Prädikat der gerade untersuchten adornten TGD in Pseudopräzedenz steht und die Adornments der Prädikate nicht ohnehin bereits identisch sind, wird die TGD dupliziert (über die Methode `duplicateTGD()` und Adornment des Prädikats auf das entsprechende Prädikat der duplizierten TGD übertragen. Hierbei ist zu berücksichtigen, dass ein Prädikat durchaus mit mehreren Atomen im TGD-Körper in Pseudopräzedenz stehen kann und für jedes dieser passenden Atome eine Kopie der TGD erzeugt werden muss – wir können den Test der Pseudopräzedenz also nicht auf den Atomen der kopierten TGD durchführen. Durch die Methode `transferAdornment()` werden die Adornments des Prädikats auf die Terme des Prädikats übertragen. Einige dieser Terme kommen möglicherweise auch in anderen Atomen der TGD vor – tatsächlich handelt es sich bei diesen gleichnamigen Termen aber nicht notwendigerweise um dieselben Java-Objekte. Durch die Methode `propagateOnSameTerm()` werden Adornments gleichnamiger Terme der TGD aneinander angepasst. Durch diese Anpassung kann es nun zu Abweichungen zwischen den Adornments der Terme und der Adornments der Atome kommen. Derartige Inkonsistenzen werden durch die Methode `headAdornment()` erkannt. Selbst wenn eine Inkonsistenz auftritt, muss die neu adornte TGD nicht verworfen werden – es könnte nämlich sein, dass ein weiteres adorntes Prädikat benötigt wird, um eine konsistent adornte TGD zu generieren. Die adornte TGD wird also also der Liste der zu untersuchenden TGDs hinzugefügt, wenn sie noch nicht in dieser vorhanden ist. An dieser Stelle gelten zwei TGDs als gleich, wenn ihr Körper übereinstimmen – da `headAdornment()` vorzeitig abgebrochen wurde, ist das Adornment des TGD-Kopfes wenig aussagekräftig.

Wenn die neu adornte TGD über einen konsistenten Körper verfügt, passt `headAdornment()` ihren Kopf entsprechend an und erzeugt wenn nötig neue f-Adornments. Ist die so angepasste TGD noch nicht in

der Liste adornter TGDs vorhanden, wird sie dieser hinzugefügt. Als nächstes wird die adornte TGD mit allen bisher erzeugten TGDs verglichen und auf ihre Substituierbarkeit hin untersucht. Zwei TGDs sind substituierbar, wenn sie sich von der selben ursprünglichen TGD ableiten (d.h. denselben Index tragen) und alle Adornments bis auf Stellen unterschiedlicher f-Adornments identisch sind. Beide TGDs müssen in derartigen Stellen f-Adornments tragen. Außerdem soll eine rechtseindeutige Abbildung von den Adornments der neu entstandenen TGD zu den Adornments der bereits vorhandenen TGD definiert werden können. Des Weiteren muss die Menge der unterschiedlichen Nullwerte der einen TGD von der Menge der unterschiedlichen Nullwerte der anderen TGD disjunkt sein. Wurde eine derartige Substitution gefunden, so wird ein Knoten für jedes Atom des Kopfes der bereits vorhandenen TGD erzeugt und eine Kante vom Knoten des zu untersuchenden adornten Prädikats (das in den Körper eingesetzt worden war) zu diesen neuen Knoten gezogen. Anschließend wird der boolesche Wert *cyc* auf *true* gesetzt. Obwohl eine erfolgreiche Substitution häufig mit der Identifikation eines Zyklus des Prädikatengraphen einhergeht, ist dies nicht immer der Fall – *cyc* hat also nicht die gleiche Bedeutung wie die gleichnamige boolesche Variable aus Algorithmus 4. Wenn nämlich ein Zyklus im Prädikatengraph gefunden wird, so endet `adnPP()` ohnehin (und gibt den Wert *true* zurück), weshalb die Belegung einer lokalen Variablen bedeutungslos wäre.

Wenn keine Substitution möglich war – *cyc* also den Wert *false* hat – werden alle Prädikate des TGD-Kopfes auf ihre Vorhandensein in der Liste der adornten Prädikate hin untersucht. Ist ein Prädikat noch nicht vorhanden, so wird es der Liste hinzugefügt. Außerdem wird ein entsprechender Knoten erzeugt und der Prädikatengraph um diesen Knoten ergänzt. Vom Knoten des zu untersuchenden adornten Prädikats (das in den Körper eingesetzt worden war) werden Kanten zu jedem dieser neuen Knoten gezogen. Wenn über alle Prädikate und TGDs iteriert worden ist, ohne dass ein Zyklus im Prädikatengraph gefunden werden konnte, gibt `adnPP()` den Wert *false* zurück. Dies dauert meist wesentlich länger als das Finden eines Zyklus.

```
1  public boolean prepareAdn(Set<Tgd>tgds) {
2   // gibt urspruenglichen TGDs eindeutigen Index
3   nameTGDs(tgds);
4   ArrayList<Tgd>tgdlist=setToList(tgds);
5   ArrayList<RelationalAtom>preds=new ArrayList<RelationalAtom>();
6   for(Tgd t: tgdlist) {
7    for(RelationalAtom a: t.getBody()) {
8     // Adornments der Terme werden zu Adornments des Atoms
9     a.receiveAdornments();
10   }
11   // erzeuge f-Adornments der existenzquantifizierten Variablen
12   t.setAdornmentsForSkolemization();
13   for(RelationalAtom a: t.getHead()) {
14    a.receiveAdornments();
15   }
16   for(RelationalAtom head: t.getHead()) {
17    boolean unknown=true;
18    for(RelationalAtom pred: preds) {
19     if(stringifyAtomhead(pred).equals(stringifyAtomhead(head)))unknown=false;
20    }
21    if(unknown) {
22     // neue Praedikate werden gespeichert
23     preds.add(head);
24    }
25   }
26  }
```

```
27  return adnPP(preds, tgdlist);
28 }
```

Listing 4.1: Initialphase Adn++

```
1  public boolean adnPP(ArrayList<RelationalAtom> preds, ArrayList<Tgd> tgdlist) {
2   boolean cyk=false;
3   // um effizient Koerper und Kopf neuer und alter TGDs ueber ihren Hash zu
        vergleichen
4   HashSet<String>bodystrings=new HashSet<String>();
5   HashSet<String>headstrings=new HashSet<String>();
6   for(Tgd t: base) {
7    String tst=stringifyTGD(t);
8    bodystrings.add(tst.split("->")[0].trim());
9    headstrings.add(tst.split("->")[1].trim());
10  }
11  // Zyklus durch adornte Praedikate
12  for(int predindex=0;predindex<preds.size();++predindex){
13   RelationalAtom p=preds.get(predindex);
14   // erzeugt den Praedikat-Knoten, wenn noch nicht vorhanden
15   GraphUtilities.Node node1=graph.getNode(p);
16   // Zyklus durch adornte TGDs
17   for(int tgdindex=0; tgdindex<tgdlist.size();++tgdindex){
18    Tgd t=tgdlist.get(tgdindex);
19    // p kann in den Koerper von t eingesetzt werden und ist kompatibel
20    if(pseudoPrecedence(p, t)){
21     for(RelationalAtom p2: t.getBody()) {// Zyklus durch alle Koerperatome
22      if(pseudoPrecedence(p, p2)&&(!p.getAdornments().equals(p2.getAdornments()))
           &&(isB(p2))) {
23       Tgd adornedTGD=duplicateTGD(t);
24       for(RelationalAtom p3: adornedTGD.getBody()) {
25        if(p2.equals(p3)) {
26         p3.setAdornments(p.getAdornments());
27         // Adornments werden auf Atom uebertragen
28         for(int i=0;i<p3.getTerms().size();++i) {
29          Term con=p3.getTerms().get(i);
30          if(con.getTermType()==TermType.Const)p3.getAdornments().set(i, new
               Adornment());
31         }
32         // Adornments werden zu Adornments der Terme
33         p3.transferAdornments();
34         // gleichnamige Terme erhalten gleiche Adornments
35         propagateOnSameTerm(p3, adornedTGD);
36        }
37       }
38       // Adornments des TGD-Koerpers nicht konsistent
39       if(!headAdornment(adornedTGD)) {
40        boolean tgdisnew1=true;
41        if(bodystrings.contains(stringifyTGD(adornedTGD).split("->")[0])) tgdisnew1
             =false;
42        if(tgdisnew1) {
43        // nicht konsistente TGD wird vielleicht spaeter konsistent  wenn weitere
             adornte Praedikate eingefuegt wurden
44         tgdlist.add(adornedTGD);
```

```
45          String atgt=stringifyTGD(adornedTGD);
46          bodystrings.add(atgt.split("->")[0].trim());
47          headstrings.add(atgt.split("->")[1].trim());
48         }
49        }
50        // Adornments sind konsistent, Adornments fuer existenzquant. Variablen im
              Kopf wurden neu bestimmt
51        else {
52         boolean tgdisnew=true;
53         if(headstrings.contains(stringifyTGD(adornedTGD).split("->")[1].trim()))
               tgdisnew=false;
54         if(!bodystrings.contains(stringifyTGD(adornedTGD).split("->")[0].trim()))
               tgdlist.add(adornedTGD);
55         // hier erfolgt Test auf Substitution der Adornments
56         if(tgdisnew) {
57          String btgd=stringifyTGD(adornedTGD);
58          bodystrings.add(btgd.split("->")[0].trim());
59          headstrings.add(btdg.split("->")[1].trim());
60          cyk=false;
61          for(Tgd oldTGD: tgdlist) {
62           // die Adornments der TGDs sind substituierbar
63           if(substitutable(adornedTGD, oldTGD)) {
64            for(RelationalAtom sub: oldTGD.getHead()) {
65             GraphUtilities.Node node3=graph.getNode(sub);
66             // Praedikatengraph wird erweitert
67             node1.addLink(node3);
68            }
69            // TGDs nicht azyklisch, Chase terminiert wohl nicht
70            if(graph.sucheZyklus()) return true;
71            // durch Substitution wurde kein Zyklus, sondern eine "zusammenlaufende
                  Struktur" gefunden
72            cyk=true;
73           }
74          }
75         }
76         // da Substitution nicht erfolgreich war, Speichern aller neuen adornten
               Praedikate des Kopfes
77         if(tgdisnew&&(!cyk)) {
78          for(RelationalAtom pred: adornedTGD.getHead()) {
79           boolean unknown=true;
80           for(RelationalAtom exis: preds) {
81            if((stringifyAtomhead(pred).equals(stringifyAtomhead(exis))))unknown=
                  false;
82           }
83           if(unknown) {
84            preds.add(pred);
85            GraphUtilities.Node node2=graph.getNode(pred);
86            // Praedikatengraph wird erweitert
87            node1.addLink(node2);
88           }
89          }
90         }
91        }
```

```
92          }
93        }
94      }
95    }
96  }
97  // da kein vorlaeufiger Abbruch nach Finden eines Zyklus: TGDs sind azyklisch,
        Chase terminiert garantiert
98  return false;
99 }
```

Listing 4.2: Hauptphase Adn++

4.4. Einbindung in ChaTEAU

Grundsätzlich bieten sich zwei Möglichkeiten an, einen Terminierungstester in ChaTEAU einzubinden: die pragmatische Variante und die entkoppelte Variante. Für die pragmatische Variante steht die Durchführung des eigentlichen Chase im Vordergrund, der Terminierungstest sollte so schnell wie möglich – und daher mit möglichst wenigen überprüften Terminierungskriterien – durchgeführt werden und nicht unabhängig vom Chase stehen (Aktivitätsdiagramm in Abbildung 4.4). Hierfür wird erst ein Test durchgeführt, der nur polynomiale Zeit benötigt (also z.B. Schwache Azyklizität oder Safety). Wenn dieser Test bereits Terminierung des Chase garantiert, wird dieser sofort gestartet. Wenn jedoch an dieser Stelle noch kein sicheres Ergebnis vorliegt, wird ein langsamerer, mächtigerer Test durchgeführt (wie z.B. Constraint-Rewriting, welches im schlechtesten Fall exponentielle Zeit benötigt). Auch in diesem Fall wird der Chase sofort gestartet, wenn seine Terminierung garantiert ist. Wenn eine Terminierung hingegen nicht garantiert werden kann, ist es nicht möglich, den Chase manuell zu starten.

Für die entkoppelte Variante stehen die Terminierungstests selbst im Vordergrund: Mehrere von diesen können unabhängig voneinander durchgeführt werden, auch wenn hierdurch keine zusätzlichen Informationen über die Terminierung des Chase entstehen (Aktivitätsdiagramm in Abbildung 4.5). Die Durchführung des Chase ist ebenfalls unabhängig von diesen Tests. Der Nutzer könnte beispielsweise einige der Tests auswählen und anschließend selbst entscheiden, ob er den Chase durchführt (die Wahlmöglichkeit des Nutzers ist im Aktivitätsdiagramm nicht dargestellt). Er könnte auch den Chase durchführen, ohne zuvor dessen Terminierung zu überprüfen. Tatsächlich gibt es zahlreiche Fälle, in denen selbst Constraint-Rewriting fälschlicherweise ein Nicht-Terminieren des Chase vorhersagt (z.B. wenn EGDs vorhanden sind).

In der vorliegenden Arbeit haben wir uns für die entkoppelte Variante entschieden. Die Terminierung des Chase auf allen Instanzen ist unentscheidbar, und die Terminierung des Chase auf einer bestimmten Instanz wird von statischen Terminierungskriterien gar nicht erst untersucht. Ein Nutzer sollte also die Möglichkeit haben, einen (für ihn) offensichtlich terminierenden Chase durchzuführen, selbst wenn die Terminierung durch keines der implementierten Kriterien erkannt werden kann. Darüber hinaus sind TGD-Terminierungsklassen auch für sich allein bereits ein interessanter Forschungsgegenstand – in Unterabschnitt 3.1.1 haben wir schließlich lediglich gezeigt, dass es Beispiele für jede der Klassen gibt, aber nicht, wie groß die einzelnen Klassen sind oder gar ob praktisch relevante TGD-Mengen in spezifische Klassen fallen.

Nachdem wir in diesem Kapitel beschrieben haben, wie der Terminierungstesters für ChaTEAU konzipiert und als Java-Programm implementiert wurde, wollen wir im folgenden Kapitel die Möglichkeiten dieses Tools auswerten.

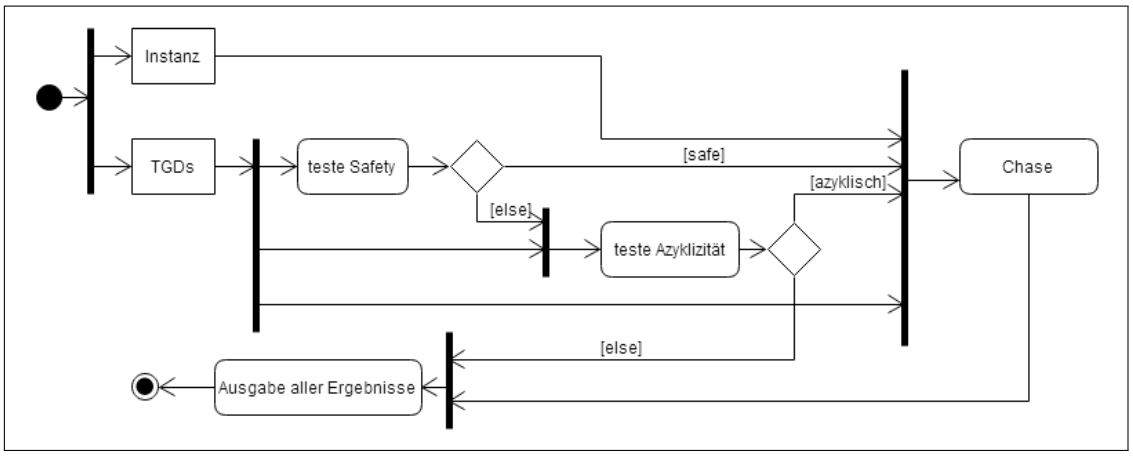


Abbildung 4.4.: Aktivitätsdiagramm eines möglichen Workflows (pragmatische Variante): Der Chase wird nur durchgeführt, wenn seine Terminierung nachgewiesen werden konnte. Wenn dies bereits durch den ersten Terminierungstest geschieht, sind keine weiteren Test nötig. Da sowohl Schwache Azyklizität als auch Safety in polynomialer Zeit überprüft werden können, kann auf das schwächere Kriterium (Schwache Azyklizität) verzichtet werden.

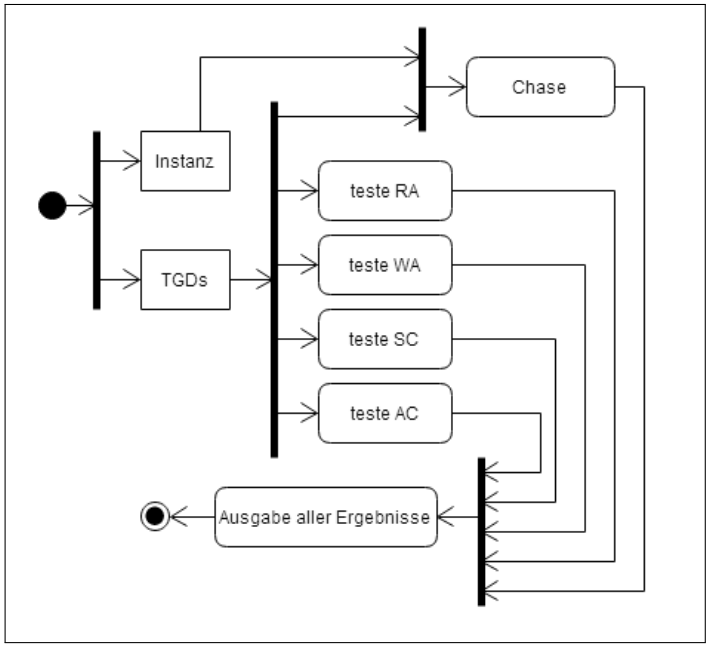


Abbildung 4.5.: Aktivitätsdiagramm eines möglichen Workflows (entkoppelte Variante): Sowohl Chase als auch die einzelnen Terminierungstests werden unabhängig voneinander durchgeführt. RA: Reiche Azyklizität, WA: Schwache Azyklizität, SC: Safety, AC: Azyklizität.

5. Auswertung

Im Folgenden soll der Terminierungstester für ChaTEAU, dessen Konzeption und Implementierung im Kapitel 4 beschrieben wurde, anhand verschiedener Testszenarien evaluiert werden. Einerseits ist hierbei von Bedeutung, dass der Terminierungstester korrekte Ergebnisse generiert (soweit die Unentscheidbarkeit der Chase-Terminierung dies zulässt), andererseits soll die benötigte Laufzeit des Terminierungstests die für den Chase benötigte Gesamtlaufzeit nicht unnötig verlängern – denn auch ohne eine explizites Terminierungskriterium kann die Nicht-Terminierung des Chase zumindest vermutet werden, wenn dessen Laufzeit einen zuvor festgelegten Maximalwert überschreitet. Am Beispiel ChaseTEQ konnten wir sehen, dass insbesondere die Durchführung des Adornment-Algorithmus unverhältnismäßig viel Zeit benötigen kann.

5.1. Beispielanwendung

Im Folgenden soll die Anwendung des Terminierungstesters an drei beispielhaften Integritätsbedingungsmengen demonstriert werden. Für Anwendungsfall 1 erwarten wir – basierend auf unseren theoretischen Vorüberlegungen – dass der Chase terminiert, für Anwendungsfall 2 erwarten wir, dass der Chase nicht terminiert, und für Anwendungsfall 3 erwarten wir, dass der Chase aufgrund der Auswirkungen einer EGD nicht terminiert. Es wurden absichtlich Integritätsbedingungsmengen ausgewählt, die vom Terminierungstester in überschaubarer Zeit bearbeitet werden können – Ziel ist es an dieser Stelle also nicht, die Belastungsgrenzen der Anwendung zu testen.

Die Anwendungsfälle liegen als XML-Dateien vor. Diese können aus der GUI heraus geladen werden (Abbildung 5.1 a). Wurde ein Anwendungsfall geladen, so werden in der XML-Datei spezifizierte Datenbankinstanzen und Integritätsbedingungen im Input-Textfeld angezeigt. Über das Menü ist es möglich, beliebige Kombinationen durchzuführender Terminierungstests auszuwählen (Abbildung 5.1 b). Für die drei zu untersuchenden Anwendungsfälle wählen wir alle verfügbaren Kriterien – Reiche Azyklizität, Schwache Azyklizität, Safety, Azyklizität und Azykliziät nach Durchführung von Substitutionsloser Simulation (EGD-Rewriting) aus. Die Kriterien werden in dieser Reihenfolge getestet, eine Anpassung der Reihenfolge ist (zumindest über die GUI) nicht möglich.

Anwendungsfall 1 besteht aus den TGDs r_1, r_2 und r_3:
$r_1 : \text{No}(ma, mo, no) \rightarrow \exists VO1, \ddot{A}Q1, MA, MO1, MO3, MO4 : \text{Mo}(mo, VO1, \ddot{A}Q1),$
$\text{Ko}(MA, MO1, \ddot{A}Q1, MO3, MO4)$
$r_2 : \text{Mo}(mo, vo, \ddot{a}q), \text{Ko}(ma, mo, \ddot{a}q, mo3, mo4) \rightarrow \exists MA, NO : \text{No}(MA, \ddot{a}q, NO)$
$r_3 : \text{Mo}(mo, vo, \ddot{a}q) \rightarrow \exists VO2 : \text{Mo}(\ddot{a}q, VO2, mo).$

In Unterabschnitt 3.1.1, Beispiel 3.8 dienten diese TGDs dazu, das Kriteriums der Lokalen Stratifizierung zu veranschaulichen. Im Terminierungstester ist zwar nicht das Kriterium der Lokalen Stratifizierung implementiert, da gemäß [GST11] jedoch die Terminierungsklasse der Azyklizität die Klasse der Lokalen Stratifizierung einschließt, erwarten wir, dass der Terminierungstester die Terminierung des Chase auf diesen TGDs erkennt.

Wir öffnen die Beispieldatei `Beispiel14_LS.xml`. Die im Input-Fenster angegebene Reihenfolge der TGDs stimmt nicht notwendigerweise mit der Reihenfolge überein, in der sie in der XML-Datei definiert wurden; tatsächlich ändert sich die dargestellte Reihenfolge, wenn wir die Quelldatei erneut öffnen. Dieser fehlende Determinismus der Reihenfolge wirkt sich auf die Geschwindigkeit des Constraint-Rewritings aus – allerdings nur im Fall des nicht terminierenden Chase. Das Ergebnis der ausgewählten Terminierungstests wird im Output-Textfeld angezeigt (Abbildung 5.2). Wie zu erwarten können die weniger mächtigen Kriterien das Terminieren des Chase nicht erkennen. Wir erkennen außerdem, dass EGD-Rewriting (bei Abwesenheit von EGDs) das Ergebnis des Tests auf Azyklizität nicht verändert hat – die TGDs sind azyklisch und der Chase wird definitiv terminieren. Wir könnten dieses Testergebnis nun verifizieren, indem wir die CHASE-Schaltfläche betätigen und den Chase tatsächlich durchführen – da wir an dieser Stelle jedoch nicht die Chase-Funktionalität von ChaTEAU evaluieren wollen, verzichten wir darauf, dies darzustellen.

Anwendungsfall 2 besteht aus den TGDs r_4 und r_5:
$r_4 : \text{St}(ma, na, vo, st, in) \rightarrow \exists FS : \text{Fä}(st, in, FS)$
$r_5 : \text{Fä}(st, in, fs) \rightarrow \exists NA, VO, IN : \text{St}(fs, NA, VO, st, IN)$.

Diese TGDs ähneln den in Beispiel 3.1 verwendeten Integritätsbedingungen, allerdings wurde die allquantifizierte Variable in von r_2 aus Beispiel 3.1 (bzw. r_5 in Anwendungsfall 2) durch die existenzquantifizierte Variable IN ersetzt. Wir erwarten, dass durch diese Modifikation die TGDs ihre Schwache Azyklizität verlieren, dies bedeutet jedoch noch nicht, dass auch die anderen Kriterien verletzt werden.

Wir öffnen hierfür Beispieldatei `Beispiel7_WA.xml` und führen sämtliche zur Verfügung stehenden Terminierungstests durch. Tatsächlich kann das Programm nachweisen, dass r_4 und r_5 nicht nur nicht schwach azyklisch, sondern auch nicht safe und nicht azyklisch sind. Auch EGD-Rewriting ändert nichts an diesem Ergebnis (Abbildung 5.3). Wie zu erwarten, geht der Verlust der Schwachen Azyklizität mit einer Abwesenheit von Reicher Azyklizität einher, allerdings sind auch die TGDs aus Beispiel 3.1 nicht reich azyklisch. Tatsächlich beweist das Ergebnis des Terminierungstesters nicht, dass der Chase tatsächlich nicht terminieren wird (auf einer leeren Instanz würde er z.B. terminieren) – wir überlassen also dem Nutzer die Entscheidung, ob er das Risiko tragen will, die CHASE-Schaltfläche auszuwählen. Auf der in `Beispiel7_WA.xml` definierten Datenbankinstanz terminiert der Standard-Chase tatsächlich nicht. Es wurde keine spezielle Möglichkeit vorgesehen, den Chase während seiner Durchführung aus der GUI heraus zu unterbrechen.

Anwendungsfall 3 besteht aus den TGDs r_6 und r_7:
$r_6 : Module(mo1, mo1, äq1) \rightarrow \exists MO2, VO2 : Module(MO2, VO2, mo1)$
$r_7 : Module(mo1, vo1, äq1) \rightarrow mo1 = vo1$.

Diese Integritätsbedingungen stellen ein Beispiel für die Rolle von EGDs bei der Nicht-Terminierung des Chase dar. Die TGD r_6 stellt die (praxisferne) Forderung, dass Module, die sich selbst voraussetzen, Äquivalenzmodul für ein anderes Modul sein müssen. EGD r_7 wiederum fordert, dass sich alle Module selbst voraussetzen. Während durch r_6 also für Modul und Voraussetzungsmodul zwei verschiedene Nullwerte gesetzt werden (weshalb sich r_6 zunächst nicht selbst triggert), setzt r_7 diese Nullwerte gleich und führt so zum Nicht-Terminieren des Chase.

Wir öffnen für diesen Anwendungsfall die Beispieldatei `Beispiel16_EGD.xml`. Nach Durchführung der Terminierungstests zeigt sich, dass Constraint-Rewriting zum unerwarteten (und falschen) Ergebnis gekommen ist, die Terminierung des Chase sei garantiert (Abbildung 5.4). Alle übrigen Terminierungstests – Reiche Azyklizität, Schwache Azyklizität, Safety und Constraint-Rewriting nach EGD-Rewriting – geben hingegen eine Warnung aus, der Chase würde möglicherweise nicht terminieren. Tatsächlich

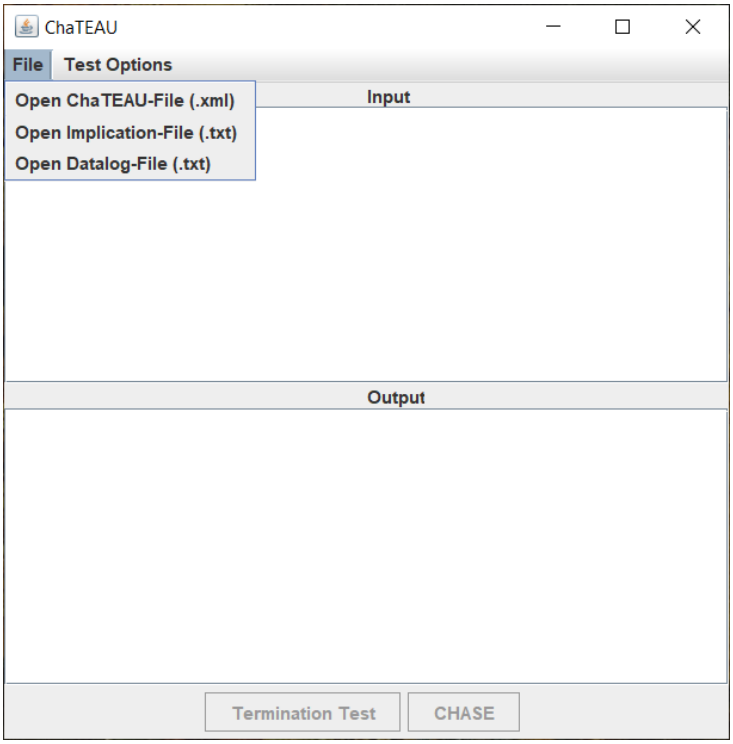
kann keines der implementierten statischen Testverfahren EGDs analysieren – auch ohne r_7 würden die drei Varianten der Schwachen Azyklizität das Nicht-Terminieren des Chase vorhersagen. Nur nach dem Umschreiben der Integritätsbedingungen in äquivalente TGD durch EGD-Rewriting ist Constraint-Rewriting dazu in der Lage, die Auswirkung von r_7 korrekt zu bestimmen. Hierbei wird keinesfalls nur r_7 modifiziert, sondern zahlreiche zusätzliche TGDs und Atome (des Relationssymbols Eq) eingeführt. Der Test findet also tatsächlich auf einer wesentlich umfangreicheren Menge von Integritätsbedingungen statt, als wir hier dargestellt haben. Dies hat selbstverständlich Auswirkungen auf die Laufzeit des Terminierungstests.

Zusammen mit den Testergebnissen gibt die GUI des Terminierungstesters die jeweils benötigte Zeit in Millisekunden aus (siehe Abbildung 5.2, Abbildung 5.3 und Abbildung 5.4). Um verlässliche Ergebnisse zu erhalten, wurden die Tests zehnmal wiederholt und Mittelwert sowie Standardabweichung der Laufzeit bestimmt. Die Ergebnisse dieses Tests sind in Tabelle 5.1 eingetragen. Auffällig ist, dass die drei Varianten der Schwachen Azyklizität fast identische, sehr kurze Laufzeiten aufweisen. Da die Zeitkomplexität der zugrundeliegenden Algorithmen polynomial ist [Spe11], können wir erwarten, dass dieses für sehr kleine Anwendungsfälle erzielte Ergebnis auch auf große Mengen von Integritätsbedingungen übertragbar ist. Constraint-Rewriting scheint auf den ersten Blick ebenfalls zu sehr schnellen Ergebnissen zu kommen. Während einfaches Constraint-Rewriting jedoch etwa zehnmal langsamer als der Test auf Safety ist, kommt die Kombination von Constraint-Rewriting und EGD-Rewriting zum Teil (für Anwendungsfall 1) auf die 500-fache Laufzeit des Safety-Kriteriums. Da die Erweiterung der sehr kleinen zu Grunde liegenden TGD-Menge um weitere TGDs durch die Substitutionslose Simulation derart drastische Auswirkungen auf die Laufzeit hat, können wir annehmen, dass der Test auf Azyklizität auch ohne EGD-Rewriting nur sehr schlecht skaliert. Diese Vermutung wird durch die aus der Literatur [GST11] bekannten exponentiellen Zeitkomplexität des Constraint-Rewritings bestätigt.

Test	AF1	AF2	AF3
RA	$0,4 \pm 0,7$ ms	$0,0 \pm 0,0$ ms	$0,0 \pm 0,0$ ms
WA	$0,1 \pm 0,3$ ms	$0,1 \pm 0,3$ ms	$0,0 \pm 0,0$ ms
SC	$0,4 \pm 0,5$ ms	$0,1 \pm 0,3$ ms	$0,0 \pm 0,0$ ms
AC	$3,0 \pm 4,0$ ms	$6,4 \pm 2,9$ ms	$0,1 \pm 0,3$ ms
SfS	$228,6 \pm 76,5$ ms	$22,1 \pm 3,8$ ms	$68,6 \pm 9,5$ ms

Tabelle 5.1.: Arithmetisches Mittel und Stichproben-Standardabweichung (n=10) der benötigten Zeit für den Test auf Reiche Azyklizität (RA), Schwache Azyklizität (WA), Safety (SC) Azyklizität (AC) und Azyklizität nach Substitutionslose Simulation (SfS). AF1 (Anwendungsfall 1) bezieht sich auf r_1, r_2 und r_3, AF2 (Anwendungsfall 2) bezieht sich auf r_4 und r_5 und AF3 (Anwendungsfall 3) bezieht sich auf r_6 und r_7. Der Laufzeittest erfolgte auf einem Intel(R) Core(TM) i5-7200 CPU @ 2,5 GHz 2,7 GHz, 8 GB RAM, 64-Bit-Betriebssystem (Windows 10 Pro N).

(a)



(b)

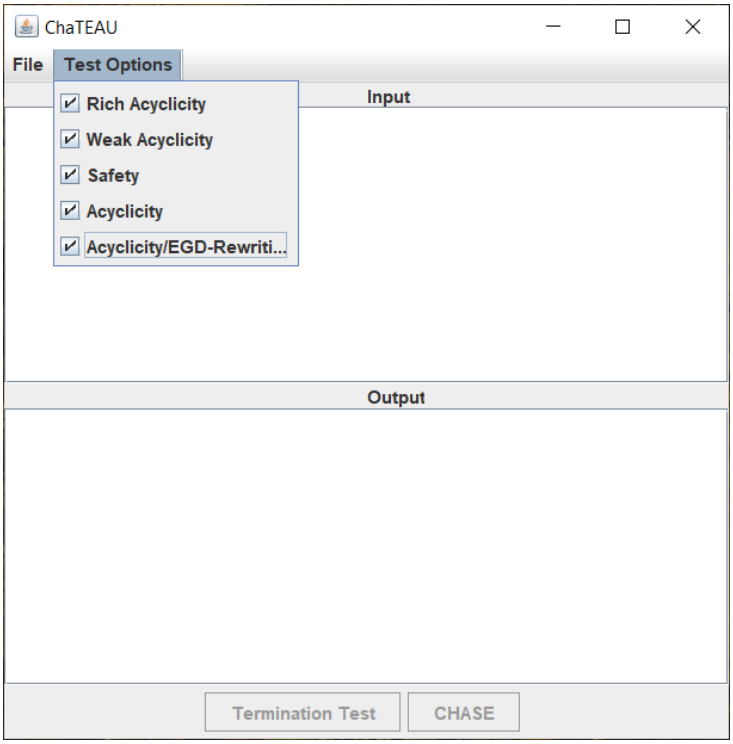


Abbildung 5.1.: Die Graphische Benutzeroberfläche erlaubt es, Integritätsbedingungen aus XML-Dateien zu importieren (a). Über eine Checkbox kann eine beliebige Kombination von Terminierungstests – im vorliegenden Fall alle vorhandenen – ausgewählt werden (b).

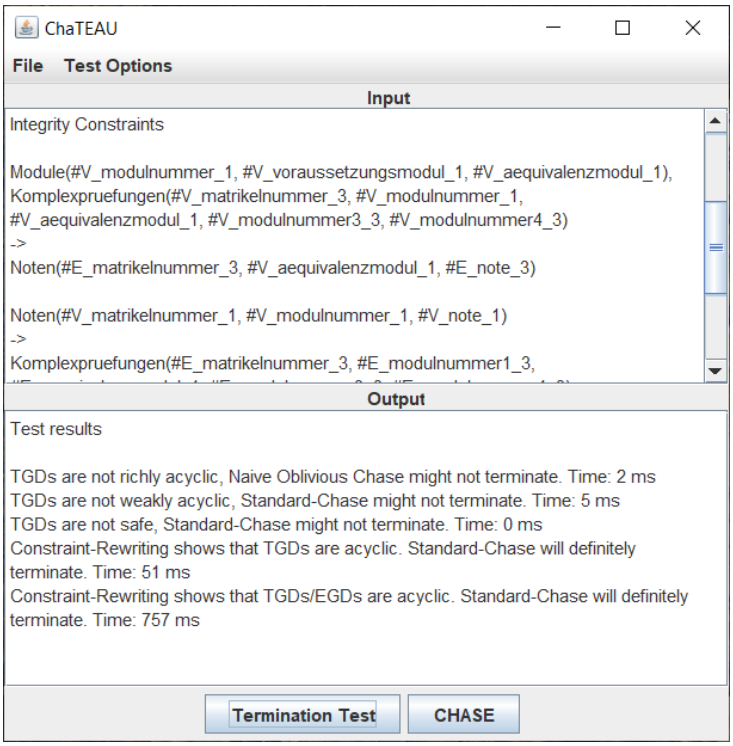


Abbildung 5.2.: Ergebnis von Anwendungsfall 1 (Anwendung der TGDS r_1, r_2, r_3).

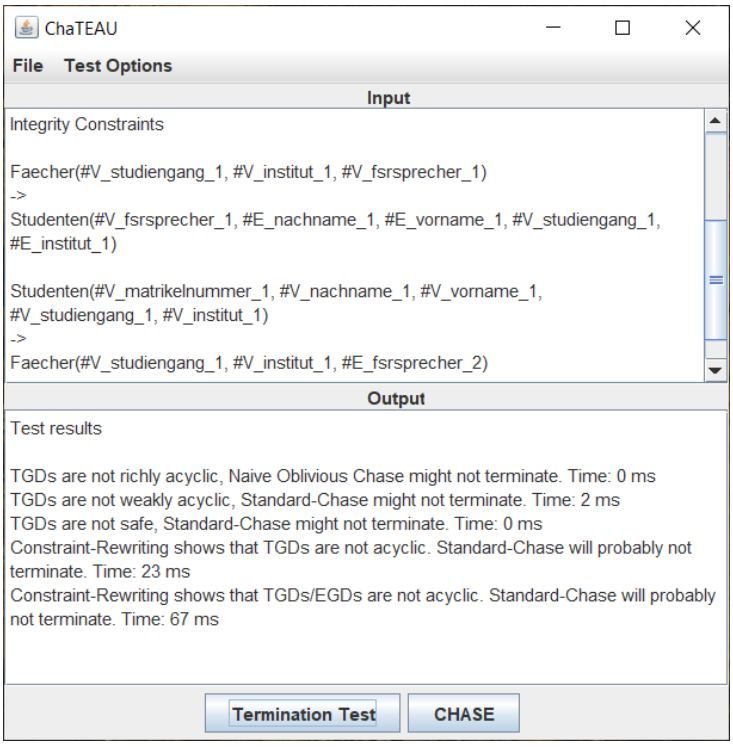


Abbildung 5.3.: Ergebnis von Anwendungsfall 2 (Anwendung der TGDS r_4, r_5).

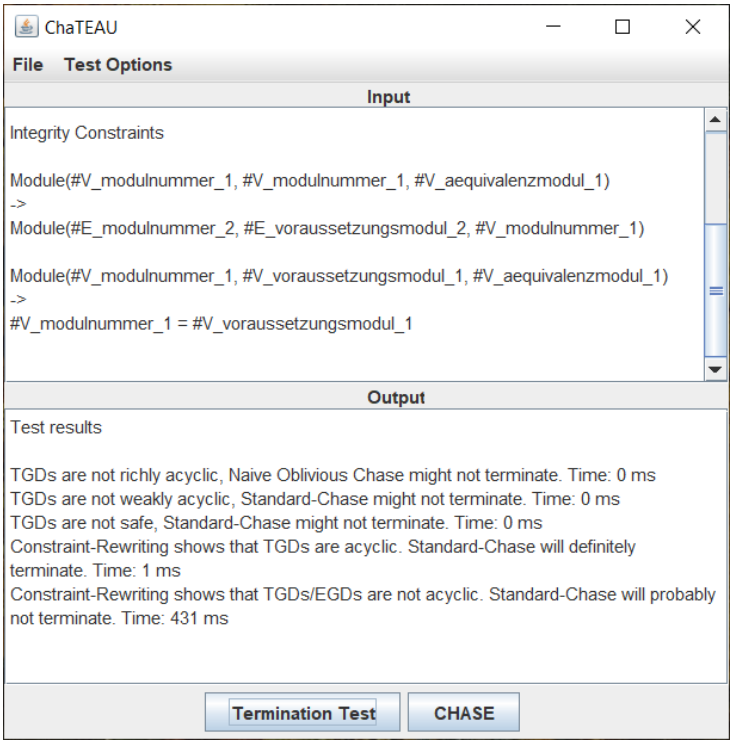


Abbildung 5.4.: Ergebnis von Anwendungsfall 3 (Anwendung der TGD r_6 und EGD r_7).

5.2. Grenzen der Anwendung

Eine grundsätzliche Beschränkung des Terminierungstesters betrifft also die Komplexität der Integritätsbedingungen – sowohl eine zu hohe Anzahl von TGDs als auch eine zu hohe Anzahl von Atomen in den TGDs können die Laufzeit des Constraint-Rewritings derart erhöhen, dass zumindest dieser Terminierungstest in der Praxis untauglich wird. Da EGD-Rewriting durch Substitutionslose Simulation sowohl die Anzahl der TGDs als auch die Anzahl der Atome in den TGDs erhöht, ist eine Kombination von Constraint-Rewriting und EGD-Rewriting selbst bei scheinbar einfachen Anfragen nicht in kurzer Zeit durchführbar. Tatsächlich haben wir kein Kriterium, um zu entscheiden, wann eine Menge von Integritätsbedingungen zu komplex ist, um mit einer Kombination der beiden Rewriting-Verfahren untersuchbar zu sein. Beispiel 3.10 ist ähnlich komplex wie die zuvor evaluierten TGD-Mengen, die Kombination von Constraint-Rewriting mit Substitutionsloser Simulation benötigt jedoch über eine halbe Minute (35718.0 ± 2855.5 ms, n=10).

Um die Abhängigkeit der Laufzeit von der Anzahl der Integritätsbedingungen näher zu untersuchen, betrachten wir folgende einfach strukturierte und induktiv definierbare Datenbank D:

1. R_1 ist in D.
2. Wenn eine Relation R_i in D ist, so hat sie drei Attribute: X ist Primärschlüssel, Y ist Fremdschlüssel zum Primärschlüssel von Relation R_{i1}, Z ist Fremdschlüssel zum Primärschlüssel der Relation R_{i2}, $\mathrm{R}_{i1}, \mathrm{R}_{i2} \in D$.
3. Sonst sind keine Relationen in D – wir werden jedoch für einzelne Relationen Z als Fremdschlüssel zum Primärschlüssel von R_1 definieren, um Zyklizität zu gewährleisten.

Aus Punkt 2 folgen die drei Integritätsbedingungen r_1, r_2 und r_{3a}, während sich aus der Ausnahmeregelung von Punkt 3 r_{3b} anstelle von r_{3a} ableiten lässt:

$r_1 : \mathrm{R}_i(x1, y1, z1), \mathrm{R}_i(x1, y2, z2) \rightarrow y1 = y2, z1 = z2$
$r_2 : \mathrm{R}_i(x1, y1, z1) \rightarrow \exists Y2, Z2 : \mathrm{R}_i1(y1, Y2, Z2)$
$r_{3a} : \mathrm{R}_i(x1, y1, z1) \rightarrow \exists Y2, Z2 : \mathrm{R}_i2(z1, Y2, Z2)$
$r_{3b} : \mathrm{R}_i(x1, y1, z1) \rightarrow \exists Y2, Z2 : \mathrm{R}_1(z1, Y2, Z2)$.

Wir erweitern nun unsere Integritätsbedingungsbasis entsprechend der induktiven Definition der Datenbank iterativ, wobei die zugrundeliegenden Relationen stets einen balancierten Binärbaum bilden. Bei jedem Schritt kommen eine EGD und zwei TGDs hinzu (d.h. für die Tabellen R_{i1} und R_{i2} sind nicht zwingenderweise Primärschlüssel definiert). Wir wenden nach jeder Erweiterung die Kombination von Constraint-Rewriting und EGD-Rewriting an und bestimmen die Laufzeit. In einem zweiten Testdurchlauf bauen wir die Integritätsbedingungsbasis auf gleiche Weise auf, ersetzen jedoch die zuletzt hinzugefügte TGD vom Typ r_{3a} durch eine entsprechende TGD des Typs r_{3b}. Die verwendeten Integritätsbedingungen sind (zum Teil auskommentiert) in der Datei `Beispiel18_Binaerbaum.txt` im Implikationsformat aufgeführt.

Die Analyse zeigt, dass sich die Laufzeit der Untersuchung azyklischen Bedingungen für jede zusätzliche Relation (d.h. drei zusätzlichen Integritätsbedingungen) etwa um den Faktor 10 erhöht (zumindest für 15 oder weniger Integritätsbedingungen, siehe Tabelle 5.2). Die Untersuchung nicht azyklischer Bedingungen ist deutlich zeitaufwendiger als die Analyse azyklischer Bedingungen, aber nur, wenn die Anzahl der Integritätsbedingungen niedrig ist. Wenn hingegen viele (mindestens 15) Bedingungen untersucht werden, gleichen sich die benötigten Laufzeiten einander an, und die Untersuchung nicht azyklischer Integritätsbedingungen ist schlussendlich sogar schneller als die der azyklischen Bedingungen (Abbildung 5.5). Die Standardabweichung ist hierbei jedoch enorm hoch – wir wollen an dieser Stelle daran

Anzahl	Azyklisch	Zyklisch
3	$5,6 \pm 1,0$ ms	$117,7 \pm 19,1$ ms
6	$34,2 \pm 7,8$ ms	$390,2 \pm 81,9$ ms
9	$214,7 \pm 47,0$ ms	$1356,8 \pm 444,4$ ms
12	$2386,0 \pm 254,8$ ms	$5433,1 \pm 1286,5$ ms
15	$13261,7 \pm 590,2$ ms	$11537,4 \pm 5266,3$ ms
18	$54034,5 \pm 913,0$ ms	$28313,2 \pm 13129,3$ ms
21	$173800,6 \pm 8493,1$ ms	$56441,4 \pm 21024,5$ ms
24	$624006,8 \pm 37804,0$ ms	$71345,7 \pm 25326,3$ ms

Tabelle 5.2.: Arithmetisches Mittel und Stichproben-Standardabweichung (n=10) der benötigten Zeit für den Test auf Azyklizität nach Substitutionsloser Simulation in Millisekunden. Die Anzahl bezieht sich auf die Menge untersuchter Integritätsbedingungen, wobei je 3 Bedingungen benötigt werden, um Primär- und Fremdschlüsselbeziehungen einer Relation zu charakterisieren. Azyklisch bezieht sich auf eine azyklische Regelmenge, für die durchgehend r_1, r_2 und r_{3a} verwendet wurden, zyklisch bezieht sich auf eine nicht azyklische Regelmenge, deren letzte TGD durch eine TGD vom Typ r_{3b} ersetzt wurde. Der Laufzeittest erfolgte auf einem Intel(R) Core(TM) i5-7200 CPU @ 2,5 GHz 2,7 GHz, 8 GB RAM, 64-Bit-Betriebssystem (Windows 10 Pro N).

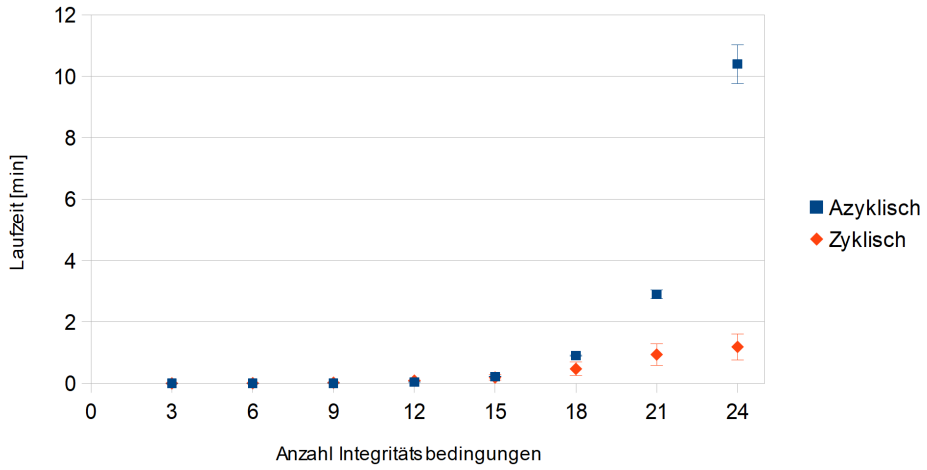


Abbildung 5.5.: Graphische Darstellung der Laufzeiten aus Tabelle 5.2 in Minuten.

erinnern, dass bei jedem Durchlauf die Reihenfolge der eingelesenen Integritätsbedingungen verändert wird (aufgrund der Mengensemantik von Java und ChaTEAU), und bestimmte Reihenfolgen ein schnelleres Finden des Zyklus und somit einen vorzeitigen Abbruch des Algorithmus erlauben. Die Laufzeiten der anderen Terminierungskriterien (inklusive direktem Constraint-Rewriting) sind nicht in Tabelle 5.2 eingetragen, liegen jedoch auch für die maximale Anzahl von Integrationsbedingungen deutlich unter einer Sekunde.

Eine weitere Beschränkung beruht auf einer fundamentalen Eigenschaft der implementierten Terminierungskriterien: Obwohl ChaTEAU den Standard-Chase durchführt, untersuchen die Terminierungstests das Verhalten des Skolem-Oblivious Chase. Dieses Verhalten kann anhand der TGDs der Beispieldatei `Beispiel17_Implication.txt` veranschaulicht werden, die über den Menüpunkt „Open Implication“ der graphische Benutzeroberfläche in den Terminierungstester importiert werden können. Da der Standard-Chase in allen Fällen hält, in denen auch der Skolem-Oblivious Chase terminiert, führt dies nicht zu fehlerhaften Ergebnissen – fehlerhaft bedeutet für das implementierte Werkzeug, dass ein sicheres Anhalten des Chase garantiert wird, obwohl er tatsächlich nicht terminiert. Allerdings folgt hieraus eine Schwächung der Terminierungstests.

5.3. Empfehlung

Der Terminierungstester stellt vier verschiedene einzelne Tests zur Verfügung. Wie wir anhand von r_1, r_2 und r_3 aus Abschnitt 5.1 zeigen konnten, gibt es Fälle, in denen nur Constraint-Rewriting das Terminieren des Chase korrekt vorhersehen kann (was auch unseren theoretischen Kenntnissen entspricht). In Gegenwart von EGDs (r_6 und r_7) liefert Constraint-Rewriting hingegen als einziger der durchgeführten Terminierungstests unter Umständen ein falsches Ergebnis. Die Kombination von Constraint-Rewriting und EGD-Rewriting führt zwar in allen gezeigten Fällen zum korrekten Ergebnis, die Laufzeitanalyse demonstriert jedoch, dass dieses Verfahren wesentlich länger dauert als die übrigen zur Verfügung stehenden Test. Da wir uns in der vorliegenden Arbeit für die „entkoppelte Variante“ der Einbindung des Terminierungstesters (siehe Abbildung 4.5) entschieden haben, lassen wir letztendlich dem Nutzer die Entscheidung, welches Kriterium überprüft werden soll. Der Nutzer soll hierbei durch folgende Empfehlungen unterstützt werden:

- Safety ist stets Schwacher Azyklizität vorzuziehen, da beide Kriterien eine annähernd gleiche Laufzeit aufweisen (siehe Laufzeittest in Abschnitt 5.1). Wird allerdings ein Vergleich verschiedener Terminierungstester angestrebt, bietet sich das Kriterium der Schwache Azyklizität aufgrund seiner häufigen Implementierung (z.B. in Llunatic, Graal und ChaseTEQ) an. Reiche Azyklizität ist aufgrund der fehlenden Implementierung des Naiven Oblivious Chase bedeutungslos für ChaTEAU.
- Wenn mindestens ein Test die Terminierung des Chase garantiert, so sind die Ergebnisse der anderen Tests zu ignorieren.
- Der Test auf Azyklizität sollte zunächst vermieden werden, da er mitunter zu viel Zeit kostet. Reicht das Safety-Kriterium nicht aus, das sichere Terminieren des Chase zu garantieren, kann Azyklizität auch nachträglich getestet werden – die „Termination Test“-Schaltfläche kann beliebig oft (mit jeweils veränderter Auswahl von Terminierungskriterien) betätigt werden.
- Sind EGDs vorhanden, sollte nicht der Test auf Azyklizität verwendet werden, sondern stattdessen die Option „Acyclicity/EGD-Rewriting“ gewählt werden.

Falls in zukünftigen Arbeiten die „pragmatische Variante“ der Einbindung realisiert werden soll (Abbildung 4.4), ist dies sehr einfach durch eine Fallunterscheidung, z.B. im Listener der „CHASE“-Schaltfläche in der Methode `draw()` der Klasse `TerminationTestGUI`, möglich. Vor Durchführung des Chase (d.h. nach Auswahl der Schaltfläche „CHASE“) würde auf jeden Fall der Test des Safety-Kriteriums erfolgen. Falls der Test das Terminieren des Chase garantiert, wird sofort der Chase durchgeführt, andernfalls wird zunächst die Azyklizität der Integritätsbedingungen bestimmt. Allerdings überprüft der Terminierungstester bisher nicht, ob eine übergebene Menge von Integritätsbedingungen EGDs enthält. Für die Implementierung der „pragmatischen Variante“ sollte dies unbedingt getestet werden. Bei Vorhandensein von EGDs sollte Constraint-Rewriting nur nach vorangegangenem EGD-Rewriting durchgeführt werden.

Wir konnten in diesem Kapitel zeigen, dass der von uns konzipierte Terminierungstester die an ihn gestellten Anforderungen erfüllt. Im folgenden Kapitel wollen wir zusammenfassen, was im Rahmen der vorliegenden Arbeit erreicht wurde und welche Anknüpfungspunkte für zukünftige Arbeiten wir sehen.

6. Fazit und Ausblick

Abschließend wollen wir die Vorgaben der Aufgabenstellung (Abschnitt 1.1) mit den erreichten Ergebnissen vergleichen. Die in der Einleitung definierten Ziele wurden durch die Implementierung eines Terminierungstesters für ChaTEAU erfüllt, dennoch ist die Thematik keinesfalls abschließend behandelt worden, und es ist absehbar, dass die Problematik der Chase-Terminierung Ansätze für zukünftige Forschungsarbeiten liefern kann, von denen wir zwei näher beschreiben wollen.

6.1. Fazit

Entsprechend der Aufgabenstellung war der erste Schritt der vorliegenden Arbeit eine Literaturrecherche der bekannten Terminierungskriterien. Hierbei haben wir einen Fokus auf die Arbeiten von Greco et al. gelegt. Bei der Untersuchung weiterer Chase-Werkzeuge (insbesondere Graal) fiel uns jedoch auf, dass TGDs (abgesehen von den besprochenen Terminierungsklassen) auf Basis syntaktischer Eigenschaften in weitere Klassen eingeteilt werden können – auf diese wurde jedoch nicht näher eingegangen. Die Terminierungskriterien des Chase wurden anhand selbst gewählter Beispiele veranschaulicht. Tatsächlich handelt es sich bei den gezeigten Beispielen lediglich um Transformationen abstrakter Beispiele aus [GMS12] – obwohl die Beispiele also auf einer realistischen Datenbank (d.h. einer Datenbank mit praxisnahen Relations- und Attributnamen) basieren, ist die Semantik der TGDs nicht an der Praxis orientiert.

Die vorgestellten statischen Terminierungsklassen[1] sind zwar nicht durchgängig hierarchisch aufgebaut (d.h. es gibt nicht vergleichbare Testkriterien), die Klasse der Azyklizität bildet jedoch eine Obermengen aller anderen statischen Terminierungsklassen. Da der Test auf Azyklizität gleichzeitig relativ einfach umsetzbar war und darüber hinaus auch Ansätze für zukünftige Erweiterungen bietet, wurde dieser Test als zentrales Element des Terminierungstesters implementiert.

Darüber hinaus wurden drei weitere Testkriterien umgesetzt: Schwache Azyklizität, Reiche Azyklizität und Safety. Da außerdem der Algorithmus der Substitutionslosen Simulation implementiert wurde, kann auch in Gegenwart von EGDs der Test auf Azyklizität durchgeführt werden. Mit dieser Kombination von Constraint-Rewriting und EGD-Rewriting gehen wir über die Funktionalität von ChaseTEQ, der unseres Wissens nach einzigen anderen Implementierung eines Adn-Algorithmus, hinaus.

Obwohl ChaTEAU in Zukunft auch st-TGDs verarbeiten kann, wurden diese im Terminierungstester nicht gesondert untersucht. Tatsächlich haben st-TGDs auf Basis der in der vorliegenden Arbeit gemachten Annahmen ohnehin keinen Einfluss auf die Terminierung des Chase. Die Erweiterungen des Chase durch bestimmte Aggregatfunktionen (siehe Unterabschnitt 6.2.2) könnte dazu führen, dass auch st-TGDs vom Terminierungstester berücksichtigt werden müssen. Da ChaTEAU st-TGDs intern wie TGDs behandelt (und lediglich testet, dass Körper und Kopf der TGD ausschließlich über der Quell- bzw. Zieldatenbank definiert sind), werden hierfür aber keine weiteren Anpassungen des Terminierungstesters benötigt.

[1] Statische Terminierungsklassen sind Mengen von TGD-Mengen, für die der Chase gemäß eines bestimmten statischen Terminierungskriteriums hält.

Der implementierte Terminierungstester sagt – bei Auswahl des geeigneten Terminierungskriteriums – für alle in dieser Arbeit vorgestellten Beispiele die Terminierung des Chase korrekt voraus. Es hat sich jedoch gezeigt, dass der Test auf Azyklizität nicht gut skaliert, und insbesondere eine Kombination dieses Testes mit Substitutionsloser Simulation schon bei relativ kleinen Datenbanken zu enormen Laufzeiten führen kann. Es bietet sich also in der Praxis an, auf diese Kombination zunächst zu verzichten. Mit hinreichendem Sachverstand könnte der Anwender EGDs ignorieren (d.h. auf EGD-Rewriting verzichten), wenn diese offensichtlich keine Auswirkungen auf die Chase-Terminierung haben. Beispielsweise ist dies für die in Abschnitt 5.2 untersuchten funktionalen Abhängigkeiten der Fall. Allerdings wäre für ein derart einfach strukturiertes Beispiel ohnehin der (schnell durchführbare) Test auf Schwache Azyklizität ausreichend gewesen, um die Terminierung des Chase zu beurteilen (und im Fall der Nicht-Terminierung des Chase hätte einfaches Constraint-Rewriting dies auch ohne EGD-Rewriting vorhergesehen). Es ist zu vermuten, dass für eine Vielzahl praktischer Anwendungsfälle die schnellen Terminierungstests ausreichend sind. Mit dem hier vorgestellten Terminierungstester stellen wir ein Werkzeug bereits, mit dem diese Hypothese überprüft werden kann.

Abschließend geben wir einen Ausblick auf mögliche Erweiterungen des Terminierungstesters. Da die Entwicklung von ChaTEAU noch nicht abgeschlossen ist, sind Anpassungen des Terminierungstesters an neue Möglichkeiten des Chase-Tools notwendig.

6.2. Vorschläge für Erweiterungen

Wie zuvor bereits erwähnt, ähnelt die Methode des Constraint-Rewritings den semi-dynamischen Kriterien (Unterabschnitt 3.1.2) und bietet auf dieser Basis viel Spielraum für Erweiterungen. Verglichen mit existierenden Implementierungen des Chase (z.B. in Llunatic) ist die Implementierung von Adn++ beispielsweise relativ ineffizient – man könnte in zukünftigen Arbeiten also Ideen der Effizienzsteigerung aus der Chase-Entwicklung für Verbesserungen von Adn++ nutzen.

Im Vergleich zu anderen semi-dynamischen Kriterien zeigen sich Schwächen des Adn++ hinsichtlich der Behandlung von Konstanten. Konstanten tragen stets das Adornment b, welches für sich alleine noch keine Rückschlüsse auf den Wert der Konstante zulässt. Wird dieses Adornment nun (in mehreren Schritten) weitergegeben, kann eine mehrfach im Körper einer TGD vorkommende Variable b-Adornments unterschiedlicher Konstanten tragen, ohne dass dies zu Inkonsistenzen führt. Es würde sich also anbieten, b-Adornments Attribute für Wert und Typ einer Konstanten (bzw. ein Attribut für die Konstante selbst) zu verleihen.

Neben diesen Möglichkeiten, den Terminierungstester zu optimieren, wird in Zukunft die Notwendigkeit bestehen, ChaTEAUs Terminierungstest-Modul an die Weiterentwicklung der anderen Module von ChaTEAU anzupassen. In den abschließenden Unterkapiteln wollen wir anhand zweier beispielhafter Erweiterungen zeigen, dass dies ist für die implementierten Terminierungskriterien der Schwachen Azyklizität (bzw. Safety) und der Azyklizität durchaus möglich ist.

6.2.1. Vergleichsoperatoren

In der vorliegenden Arbeit haben wir bereits eine – implizite – Vergleichsoperation im Körper von Integritätsbedingungen zugelassen, und zwar den Test auf Gleichheit. Dieser wurde allerdings durch das mehrfache Vorkommen derselben Variable (bzw. die Ersetzung einer Variablen durch die gleichzusetzende Konstante) und nicht durch ein explizites Vergleichsprädikat dargestellt. Nur im Zuge des EGD-Rewritings (Substitutionslose Simulation) wurde diese implizite Darstellung durch das explizite (relationale) Atom

Eq ersetzt. Im Folgenden wollen wir untersuchen, wie sich weitere Vergleichsoperationen auf die Terminierungskriterien auswirken, die im Zuge der vorliegenden Arbeit implementiert wurden. Sei hierfür Σ eine Menge von Integritätsbedingungen der Form

$$\phi(\mathbf{x}, \mathbf{y}) \wedge (\mathbf{a} \circ \mathbf{b}) \rightarrow \exists \mathbf{z} : \psi(\mathbf{y}, \mathbf{z}),$$

wobei $\mathbf{a} \circ \mathbf{b}$ für eine Konjunktion von Ausdrücken der Form $a_i \circ b_i$ mit $a_i \in \mathbf{a}, b_i \in \mathbf{b}, \circ \in \{<, \leq, >, \geq, \neq\}$ steht und $\mathbf{a} \in (\mathbf{x} \cup \mathbf{y}), \mathbf{b} \in (\mathbf{x} \cup \mathbf{y})$ gilt.

Die Kriterien der Schwachen Azyklizität, Reichen Azyklizität und der Safety können an diese Erweiterung zwar nicht angepasst werden, müssen es aber auch nicht. Im Gegensatz z.B. zum Beispiel zum Kriterium der Stratifikation überprüfen diese drei Kriterien nicht, ob eine TGD durch eine andere TGD getriggert wird (oder überhaupt jemals getriggert werden kann). Wir könnten zwar überprüfen, ob a_i oder b_i in $a_i < b_i$ affected sind – dies bedeutet jedoch lediglich, dass a oder b_i einen Nullwert tragen können. Wenn dieser Nullwert zum Nicht-Terminieren des Chase führt, müssen wir davon ausgehen, dass dieser Nullwert tatsächlich existiert, im vorliegenden Fall würde ein Nullwert jedoch das Triggern der TGD verhindern (da der $<$-Operator nicht für Nullwerte definiert ist) und so möglicherweise das Terminieren des Chase erst herbeiführen. Wir müssen in diesem Fall also davon ausgehen, dass auch affected Positionen keinen Nullwert tragen. Wenn wir die zusätzlichen Vergleichsatome im Körper der TGDs ignorieren, kann dies allerdings nicht zu einem falschen Ergebnis führen – ein Chase, dessen Terminierung garantiert ist, kann nicht in eine Endlosschleife geraten, indem für das erfolgreiche Triggern einer TGD mehr Bedingungen gestellt werden. wir können also davon ausgehen, dass das Ergebnis der zusätzlichen Vergleichsprädikate stets *true* ist, denn dies wäre der ungünstigste Fall hinsichtlich der Nicht-Terminierung des Chase.

Wie zuvor bereits erwähnt, zeichnet sich der Adn++-Algorithmus durch sein enormes Erweiterungspotential aus. Wir können hier tatsächlich das Ergebnis des Vergleichsprädikats – auf Basis des Adornments und des Wertes von Konstanten – überprüfen. Es gilt hierfür ($\circ^* \in \{<, \leq, >, \geq\}$):
Konstante $\circ$ Konstante = (Ergebnis des Vergleichs der Konstanten),
$b \circ b = true$,
$b \neq f_j = true$,
$f_i \neq b = true$,
$f_i \neq f_j =$ (Ergebnis des Vergleichs der f-Adornments),
$b \circ^* f_j = false$,
$f_i \circ^* b = false$,
$f_i \circ^* f_j = false$.
Ist der Wert eines Vergleichsatoms $false$, so ist ist der Körper der Integritätsbedingung inkonsistent (wobei wir den Test Konstante$\circ$Konstante selbstverständlich bereits vor dem Test auf Konsistenz durchführen können).

6.2.2. Aggregatfunktionen

Obwohl es sehr wohl Aggregatfunktionen gibt, welche für die Terminierung des Chase wahrscheinlich ohne Bedeutung sind (z.B. Minimum oder Maximum), verhalten sich viele Aggregatfunktionen einem Nullwert vergleichbar. Wir gehen im Folgenden von der nicht sehr praxisnahen Modellvorstellung aus, dass Aggregatfunktionen als Term eines TGD-Kopfes verwendet werden.

Sei hierfür Σ eine Menge von Integritätsbedingungen der Form

$$\phi(\mathbf{x}, \mathbf{y}) \rightarrow \exists \mathbf{z} : \psi(\mathbf{y}, \mathbf{z}, \mathrm{Agg}(\lambda, A, P)),$$

wobei Agg ein Vektor aus Aggregatfunktionen darstellt, die von den Attributen P abhängen. P ist eine Teilmenge der Menge aller Attribute A der Relationen λ. Beispielsweise könnte durch die `Average`-Aggregatfunktion die Durchschnittsnote der Relation Noten bestimmt werden. Für diesen Fall gilt also: $\text{Agg} = (\text{Average}), \lambda = \text{Noten}, A = \{\text{Matrikelnummer}, \text{Modulnummer}, \text{Note}\}, P = \{\text{Note}\}$.

Das Kriterium der Schwachen Azyklizität kann angepasst werden, indem zusätzliche besondere Kanten eingeführt werden: Das Ergebnis der Aggregatfunktion ändert sich nur, wenn P neue Werte entstehen, also wenn in λ neue Tupel eingefügt werden. Wenn $\lambda \not\subset \phi$ gilt, so ist der Wert der Aggregatfunktion unabhängig von Veränderungen in diesen Relationen – im Gegensatz zu Nullwerten bedeutet für Aggregatfunktionswerte die Generierung eines neuen Tupels also nicht automatisch die Generierung eines neuen Wertes. Abgesehen hiervon verhalten sich beide jedoch ähnlich – potentiell neue Tupel werden nur dann (in einer Endlosschleife des Chase) tatsächlich erzeugt, wenn Werte, die zuvor noch nicht in der Datenbank vorkommen konnten, in das neue Tupel übertragen werden. Das können bereits erzeugte Nullwerte oder Aggregatfunktionswerte sein. Wenn neue Werte in P zur Generierung eines neuen Tupels führen, werden für alle existenzquantifizierten Variablen neue Nullwerte erzeugt. Wir müssen also von den Positionen (λ, A) besondere Kanten zu den Positionen von $\mathbf{z}$ und Agg in ψ ziehen. Für bestimmte Aggregatfunktionen wäre es sinnvoll, nicht A sondern $A - P$ zu verwenden – ein neuer Nullwert im Attribut Matrikelnummer (um beim Beispiel der Durchschnittsnote zu bleiben) geht zwar mit einem neuem Tupel der Notentabelle und daher einem neuen Eintrag im Attribut Note einher, ist dieser Eintrag jedoch ebenfalls ein Nullwert, wird sich das Ergebnis des Notendurchschnitts nicht verändern (abhängig von der Spezifikation der Aggregatfunktion). Allerdings setzen wir hier voraus, dass ein Nullwert nach P übertragen wurde – wurde stattdessen ein Aggregatfunktionswert übertragen, müssen wir A verwenden und nicht $A - P$. Wenn die Durchschnittsnote der Notentabelle als Eintrag des Attributs Note gesetzt wird, hat dies selbstverständlich einen Einfluss auf den Wert der Durchschnittsnote. Darüber hinaus gibt es auch Aggregatfunktionen (z.B. Count), die nicht von einem spezifischen Attribut abhängen, für die P also die leere Menge darstellt.

Tatsächlich haben wir bei obigen Beobachtungen jedoch übersehen, dass ein Trigger nicht nur aktiviert werden, sondern überhaupt erst erzeugt werden muss. Der Körper der TGD könnte zum Beispiel nur allquantifizierte Variablen enthalten, die nicht im Kopf vorkommen – wir könnten aufgrund des Kriteriums der Schwachen Azyklizität alleine also überhaupt keine Kanten ziehen. Um einen (womöglich inaktiven) Trigger zu berücksichtigen, ziehen wir zunächst die speziellen Kanten des Kriteriums der Reichen Azyklizität (Typ 2-Kanten), d.h. besondere Kanten, die von allen Positionen allquantifizierter Variablen im Körper zu den Positionen aller existenzquantifizierten Variablen und Aggregatfunktionen im Kopf führen. Anschließend ziehen wir eine andere Art besonderer Kanten (Typ 3-Kanten), die von allen Positionen (λ, A) zu den Positionen aller existenzquantifizierten Variablen und Aggregatfunktionen im Kopf führen. Wenn nur einer dieser beiden Kantentypen gezogen werden könnte, werden keine Kanten gezogen (d.h. wenn keine Aggregatfunktion vorkommt, werden auch keine Typ 2-Kanten gezogen). Der Algorithmus basiert zwar auf dem Kriterium der Schwachen Azyklizität (d.h. Typ-2 und Typ-3-Kanten werden zusätzlich zu den normalen (Typ 0-Kante) und besonderen Kanten (Typ 1-Kante) dieses Kriteriums gezogen), könnte aber für das Safety-Kriterium erweitert werden, indem Aggregatfunktionen wie existenzquantifizierte Variablen verwendet werden, um die Affectedness einer Position zu bestimmen. Wenn letztendlich sowohl ein Zyklus durch Typ-2-Kanten (Trigger generierende Kanten) als auch ein Zyklus durch Typ-3-Kanten (Trigger aktivierende Kanten) gefunden werden kann, sind die Integritätsbedingungen nicht schwach azyklisch unter Berücksichtigung von Aggregatfunktionen und der Chase terminiert wahrscheinlich nicht. Wenn Zyklen durch mindestens eine Typ 1-Kante und zusätzlich eventuell Typ 0-Kanten (aber keine anderen Kantentypen) führen, sind die Integritätsbedingungen schwach azyklisch auch ohne Berücksichtigung von Aggregatfunktionen. Andernfalls ist die Terminierung des Chase garantiert.

Der Adn++-Algorithmus lässt sich nur mit einigem Aufwand an Aggregatfunktionen anpassen. Aggregatfunktionen sollten Adornments erhalten, die sich vom f-Adornment existenzquantifizierter Variablen ableiten. Wie diese werden die zusätzlichen Adornments – nennen wir sie an dieser Stelle a-Adornments – im Rahmen des Adn++-Algorithmus regelmäßig neu bestimmt und auf Atome in Körpern anderer TGDs übertragen. Man könnte zum einen in a-Adornments (korrespondierend zum Wert der Aggregatfunktion) sämtlich bisher erzeugten adornten Prädikate mit Relationssymbol aus λ (korrespondierend zum Wert aller Tupel, die zum Wert der Aggregatfunktion beitragen) speichern. Zum anderen müsste ebenfalls die Skolemisierung der f-Adornments angepasst werden – diese sollten nicht mehr nur vom Adornment allquantifizierter Variablen des TGD-Kopfes abhängen, sondern auch vom Adornment aller Aggregatfunktionen des TGD-Kopfes.

Literaturverzeichnis

[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

[BLM+15] Baget, Jean-François ; Leclère, Michel ; Mugnier, Marie-Laure ; Rocher, Swan ; Sipieter, Clément: Graal: A Toolkit for Query Answering with Existential Rules. In: *Rule Technologies: Foundations, Tools, and Applications - 9th International Symposium, RuleML 2015, Berlin, Germany, August 2-5, 2015, Proceedings*, 2015, 328–344

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

[CGMT16] Calautti, Marco ; Greco, Sergio ; Molinaro, Cristian ; Trubitsyna, Irina: Exploiting Equality Generating Dependencies in Checking Chase Termination. In: *PVLDB* 9 (2016), Nr. 5, 396–407. `http://dx.doi.org/10.14778/2876473.2876475`. – DOI 10.14778/2876473.2876475

[CGP15] Calautti, Marco ; Gottlob, Georg ; Pieris, Andreas: Chase Termination for Guarded Existential Rules. In: *Proceedings of the 9th Alberto Mendelzon International Workshop on Foundations of Data Management, Lima, Peru, May 6 - 8, 2015*, 2015

[CP19] Calautti, Marco ; Pieris, Andreas: Oblivious Chase Termination: The Sticky Case. In: *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal*, 2019, 17:1–17:18

[DNR08] Deutsch, Alin ; Nash, Alan ; Remmel, Jeffrey B.: The chase revisited. In: *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008, June 9-11, 2008, Vancouver, BC, Canada*, 2008, 149–158

[DT03] Deutsch, Alin ; Tannen, Val: Reformulation of XML Queries and Constraints. In: *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, 2003, 225–241

[FGST11] Francesco, Andrea D. ; Greco, Sergio ; Spezzano, Francesca ; Trubitsyna, Irina: ChaseT: A Tool for Checking Chase Termination. In: *Scalable Uncertainty Management - 5th International Conference, SUM 2011, Dayton, OH, USA, October 10-13, 2011. Proceedings*, 2011, 520–524

[FKMP03] Fagin, Ronald ; Kolaitis, Phokion G. ; Miller, Renée J. ; Popa, Lucian: Data Exchange: Semantics and Query Answering. In: *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, 2003, 207–224

[FKMP05] Fagin, Ronald ; Kolaitis, Phokion G. ; Miller, Renée J. ; Popa, Lucian: Data exchange: semantics and query answering. In: *Theor. Comput. Sci.* 336 (2005), Nr. 1, 89–124. http://dx.doi.org/10.1016/j.tcs.2004.10.033. – DOI 10.1016/j.tcs.2004.10.033

[GHK+13] Grau, Bernardo C. ; Horrocks, Ian ; Krötzsch, Markus ; Kupke, Clemens ; Magka, Despoina ; Motik, Boris ; Wang, Zhe: Acyclicity Notions for Existential Rules and Their Application to Query Answering in Ontologies. In: *J. Artif. Intell. Res.* 47 (2013), 741–808. http://dx.doi.org/10.1613/jair.3949. – DOI 10.1613/jair.3949

[GMPS13] Geerts, Floris ; Mecca, Giansalvatore ; Papotti, Paolo ; Santoro, Donatello: The LLUNATIC Data-Cleaning Framework. In: *PVLDB* 6 (2013), Nr. 9, 625–636. http://dx.doi.org/10.14778/2536360.2536363. – DOI 10.14778/2536360.2536363

[GMPS14] Geerts, Floris ; Mecca, Giansalvatore ; Papotti, Paolo ; Santoro, Donatello: An Overview of the Llunatic System. In: *22nd Italian Symposium on Advanced Database Systems, SEBD 2014, Sorrento Coast, Italy, June 16-18, 2014*, 2014, S. 159–166

[GMS12] Greco, Sergio ; Molinaro, Cristian ; Spezzano, Francesca: *Incomplete Data and Data Dependencies in Relational Databases.* Morgan & Claypool Publishers, 2012 (Synthesis Lectures on Data Management). http://dx.doi.org/10.2200/S00435ED1V01Y201207DTM029. http://dx.doi.org/10.2200/S00435ED1V01Y201207DTM029

[GO18] Grahne, Gösta ; Onet, Adrian: Anatomy of the Chase. In: *Fundam. Inform.* 157 (2018), Nr. 3, 221–270. http://dx.doi.org/10.3233/FI-2018-1627. – DOI 10.3233/FI–2018–1627

[Gra] GraphIK team: *Getting Started.* http://graphik-team.github.io/graal/doc/getting-started. – Letzer Zugriff: 24.02.2020

[GST11] Greco, Sergio ; Spezzano, Francesca ; Trubitsyna, Irina: Stratification Criteria and Rewriting Techniques for Checking Chase Termination. In: *PVLDB* 4 (2011), Nr. 11, 1158–1168. http://www.vldb.org/pvldb/vol4/p1158-greco.pdf

[HS07] Hernich, André ; Schweikardt, Nicole: CWA-solutions for data exchange settings with target dependencies. In: *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, 2007, 113–122

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

[Mar09] Marnette, Bruno: Generalized schema-mappings: from termination to tractability. In: *Proceedings of the Twenty-Eigth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2009, June 19 - July 1, 2009, Providence, Rhode Island, USA*, 2009, 13–22

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

[MSL09a] Meier, Michael ; Schmidt, Michael ; Lausen, Georg: On Chase Termination Beyond Stratification. In: *PVLDB* 2 (2009), Nr. 1, 970–981. http://dx.doi.org/10.14778/1687627.1687737. – DOI 10.14778/1687627.1687737

[MSL09b] Meier, Michael ; Schmidt, Michael ; Lausen, Georg: Stop the Chase: Short Contribution. In: *Proceedings of the 3rd Alberto Mendelzon International Workshop on Foundations of Data Management, Arequipa, Peru, May 12-15, 2009*, 2009

[Ren19] Renn, Fabian: *Erweiterung des CHASE-Werkzeugs ChaTEAU um Anfragetransformationen*, Universität Rostock, Bachelorthesis, 2019

[Roc16] Rocher, Swan: *Querying Existential Rule Knowledge Bases: Decidability and Complexity*, Université de Montpellier, Diss., 2016

[San14] Santoro, Donatello: *Advanced Techniques for Mapping and Cleaning*, Roma Tre University, Diss., 2014

[SG10] Spezzano, Francesca ; Greco, Sergio: Chase Termination: A Constraints Rewriting Approach. In: *PVLDB* 3 (2010), Nr. 1, 93–104. `http://dx.doi.org/10.14778/1920841.1920858`. – DOI 10.14778/1920841.1920858

[Spe11] Spezzano, Francesca: *On the Problem of Checking Chase Termination*, Università della Calabria, Diss., 2011

[Wik] Wikipedia: *Zyklus (Graphentheorie) — Wikipedia, Die freie Enzyklopädie*. `https://de.wikipedia.org/wiki/Zyklus_(Graphentheorie)`. – Letzer Zugriff: 24.02.2020

Programmcodeverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Symbolverzeichnis

A	Menge der Attribute
A_i	Attribut
$\lambda(\mathbf{x})$	Konjunktion relationaler Atome
B_r	Körper von r
$\phi(\mathbf{x}, \mathbf{z})$	Körpers einer Integritätsbedingung
H_r	Kopf von r
$\psi(\mathbf{x}, \mathbf{y})$	Kopf einer TGD
$\rho(\mathbf{x})$	Konjunktion relationaler Atome
KONST	Menge der Konstanten
$\mathbb{N}$	Menge der natürlichen Zahlen
$\mathbb{N}^+$	Menge der natürlichen Zahlen ohne 0
η_i	(Markierter) Nullwert
NULL	Menge der (markierten) Nullwerte
R_i	Position
R	Relationenname
$R(U_i)$	Relationenschema
$R(A_1, ..., A_n)$	Relationenschema
R	Menge der Relationennamen
Σ	Menge von Integritätsbedingungen
VAR	Menge der Variablen
$\mathrm{R}^{\alpha_1 \cdots \alpha_n}$	Adornments eines n-stelligen relationalen Atoms R
R^{α}	Adornments eines relationalen Atoms R
$\mathrm{dom}(A_i)$	Attributdomäne
$\Pi_x(B_r)$	Menge der Plätze aus B_r, in denen x vorkommt
$\bowtie$	Natürlicher Verbund
$G(\Sigma)$	Chase-Graph von Σ
I_i	Datenbankinstanz
$\mathbf{R}$	Datenbankschema
$\mathrm{Sol}(D, \Sigma)$	Menge aller Lösungen über Σ
$\mathrm{dep}(\Sigma)$	Abhängigkeitsgraph von Σ

D	Datenbankinstanz
$\emptyset$	Leere Menge
$D \models r_i$	D erfüllt r_i
$\Sigma \models r_i$	Σ impliziert r_i
E	Graph
h	Homomorphismus
$\mathrm{In}(r, x)$	Menge der Input-Plätze
$\mathrm{Out}(r, x)$	Menge der Output-Plätze
$\mathrm{pos}(\Sigma)$	Menge der Positionen aus Σ
$\mathrm{aff}(r, P)$	Menge der affected Positionen aus P (SR)
$\mathrm{aff}(\Sigma)$	Menge der affected Positionen aus Σ (SC)
$r_1 <_c r_2$	Präzedenzrelation der c-Stratifizierung
$q' < q$	Feuern von Plätzen
$r_1 <_P r_2$	Präzedenzrelation der Safe Restriction
$r_1 < r_2$	Präzedenzrelation der Stratifizierung
$\mathrm{prop}(\Sigma)$	Propagationsgraph von Σ
r_i	Integrationsbedingung
r	Relation
$\cap$	Schnitt
$\mathrm{sk}(\Sigma)$	Menge der skolemisierten Bedingungen aus Σ
θ	Substitution
$t[A_i]$	Projektion von t auf das Attribut A_i
$t[X]$	Projektion von t auf die Attributmenge X
$R(t)$	Tupel einer Relation R
τ	Integritätsbedingung
$\Gamma(\Sigma)$	Triggergraph von Σ (SwA)
$\Delta(\Sigma)$	Triggergraph von Σ (LS)
$r_1 \hookrightarrow r_2$	Auslösen (LS)
$r_1 \rightsquigarrow r_2$	Triggern (SwA)
t	Tupel
$Q \sqsubseteq Q'$	Teilmenge unter Unifizierbarkeit
$(a, i) \sim (a', i)$	Unifizierbarkeit
x, y	Variable
$\mathbf{v}$	Vektor aus Variablen
$\vec{v}$	Vektor aus Variablen
$\mathbf{v}_i$	Teilvektor aus Komponenten des Vektors $\mathbf{v}$

x_i	Komponente eines Vektors aus Variablen
$\cup$	Vereinigung

Abkürzungsverzeichnis

AC	Azyklizitätskriterium
CStr	Korrigierte Stratifizierung
EDG	Equality Generating Dependency
FD	Funktionale Abhängigkeit
ID	Identifikationsnummer
IR	Induktive Restriktion
JD	Verbundabhängigkeit
LS	Lokale Stratifizierung
MFA	Model-Faithful Acyclicity
MSA	Model-Summarising Acyclicity
RA	Reiche Azyklizität
SC	Safety-Kriterium
SfS	Substitutionslose Simulation
SR	Safe Restriction
st-TGD	Source-to-Target Tuple-Generating Dependency
Str	Stratifizierung
SwA	Superschwache Azyklizität
TGD	Tuple Generating Dependency
WA	Schwache Azyklizität

A. Anhang

A.1. Studentendatenbank

Die in der Arbeit verwendeten Beispiele basieren auf einer Datenbank mit folgendem Schema:

Studenten($Matrikelnummer, Nachname, Vorname, Studiengang, Institut$)
Fächer($Studiengang, Institut, FSR\_Sprecher$)
Noten($Matrikelnummer, Modulnummer, Note$)
Noten_Informatik($Nachname, Modulnummer, Note, Institut$)
Module($Modulnummer, Voraussetzungsmodul, Äquivalenzmodul$)
Komplexprüfungen($Matrikelnummer, Modulnummer1, Modulnummer2, Modulnummer3$, $Modulnummer4$).

A.2. Relationennamen

Die sechs Relationen der Studentendatenbank wurden aus Platzgründen meist abgekürzt. Diese Notwendigkeit bestand einerseits bei der Definition von Integritätsbedingungen und andererseits bei der Benennung von Positionen.

Relation	Abkürzung
Studenten	St
Fächer	Fä
Noten	No
Noten_Informatik	Ni
Module	Mo
Komplexprüfungen	Ko

Tabelle A.1.: Relationennamen wurden durch einen großgeschriebenen und einen kleingeschriebenen Buchstaben (in der Regel die ersten beiden Buchstaben des Namens) abgekürzt.

A.3. Variablen

In der Studentendatenbank sind insgesamt fünfzehn verschiedene Attributnamen vorhanden, von denen einige in mehreren Relationen vorkommen. Variablen werden bei ihrem ersten Vorkommen nach dem Attribut benannt, in dem sie stehen. Selbstverständlich kann diese Variable anschließend in einem anders benannten Attribut vorkommen, z.B. um eine Inklusionsabhängigkeit zu definieren. Der Typ der Attribute ist für die vorliegende Arbeit nicht von Bedeutung, lässt sich aber in der Regel aus ihrem Namen herleiten.

Attribut	Variable	Erklärung
Matrikelnummer	*ma*	
Nachname	*na*	
Vorname	*vo*	
Studiengang	*st*	
Institut	*in*	
Studiengang	*st*	
FSR_Sprecher	*fs*	Matrikelnummer des Fachschaftsrat-Sprechers
Modulnummer	*mo*	
Note	*no*	
Voraussetzungsmodul	*vo*	Modulnummer des Voraussetzungsmoduls
Äquivalenzmodul	*äq*	Modulnummer des Äquivalenzmoduls
*Modulnummer*1	*mo*1	Modulnummer des ersten geprüften Moduls
*Modulnummer*2	*mo*2	Modulnummer des zweiten geprüften Moduls
*Modulnummer*3	*mo*3	Modulnummer des dritten geprüften Moduls
*Modulnummer*4	*mo*4	Modulnummer des vierten geprüften Moduls

Tabelle A.2.: Variablen für den Wert eines Attributs leiten sich von den ersten beiden Buchstaben des Attributnamens her. Allquantifizierte Variablen wurden kleingeschrieben (hier dargestellt) und extistenzquantifizierte Variablen großgeschrieben. Zahlen, die Teil des Variablennamens sind (z.B. *mo*1), werden nicht tiefgestellt.

A.4. Positionen

Bei der Definition der Schwachen Azyklizität und einiger hiervon abgleiteter Kriterien, wie Reiche Azyklizität und Safety, wird das Konzept der Position verwendet.

Position	(Relation, Attribut)
St_1	(Studenten, *Matrikelnummer*)
St_2	(Studenten, *Nachname*)
St_3	(Studenten, *Vorname*)
St_4	(Studenten, *Studiengang*)
St_5	(Studenten, *Institut*)
$Fä_1$	(Fächer, *Studiengang*)
$Fä_2$	(Fächer, *Institut*)
$Fä_3$	(Fächer, *FSR_Sprecher*)
No_1	(Noten, *Matrikelnummer*)
No_2	(Noten, *Modulnummer*)
No_3	(Noten, *Note*)
Ni_1	(Noten_Informatik, *Nachname*)
Ni_2	(Noten_Informatik, *Modulnummer*)
Ni_3	(Noten_Informatik, *Note*)
Ni_4	(Noten_Informatik, *Institut*)
Mo_1	(Module, *Modulnummer*)
Mo_2	(Module, *Voraussetzungsmodul*)
Mo_3	(Module, *Äquivalenzmodul*)
Ko_1	(Komplexprüfungen, *Matrikelnummer*)
Ko_2	(Komplexprüfungen, *Modulnummer*1)
Ko_3	(Komplexprüfungen, *Modulnummer*2)
Ko_4	(Komplexprüfungen, *Modulnummer*3)
Ko_5	(Komplexprüfungen, *Modulnummer*4)

Tabelle A.3.: Positionen markieren ein Attribut einer bestimmten Relation, man kann sie also als Tupel (Relationenname, Attributname) betrachten. In der vorliegenden Arbeit wurden hingegen die Attribute jeder Relation durchnummeriert und eine Position durch den abgekürzten Relationennamen sowie die Nummer des Attributs (tiefgestellt) gekennzeichnet.

A.5. Quelltext von ChaTEAU

Dieser Teil des Anhangs beinhaltet den Quelltext der Klassen von ChaTEAU, die im Rahmen dieser Arbeit erstellt oder verändert wurden. In Abbildung 4.3 sind aufgrund der besseren Darstellbarkeit als Klassendiagramm einige zusätzliche Klassen (Atom, IntegrityConstraint) aufgeführt, die nicht verändert wurden. Der Quellcode dieser Klassen wird im Folgenden nicht gegeben.

```
1  package terminationtest;
2
3  import java.util.ArrayList;
4  import java.util.ListIterator;
5  import java.util.Set;
6  import atom.Atom;
7  import atom.RelationalAtom;
8  import integrityConstraint.IntegrityConstraint;
9  import integrityConstraint.Tgd;
10 import term.Term;
11 import term.TermType;
12 import term.Variable;
13 import term.VariableType;
14
15 /**
16  * The class WA provides the algorithms for three very similar termination criteria: Weak
        Acyclicity (WA) itself,
17  * Rich Acyclicity (RA) and Safety (SC). Rich Acyclicity indicates the termination of the
        Naive Oblivious Chase
18  * (not implemented in ChaTEAU), whereas Safety is simply a more powerful version of Weak
        Acyclicity.
19  * The methods {@code checkWA(constraints)}, {@code checkRA(constraints)} and {@code
      checkSC(constraints)} each
20  * return a boolean which informs you if a dangerous cycle has been found. Therefore, a
        positive result serves
21  * as a warning sign - do not start the Chase algorithm since it might not terminate.
22  *
23  * @author Andreas Goerres
24  */
25 public class WA {
26
27  private ArrayList<Position>positions=new ArrayList<Position>();
28
29  /**
30   * constructor
31   */
32  public WA() {
33  }
34
35  /**
36   * Gets or, if it doesn't already exist, creates a position characterized by a relation
        and an index.
37   *
38   * @param posname the name of the position's relation
39   * @param posindex the position's index
40   * @return the position characterized by posname and posindex
41   */
42  public Position getPosition(String posname, int posindex){
43   for(Position pos: this.positions) {
44    if(pos.getRelationName().equals(posname)&&(pos.getIndex()==posindex)){
45     return pos;
46    }
47   }
```

```
48    Position pos=new Position(posname, posindex);
49    positions.add(pos);
50    return pos;
51   }
52
53   /**
54    * Safety criterion is based on the notion of "affected" positions. Here, positions are
          tested and, if
55    * necessary, marked as affected.
56    * First part of the recursive algorithm: Positions of existence quantified variables
          are always affected.
57    *
58    * @param constraints the integrity constraints whose positions are marked as affected
          if they can carry
59    * a null value
60    */
61   public void affectInit(Set<IntegrityConstraint> constraints) {
62    for(IntegrityConstraint ic1: constraints) {
63     if(ic1 instanceof Tgd) {
64
65      Tgd ic=(Tgd)ic1;
66
67      for(Atom headatom2: ic.getHead()) {
68       if(headatom2 instanceof RelationalAtom) {
69        RelationalAtom exatom=(RelationalAtom)headatom2;
70        ListIterator<Term> li3 = exatom.getTerms().listIterator();
71        String relname3=exatom.getName();
72        while(li3.hasNext()) {
73         int index3=li3.nextIndex();
74         Term headterm2=li3.next();
75         if(headterm2.getTermType()==TermType.Variable) {
76          Variable exterm=(Variable)headterm2.getTermValue();
77          if(exterm.getVariableType()==VariableType.E) {
78           getPosition(relname3, index3).setAffected(true);;
79          }
80         }
81        }
82       }
83      }
84     }
85    }
86    affect(constraints);
87   }
88
89   /**
90    * Second, recursive part of the algorithm: If a variable is in an affected body-
          position and
91    * all body-positions of this variable are affected: affect all head positions of this
          variable.
92    * Affected positions are recursively propagated until nothing changes anymore.
93    *
94    * @param constraints the integrity constraints whose positions are marked as affected
          if they can carry
95    * a null value
96    */
97   public void affect(Set<IntegrityConstraint> constraints) {
98    boolean changed=false;
99    for(IntegrityConstraint ic: constraints) {
100    if(ic instanceof Tgd) {
101     for(RelationalAtom bodyatom: ic.getBody()) {
102
```

```
103       ListIterator<Term> li1 = bodyatom.getTerms().listIterator();
104       String relname1=bodyatom.getName();
105       while(li1.hasNext()) {
106        boolean notallaffected=true;
107        int index1=li1.nextIndex();
108        Term bodyterm1=li1.next();
109        if(getPosition(relname1, index1).isAffected()) {
110         notallaffected=false;
111         if(bodyterm1.getTermType()==TermType.Variable) {
112          Variable bodyterm=(Variable)bodyterm1.getTermValue();
113          for(RelationalAtom bodyatom3: ic.getBody()) {
114           ListIterator<Term> li3 = bodyatom3.getTerms().listIterator();
115           String relname3=bodyatom3.getName();
116           while(li3.hasNext()) {
117            int index3=li3.nextIndex();
118            Term bodyterm3=li3.next();
119            if(bodyterm3.getTermType()==TermType.Variable) {
120             Variable secondbodyterm=(Variable)bodyterm3.getTermValue();
121             if((bodyterm.getIndexName().equals(secondbodyterm.getIndexName()))&&(bodyterm.
                    getIndex() ==secondbodyterm.getIndex())){
122              if(!getPosition(relname3, index3).isAffected()) {
123               notallaffected=true;
124              }
125             }
126            }
127           }
128          }
129          if(!notallaffected) {
130           for(Atom headatom1: ic.getHead()) {
131            if(headatom1 instanceof RelationalAtom) {
132             RelationalAtom headatom=(RelationalAtom)headatom1;
133             ListIterator<Term> li2 = headatom.getTerms().listIterator();
134             String relname2=headatom.getName();
135             while(li2.hasNext()) {
136              int index2=li2.nextIndex();
137              Term headterm1=li2.next();
138              if(headterm1.getTermType()==TermType.Variable) {
139               Variable headterm=(Variable)headterm1.getTermValue();
140               if(headterm.getVariableType()==VariableType.V) {
141                if((headterm.getIndexName().equals(bodyterm.getIndexName()))&&(headterm.
                       getIndex() ==bodyterm.getIndex())){
142                 if(!getPosition(relname2, index2).isAffected()) {
143                  getPosition(relname2, index2).setAffected(true);
144                  changed=true;
145                 }
146                }
147               }
148              }
149             }
150            }
151           }
152          }
153         }
154        }
155       }
156      }
157     }
158    }
159    if(changed)affect(constraints);
160   }
161
```

```
162  /**
163   * The safety criterion is almost identical to WA, but no edges originate in unaffected
          positions .
164   */
165  public void removeUnaffectedEdges() {
166   for(Position p: positions) {
167    if(!p.isAffected())p.removeAllEdges();
168   }
169  }
170
171  /**
172   * Same variable in body and head: draw edge between the respective positions, draw
          special
173   * edges to all head positions of existentially quantified variables. For rich
          acyclicity: Draw
174   * special edges from positions of each body variable to all head positions of
          existentially
175   * quantified variables
176   *
177   * @param constraints the integrity constraints to be tested
178   * @param rich if rich=={@code true}, rich acylicity will be tested instead of WA or
          safety
179   */
180  public void drawEdges(Set<IntegrityConstraint> constraints, boolean rich) {
181   boolean allquant; // variable in the body is universally quantified in body and head
182   for(IntegrityConstraint ic: constraints) {
183    if(ic instanceof Tgd) {
184     for(RelationalAtom bodyatom: ic.getBody()) {
185      ListIterator<Term> li1 = bodyatom.getTerms().listIterator();
186      String relname1=bodyatom.getName();
187      while(li1.hasNext()) {
188       allquant=false;
189       int index1=li1.nextIndex();
190       Term bodyterm1=li1.next();
191       if(bodyterm1.getTermType()==TermType.Variable) {
192        Variable bodyterm=(Variable)bodyterm1.getTermValue();
193        for(Atom headatom1: ic.getHead()) {
194         if(headatom1 instanceof RelationalAtom) {
195          RelationalAtom headatom=(RelationalAtom)headatom1;
196          ListIterator<Term> li2 = headatom.getTerms().listIterator();
197          String relname2=headatom.getName();
198          while(li2.hasNext()) {
199           int index2=li2.nextIndex();
200           Term headterm1=li2.next();
201           if(headterm1.getTermType()==TermType.Variable) {
202            Variable headterm=(Variable)headterm1.getTermValue();
203            if(headterm.getVariableType()==VariableType.V) {
204             if((headterm.getIndexName().equals(bodyterm.getIndexName()))&&(headterm.
                    getIndex() ==bodyterm.getIndex())){
205              allquant=true;
206              getPosition(relname1, index1).addEdges(getPosition(relname2, index2));
207             }
208            }
209           }
210          }
211         }
212        }
213        if(allquant || rich) {
214         for(Atom headatom2: ic.getHead()) {
215          if(headatom2 instanceof RelationalAtom) {
216           RelationalAtom exatom=(RelationalAtom)headatom2;
```

```
217             ListIterator<Term> li3 = exatom.getTerms().listIterator();
218             String relname3=exatom.getName();
219             while(li3.hasNext()) {
220              int index3=li3.nextIndex();
221              Term headterm2=li3.next();
222              if(headterm2.getTermType()==TermType.Variable) {
223               Variable exterm=(Variable)headterm2.getTermValue();
224               if(exterm.getVariableType()==VariableType.E) {
225                getPosition(relname1, index1).addSpecialedges(getPosition(relname3, index3))
                       ;
226               }
227              }
228             }
229            }
230           }
231          }
232         }
233        }
234       }
235      }
236     }
237    }
238
239    public void showPositions() {
240     for(Position p: positions) {
241      System.out.println("Position: "+p.getRelationName()+"_"+p.getIndex()+", Affected: "+p.
           isAffected());
242      for(Position p2: p.getEdges()) {
243       System.out.println("\tEdge to: "+p2.getRelationName()+"_"+p2.getIndex());
244      }
245      for(Position p3: p.getSpecialedges()) {
246       System.out.println("\tSpecial Edge to: "+p3.getRelationName()+"_"+p3.getIndex());
247      }
248     }
249    }
250
251    ArrayList<Position>finished=new ArrayList<Position>();
252
253    /**
254     * Find a cycle of nodes with at least one special edge. Algorithm is based on recursive
           depth first search.
255     *
256     * @return {@code true} if a cycle through a special edge has been found, {@code false}
           otherwise
257     */
258    public boolean searchSpecialCycle() {
259     for(Position p: positions) {
260      if(searchSpecialCycle(p, -1, new ArrayList<Position>()))return true;
261     }
262     return false;
263    }
264
265    /**
266     * Recursive part of the method.
267     *
268     * @param v the starting position
269     * @param durchSpecial the most recent index of "visited" with a special edge
270     * @param visited list of already visited nodes
271     * @return {@code true} if a cycle through a special edge has been found, {@code false}
           otherwise
272     */
```

```
273   public boolean searchSpecialCycle(Position v, int durchSpezial, ArrayList<Position>
          visited) {
274     if(finished.contains(v)) {
275      return false;
276     }
277     if (visited.contains(v)) {
278      if(visited.indexOf(v)>durchSpezial) {
279       return false;
280      }
281      else {
282       return true;
283      }
284     }
285     visited.add(v);
286     for(Position w2: v.getSpecialedges()) {
287      if(searchSpecialCycle(w2, visited.indexOf(v), visited)) {
288       return true;
289      }
290     }
291     for(Position w1: v.getEdges()) {
292      if(searchSpecialCycle(w1, durchSpezial, visited)) {
293       return true;
294      }
295     }
296    finished.add(v);
297    return false;
298   }
299
300   /**
301    * Based on the boolean arguments, three different termination tests can be performed:
           WA, RA and SC.
302    *
303    * @param constraints the integrity constraints to be tested
304    * @param safety {@code true} if safety should be tested instead of WA
305    * @param rich {@code true} if rich acyclicity should be tested instead of WA
306    * (don't combine with safety!)
307
308    * @return {@code true} if integrity constraints are not weakly acyclic/ richly acyclic/
           safe,
309    * {@code false} otherwise
310    */
311   public boolean checkWA(Set<IntegrityConstraint> constraints, boolean safety, boolean
          rich) {
312    this.drawEdges(constraints, rich);
313    if(safety) {
314     this.affectInit(constraints);
315     this.removeUnaffectedEdges();
316    }
317    return this.searchSpecialCycle();
318   }
319
320   /**
321    * This tests a set of integrity constraints for rich acyclicity. Naive oblivious chase
           termination
322    * is guaranteed if this test returns {@code false}.
323    *
324    * @param constraints the integrity constraints to be tested
325    * @return {@code true} if integrity constraints are not richly acyclic,
326    * {@code false} otherwise
327    */
328   public boolean checkRA(Set<IntegrityConstraint> constraints) {
```

```
329     this.drawEdges(constraints, true);
330     return this.searchSpecialCycle();
331   }
332
333   /**
334    * This tests a set of integrity constraints for weak acyclicity. Standard chase
          termination
335    * is guaranteed if this test returns {@code false}.
336    *
337    * @param constraints the integrity constraints to be tested
338    * @return {@code true} if integrity constraints are not weakly acyclic,
339    * {@code false} otherwise
340    */
341   public boolean checkWA(Set<IntegrityConstraint> constraints) {
342     this.drawEdges(constraints, false);
343     return this.searchSpecialCycle();
344   }
345
346   /**
347    * This tests a set of integrity constraints for safety. Standard chase termination
348    * is guaranteed if this test returns {@code false}.
349    *
350    * @param constraints the integrity constraints to be tested
351    * @return {@code true} if integrity constraints are not safe,
352    * {@code false} otherwise
353    */
354   public boolean checkSC(Set<IntegrityConstraint> constraints) {
355     this.drawEdges(constraints, false);
356     this.affectInit(constraints);
357     this.removeUnaffectedEdges();
358     return this.searchSpecialCycle();
359   }
360 }
```

Listing A.1: Klasse WA

```
1  package terminationtest;
2
3  import java.util.ArrayList;
4  import java.util.HashMap;
5  import java.util.HashSet;
6  import java.util.Set;
7  import atom.RelationalAtom;
8  import integrityConstraint.IntegrityConstraint;
9  import integrityConstraint.Tgd;
10 import term.Term;
11 import term.TermType;
12 import term.Variable;
13 import term.VariableType;
14
15 /**
16  * This class performs the Adn++-algorithm to determine cyclicity of TGDs. General
       integrity constraints
17  * will be analyzed (use {@code prepareConstraintAdn()}), however, EGDs will be ignored (
       which could lead
18  * to false results). If you wish to include EGDs into your analysis, perform EGD-
       rewriting (SfS) first.
19  * To start the rewriting process of TGDs, use the boolean method {@code prepareAdn()},
       which returns false
20  * if the TGDs are acyclic (which guarantees chase termination).
21  * Warning: Constraint-rewriting tends to be quite slow on large TGD sets, especially
       after performing SfS.
```

```
22   *
23   * @author Andreas Goerres
24   */
25  public class ConstraintRewriting {
26
27   /**
28    * constructor
29    */
30   public ConstraintRewriting() {
31   }
32
33   /**
34    * Every TGD receives an id, the index. Copies of the TGD will
35    * keep the index of the original TGD.
36    *
37    * @param tgds the tgds in need of an id
38    */
39   public static void nameTGDs(Set<Tgd>tgds) {
40    int i=1;
41    for(Tgd t: tgds) {
42     t.setIndex(i);
43     ++i;
44    }
45   }
46
47   /**
48    * Adornments of a TGD's atoms and terms are consistent.
49    *
50    * @param t the testet TGD
51    * @return {@code true} if the TGD is consistent, {@code false} otherwise
52    */
53   public static boolean tgdIsConsistent(Tgd t) {
54    for(RelationalAtom a: t.getBody()) {
55     if(!a.isConsistent())return false;
56    }
57    for(RelationalAtom b: t.getHead()) {
58     if(!b.isConsistent())return false;
59    }
60    return true;
61   }
62
63   /**
64    * Adornments of two TGDs can be substituted.
65    *
66    * @param source the newly created adorned TGD
67    * @param target a previously created adorned TGD
68    * @return {@code true} if the TGD is consistent, {@code false} otherwise
69    */
70   public static boolean substitutable(Tgd source, Tgd target) {
71    if(!tgdIsConsistent(target)) {
72     return false;
73    }
74    if(compareTGDs(source, target)) {
75     return false;
76    }
77    if(source.getIndex()!=target.getIndex())return false;
78    boolean full=true;
79    for(RelationalAtom a: source.getHead()) {
80     for(Term t:a.getTerms()) {
81      if(t.getTermType()==TermType.Variable) {
82       if(((Variable)t.getTermValue()).getVariableType()==VariableType.E)full=false;
```

```
83      }
84     }
85    }
86    if(full)return false;
87    HashMap<Adornment, Adornment>substitutions=new HashMap<Adornment, Adornment>();
88    for(RelationalAtom ra1: source.getBody()) {
89     for(RelationalAtom ra2: target.getBody()) {
90      if(ra1.equals(ra2)) {
91       for(int i=0;i<ra1.getTerms().size();++i) {
92        if(((ra1.getTerms().get(i).getAdornment().isB())&&(ra2.getTerms().get(i).
              getAdornment().isF()))||((ra1.getTerms().get(i).getAdornment().isF())&&(ra2.
              getTerms().get(i).getAdornment().isB()))) {
93         return false;
94        }
95        if(substitutions.containsKey(ra1.getTerms().get(i).getAdornment())) {
96         if(!substitutions.get(ra1.getTerms().get(i).getAdornment()).equals(ra2.getTerms().
              get(i).getAdornment())) return false;
97        }
98        else {
99         substitutions.put(ra1.getTerms().get(i).getAdornment(), ra2.getTerms().get(i).
              getAdornment());
100       }
101      }
102     }
103    }
104   }
105   for(RelationalAtom ra1: source.getHead()) {
106    for(RelationalAtom ra2: target.getHead()) {
107     if(ra1.equals(ra2)) {
108      for(int i=0;i<ra1.getTerms().size();++i) {
109       if(((ra1.getTerms().get(i).getAdornment().isB())&&(ra2.getTerms().get(i).
              getAdornment().isF()))||((ra1.getTerms().get(i).getAdornment().isF())&&(ra2.
              getTerms().get(i).getAdornment().isB()))) {
110        return false;
111       }
112       if(substitutions.containsKey(ra1.getTerms().get(i).getAdornment())) {
113        if(!substitutions.get(ra1.getTerms().get(i).getAdornment()).equals(ra2.getTerms().
              get(i).getAdornment())) return false;
114       }
115       else {
116        substitutions.put(ra1.getTerms().get(i).getAdornment(), ra2.getTerms().get(i).
              getAdornment());
117       }
118      }
119     }
120    }
121   }
122   for(Adornment a: substitutions.keySet()) {
123    if(substitutions.containsValue(a)) {
124     if((!a.equals(substitutions.get(a)))) return false;
125    }
126   }
127   return true;
128  }
129
130  /**
131   * Actual precedence (that is, t1<t2) is difficult to calculate. However, most of the
        interesting
132   * problem cases are dealt with later anyway (because of adornment inconsistencies).
        Here, we only
133   * check if the name of a relation in the head of t1 also appears in the body of t2 (and
```

```
        test for
134     * basic consistency of constants and existentially quantified variables). In other
            words, we
135     * perform the test using predicates (relational atoms), not the TGDs themselves.
136     *
137     * @param t1 the TGD that might trigger t2
138     * @param t2 the TGD that might be triggered by t1
139     * @return {@code true} if t1<t2, {@code false} otherwise
140     */
141    public static boolean pseudoPrecedence(Tgd t1, Tgd t2) {
142     for(RelationalAtom a1: t1.getHead()) {
143      if(pseudoPrecedence(a1, t2)) return true;
144     }
145     return false;
146    }
147
148    /**
149     * @param a1 a predicate from TGD t1's head
150     * @param t2 the TGD that might be triggered by t1
151     * @return {@code true} if t1<t2, {@code false} otherwise
152     */
153    public static boolean pseudoPrecedence(RelationalAtom a1, Tgd t2) {
154     for(RelationalAtom a2: t2.getBody()) {
155      if(pseudoPrecedence(a1, a2)) return true;
156     }
157     return false;
158    }
159
160    /**
161     * Used not only for the pseudoprecedence of TGDs, but also to detect which atoms of
            those
162     * TGDs could transfer their adornments.
163     * Special problem cases detected here: TGD t1 generates a constant, but TGD t2 requires
            a
164     * different constant, t1 generates a null value, but t2 requires a constant.
165     *
166     * @param a1 a predicate from TGD t1's head
167      * @param a2 a predicate from TGD t2's body
168
169     * @return {@code true} if t1<t2 and a1 could transfer its adornments to a2,
170     * {@code false} otherwise
171     */
172    public static boolean pseudoPrecedence(RelationalAtom a1, RelationalAtom a2) {
173     if(a1.getName().equals(a2.getName())&&(a1.getTerms().size()==a2.getTerms().size())) {
174      for(int i=0;i<a2.getTerms().size();++i) {
175       if(a2.getTerms().get(i).getTermType()==TermType.Const) {
176        if(a1.getTerms().get(i).getTermType()==TermType.Const) {
177         if(a1.getTerms().get(i).getConstType()!=a2.getTerms().get(i).getConstType()) return
                false;
178         if(!a1.getTerms().get(i).getTermValue().equals(a2.getTerms().get(i).getTermValue())
               ) return false;
179        }
180        if(a1.getTerms().get(i).getTermType()==TermType.Variable) {
181         Variable v=(Variable)a1.getTerms().get(i).getTermValue();
182         if(v.getVariableType()==VariableType.E) return false;
183        }
184       }
185      }
186      return true;
187     }
188     return false;
```

```
189  }
190
191  /**
192   * You would expect that variables with identical names have identical adornments (at
          least in the
193   * same TGD), but they might be different Java-objects. Here, we use the terms of one
          atom - probably
194   * an atom with freshly adorned terms - to adjust the adornments of all the other terms
          in the TGD (one
195   * might restrict this to variables, but right now, constants have identical b-
          adornments anyway).
196   *
197   * @param vorlage the atom with "correctly" adorned terms
198   * @param tgd the TGD containing inconsistently adorned terms
199   */
200  public static  void propagateOnSameTerm(RelationalAtom vorlage, Tgd tgd) {
201   for(Term t: vorlage.getTerms()) {
202    for(RelationalAtom a: tgd.getBody()) {
203     for(Term t2: a.getTerms()) {
204      if(t.equals(t2))t2.setAdornment(t.getAdornment());
205     }
206    }
207    for(RelationalAtom a: tgd.getHead()) {
208     for(Term t2: a.getTerms()) {
209      if(t.equals(t2))t2.setAdornment(t.getAdornment());
210     }
211    }
212   }
213  }
214
215  /**
216   * After calling {@code propagateOnSameTerm()}, there are still two problems:
217   * 1) We adjusted the term adornments, but the atom adornments might be inconsistent,
218   * 2) we did not adjust adornments of existentially quantified variables (which depend
          on the adornments of
219   * variables we previously adjusted).
220   *
221   * @param t the TGD whose head's adornments need to be adjusted if the body is
          consistent
222   * @return {@code true} if the body of t is consistent, {@code false} otherwise
223   */
224  public static boolean headAdornment(Tgd t) {
225   if(!isConsistent(t))return false;
226   forceHeadAdornment(t);
227   return true;
228  }
229
230  /**
231   * Adornments of existentially quantified variables depend on the adornments of all
          universally
232   * quantified variables of the TGD's head. Since this adornment is new, the atom's
          adornment is
233   * obviously now inconsistent and needs to be adjusted to the new adornment.
234   *
235   * @param t the TGD whose head's adornments need to be adjusted
236   * @return {@code true} if the new mapping has been added, {@code false} otherwise
237   */
238  public static void forceHeadAdornment(Tgd t) {
239   t.setAdornmentsForSkolemization();
240   for(RelationalAtom a1: t.getHead()) {
241    a1.receiveAdornments();
```

```
242     }
243    }
244
245    /**
246     * Atoms have an adornment for each of their terms. Those terms have their own
           adornments. In consistent
247     * atoms, those adornments match perfectly. Consistent TGD only contain consistent atoms
           (in the body,
248     * atoms in the head should never be inconsistent).
249     *
250     * @param t the TGD tested for inconsistencies
251     * @return {@code true} if the TGD is consistent, {@code false} otherwise
252     */
253    public static boolean isConsistent(Tgd t) {
254     for(RelationalAtom a1: t.getBody()) {
255      if(!a1.isConsistent()) {
256       return false;
257      }
258     }
259     return true;
260    }
261
262    /**
263     * Duplicates a TGD including all of its adornments. Terms are duplicated, but the terms
           ' variables remain
264     * identical Java-objects - however, we never change those anyway.
265     *
266     * @param t the TGD to be duplicated
267     * @return a new Java-object with properties identical to t
268     */
269    public static Tgd duplicateTGD(Tgd t1) {
270     HashSet<RelationalAtom>body=new HashSet<RelationalAtom>();
271     HashSet<RelationalAtom>head=new HashSet<RelationalAtom>();
272     for(RelationalAtom a: t1.getBody()) {
273      ArrayList<Term>terms=new ArrayList<Term>();
274      for(Term t: a.getTerms()) {
275       terms.add(new Term(t));
276      }
277      RelationalAtom a2=new RelationalAtom(a.getName(), terms);
278      a2.setAdornments(a.getAdornments());
279      body.add(a2);
280     }
281     for(RelationalAtom a: t1.getHead()) {
282      ArrayList<Term>terms=new ArrayList<Term>();
283      for(Term t: a.getTerms()) {
284       terms.add(new Term(t));
285      }
286      RelationalAtom a2=new RelationalAtom(a.getName(), terms);
287      a2.setAdornments(a.getAdornments());
288      head.add(a2);
289     }
290     Tgd t2=new Tgd(body, head);
291     t2.setIndex(t1.getIndex());
292     return t2;
293    }
294
295    /**
296     * Compare two TGDs. The TGDs don't have to be consistent since adornments of terms are
           also compared.
297     * Simply comparing the TGD with {@code equals()} wouldn't compare the adornments at all
           , and
```

```
298   * comparing the results of {@code stringifyTGD()} would compare adornments of atoms,
          but not of
299   * terms.
300   *
301   * @param t1 the first TGD
302   * @param t2 the second TGD
303   * @return {@code true} if t1 and t2 are equal, {@code false} otherwise
304   */
305  public static boolean compareTGDs(Tgd t1, Tgd t2) {
306   if(t1.getIndex()!=t2.getIndex())return false;
307   if(!stringifyTGD(t1).equals(stringifyTGD(t2)))return false;
308   for(RelationalAtom a: t1.getBody()) {
309    for(RelationalAtom a2: t2.getBody()) {
310     if(a.equals(a2)) {
311      if(!a.getAdornments().equals(a2.getAdornments()))return false;
312      for(int i=0;i<a.getTerms().size();++i) {
313       if(!a.getTerms().get(i).getAdornment().equals(a2.getTerms().get(i).getAdornment()))
              return false;
314      }
315     }
316    }
317   }
318   for(RelationalAtom a: t1.getHead()) {
319    for(RelationalAtom a2: t2.getHead()) {
320     if(a.equals(a2)) {
321      if(!a.getAdornments().equals(a2.getAdornments()))return false;
322      for(int i=0;i<a.getTerms().size();++i) {
323       if(!a.getTerms().get(i).getAdornment().equals(a2.getTerms().get(i).getAdornment()))
              return false;
324      }
325     }
326    }
327   }
328   return true;
329  }
330
331  /**
332   * Utility function that transforms a set of TGDs into a list of TGDs.
333   *
334   * @param set the set of TGDs
335   * @return a list containing the set's TGDs
336   */
337  public static ArrayList<Tgd>setToList(Set<Tgd>set){
338   ArrayList<Tgd>list=new ArrayList<Tgd>();
339   for(Tgd t: set) {
340    list.add((Tgd)t);
341   }
342   return list;
343  }
344
345  /**
346   * Since Adn++ only accepts TGDs, we have to filter the original set of integrity
          constraints and
347   * remove EGDs. Of course, EGD-rewriting removes EGDs anyway, so we won't call {@code
          extractTGDs()}
348   * after performing SfS.
349   *
350   * @param set the set of integrity constraint
351   * @return a subset of the original set containing only its TGDs
352   */
353  public static Set<Tgd>extractTGDs(Set<IntegrityConstraint>set){
```

```
354     Set<Tgd>list=new HashSet<Tgd>();
355     for(IntegrityConstraint t: set) {
356      if(t instanceof Tgd)
357      list.add((Tgd)t);
358     }
359     return list;
360    }
361
362    public void showPredbase() {
363     System.out.println("Generated adorned predicates: ");
364     for(RelationalAtom a: predbase) {
365      System.out.println(stringifyAtom(a));
366     }
367     System.out.println("");
368    }
369
370    /**
371     * All Adornments are b-Adornments.
372     *
373     * @param a an adorned relational atom
374     * @return {@code true} if all of the atom's adornments are b, {@code false} otherwise
375     */
376    public boolean isB(RelationalAtom a) {
377     for(Adornment ad: a.getAdornments()) {
378      if(ad.isF())return false;
379     }
380     return true;
381    }
382
383    private ArrayList<RelationalAtom>predbase=new ArrayList<RelationalAtom>();
384    private GraphUtilities graph=new GraphUtilities();
385
386    /**
387     * After parsing an xml- or txt-file, we generally have a set of integrity constraints.
         However, we
388     * could have performed EGD-rewriting, which creates a set of TGDs. In the first case,
         we have to
389     * exclude the EGDs first. Returns {@code false} if chase termination is guaranteed.
390     *
391     * @param constraints the set of EGDs and TGDs
392     * @return {@code true} if the integrity constraints are not acyclic, {@code false}
         otherwise
393     */
394    public boolean prepareConstraintAdn(Set<IntegrityConstraint>constraints) {
395     return prepareAdn(extractTGDs(constraints));
396    }
397
398    /**
399     * Performs the initial phase of constraint-rewriting. However, the separation of {@code
         prepareAdn()}
400     * and {@code adnPlusPlus()} is simply for clarity. Algorithm uses a list semantics, but
         calling it
401     * repeatedly might result in differently ordered lists (and therefore run time).
402     * Returns {@code false} if chase termination is guaranteed.
403     *
404     * @param tgds a set of TGDs
405     * @return {@code true} if the TGDs are not acyclic, {@code false} otherwise
406     */
407    public boolean prepareAdn(Set<Tgd>tgds) {
408     // gibt urspruenglichen TGDs eindeutigen Index
409     nameTGDs(tgds);
```

```
410   ArrayList<Tgd>base=setToList(tgds);
411   ArrayList<RelationalAtom>preds=new ArrayList<RelationalAtom>();
412   for(Tgd t: base) {
413    for(RelationalAtom a: t.getBody()) {
414     // Adornments der Terme werden zu Adornments des Atoms
415     a.receiveAdornments();
416    }
417    // erzeuge f-Adornments der existenzquantifizierten Variablen
418    t.setAdornmentsForSkolemization();
419    for(RelationalAtom a: t.getHead()) {
420     a.receiveAdornments();
421    }
422    for(RelationalAtom head: t.getHead()) {
423     boolean unknown=true;
424     for(RelationalAtom pred: preds) {
425      if(stringifyAtomhead(pred).equals(stringifyAtomhead(head)))unknown=false;
426     }
427     if(unknown) {
428      // neue Praedikate werden gespeichert
429      preds.add(head);
430     }
431    }
432   }
433   return adnPlusPlus(preds, base);
434  }
435
436  /**
437   * The main phase of Adn++. Method is renamed to "adnPP()" in the written thesis (due to
          lack of space).
438   * The arguments "predsunion" and "base" also have different names.
439   * Returns {@code false} if chase termination is guaranteed.
440   *
441   * @param predsunion the list of adorned predicates created in the initial phase of Adn
          ++
442   * @param base the list of adorned TGDs created in the initial phase of Adn++
443   * @return {@code true} if the TGDs are not acyclic, {@code false} otherwise
444   */
445  public boolean adnPlusPlus(ArrayList<RelationalAtom> predsunion, ArrayList<Tgd>base) {
446
447   boolean cyk=false;
448   // um effizient Koerper und Kopf neuer und alter TGDs ueber ihren Hash zu vergleichen
449   HashSet<String>bodystrings=new HashSet<String>();
450   HashSet<String>headstrings=new HashSet<String>();
451   for(Tgd t: base) {
452    String tst=stringifyTGD(t);
453    bodystrings.add(tst.split("->")[0].trim());
454    headstrings.add(tst.split("->")[1].trim());
455   }
456   // Zyklus durch adornte Praedikate
457   for(int predindex=0;predindex<predsunion.size();++predindex){
458    RelationalAtom p=predsunion.get(predindex);
459    // erzeugt den Praedikat-Knoten, wenn noch nicht vorhanden
460    GraphUtilities.Node node1=graph.getNode(p);
461    // Zyklus durch adornte TGDs
462    for(int tgdindex=0; tgdindex<base.size();++tgdindex){
463     Tgd t=base.get(tgdindex);
464     // p kann in den Koerper von t eingesetzt werden und ist kompatibel
465     if(pseudoPrecedence(p, t)) {
466      for(RelationalAtom p2: t.getBody()) {
467       if(pseudoPrecedence(p, p2)&&(!p.getAdornments().equals(p2.getAdornments()))&&(isB(
            p2))) {
```

```
468         Tgd adornedTGD=duplicateTGD(t);
469         for(RelationalAtom p3: adornedTGD.getBody()) {
470          if(p2.equals(p3)) {
471           p3.setAdornments(p.getAdornments());
472           // Adornments werden auf Atom uebertragen
473           for(int i=0;i<p3.getTerms().size();++i) {
474            Term con=p3.getTerms().get(i);
475            if(con.getTermType()==TermType.Const)p3.getAdornments().set(i, new Adornment())
                 ;
476           }
477           // Adornments werden zu Adornments der Terme
478           p3.transferAdornments();
479           // gleichnamige Terme erhalten gleiche Adornments
480           propagateOnSameTerm(p3, adornedTGD);
481          }
482         }
483         // Adornments des TGD-Koerpers nicht konsistent
484         if(!headAdornment(adornedTGD)) {
485          boolean tgdisnew1=true;
486          if(bodystrings.contains(stringifyTGD(adornedTGD).split("->")[0]))tgdisnew1=false;
487          if(tgdisnew1) {
488           // nicht konsistente TGD wird vielleicht spaeter konsistent  wenn weitere
                adornte Praedikate eingefuegt wurden
489           base.add(adornedTGD);
490           String atgt=stringifyTGD(adornedTGD);
491           bodystrings.add(atgt.split("->")[0].trim());
492           headstrings.add(atgt.split("->")[1].trim());
493          }
494         }
495         // Adornments sind konsistent, Adornments fuer existenzquant. Variablen im Kopf
             wurden neu bestimmt
496         else {
497          boolean tgdisnew=true;
498          if(headstrings.contains(stringifyTGD(adornedTGD).split("->")[1].trim()))tgdisnew=
               false;
499          if(!bodystrings.contains(stringifyTGD(adornedTGD).split("->")[0].trim()))base.add
               (adornedTGD);
500          // hier erfolgt Test auf Substitution der Adornments
501          if(tgdisnew) {
502           String btst=stringifyTGD(adornedTGD);
503           bodystrings.add(btst.split("->")[0].trim());
504           headstrings.add(btst.split("->")[1].trim());
505           cyk=false;
506           for(Tgd oldTGD: base) {
507            // die Adornments der TGDs sind substituierbar
508            if(substitutable(adornedTGD, oldTGD)) {
509             for(RelationalAtom sub: oldTGD.getHead()) {
510              GraphUtilities.Node node3=graph.getNode(sub);
511              // Praedikatengraph wird erweitert
512              node1.addLink(node3);
513             }
514             // TGDs nicht azyklisch, Chase terminiert wohl nicht
515             if(graph.sucheZyklus())return true;
516             // durch Substitution wurde kein Zyklus, sondern eine "zusammenlaufende
                  Struktur" gefunden
517             cyk=true;
518            }
519           }
520          }
521          // da Substitution nicht erfolgreich war, Speichern aller neuen adornten
               Praedikate des Kopfes
```

```
522          if(tgdisnew&&(!cyk)) {
523           for(RelationalAtom pred: adornedTGD.getHead()) {
524            boolean unknown=true;
525            for(RelationalAtom exis: predsunion) {
526             if((stringifyAtomhead(pred).equals(stringifyAtomhead(exis))))unknown=false;
527            }
528            if(unknown) {
529             predsunion.add(pred);
530             // Praedikatengraph wird erweitert
531             GraphUtilities.Node node2=graph.getNode(pred);
532             node1.addLink(node2);
533            }
534           }
535          }
536         }
537        }
538       }
539      }
540     }
541    }
542    // da kein vorlaeufiger Abbruch nach Finden eines Zyklus: TGDs sind azyklisch, Chase
         terminiert garantiert
543    return false;
544   }
545
546   /**
547    * Creates a String representation of a relational atom including all of its adornments
         and
548    * terms (but not the terms' adornments, so inconsistencies can't be detected this way).
549    * The {@code toString()}-method would omit the adornments.
550    *
551    * @param ra1 the relational atom to be stringified
552    * @return the String representation of ra1
553    */
554   public static String stringifyAtom(RelationalAtom ra1) {
555    String result="";
556    result+=ra1.getName()+"^";
557    for(int i=0;i<ra1.getAdornments().size();++i) {
558     Adornment a=ra1.getAdornments().get(i);
559     result+=a;
560    }
561    result+=" (";
562    for(int i=0;i<ra1.getTerms().size();++i) {
563     Term t= ra1.getTerms().get(i);
564     result+=t;
565     if(i<(ra1.getTerms().size()-1))result+=", ";
566    }
567    result+=")";
568    return result;
569   }
570
571   /**
572    * Creates a String representation of a relational atom including all of its adornments,
         but
573    * without the terms. The {@code toString()}-Method would omit the adornments, but
         include
574    * the terms, while {@code stringifyAtom()} would include both terms and adornments.
575    *
576    * @param ra1 the relational atom to be stringified without its terms
577    * @return the String representation of ra1
578    */
```

```
579   public static String stringifyAtomhead(RelationalAtom ra1) {
580    String result="";
581    result+=ra1.getName()+"^";
582    for(int i=0;i<ra1.getAdornments().size();++i) {
583     Adornment a=ra1.getAdornments().get(i);
584     result+=a;
585    }
586    return result;
587   }
588
589   /**
590    * Creates a String representation of a TGD including all of its adornments (but not the
591    * terms' adornments, so inconsistencies can't be detected this way).
592    * The {@code toString()}-method would omit the adornments and add line breaks after
         each atom,
593    * while this method doesn't add line separators. You would expect that the String
         representation
594    * of a TGD is not unique (since body and head are both sets of relational atoms), but I
          found it
595    * to be sufficient for testing equality of TGDs (and to detect the presence of a TGD-
         representation
596    * in a hashset of TGD-representations). For detecting containment in a hashset directly
         ,
597    * I would have to modify the {@code equals()} and {@code hash()}-methods of TGDs (and
         even relational
598    * atoms), which might interfere with the actual chase.
599    *
600    * @param tgd the TGD to be stringified
601    * @return the String representation of the TGD
602    */
603   public static String stringifyTGD(Tgd tgd) {
604    if(tgd==null)return "(no TGD)";
605    String result="";
606    for(RelationalAtom ra1: tgd.getBody()) {
607     result+=ra1.getName()+"^";
608     for(int i=0; i<ra1.getAdornments().size();++i) {
609      Adornment a=ra1.getAdornments().get(i);
610      result+=a;
611     }
612     result+=" (";
613     for(int i=0;i<ra1.getTerms().size();++i) {
614      Term t= ra1.getTerms().get(i);
615      result+=t;
616      if(i<(ra1.getTerms().size()-1))result+=", ";
617     }
618     result+="), ";
619    }
620    if(result.contains(","))result=result.substring(0, result.lastIndexOf(","));
621    result+=" -> ";
622    for(RelationalAtom ra1: tgd.getHead()) {
623     result+=ra1.getName()+"^";
624     for(int i=0; i<ra1.getAdornments().size();++i) {
625      Adornment a=ra1.getAdornments().get(i);
626      result+=a;
627     }
628     result+=" (";
629     for(int i=0;i<ra1.getTerms().size();++i) {
630      Term t= ra1.getTerms().get(i);
631      result+=t;
632      if(i<(ra1.getTerms().size()-1))result+=", ";
633
```

```
634     }
635     result+="), ";
636    }
637    if(result.contains(","))result=result.substring(0, result.lastIndexOf(","));
638    return result;
639   }
640  }
```

Listing A.2: Klasse ConstraintRewriting

```
1  package terminationtest;
2
3  import java.util.HashSet;
4
5  /**
6   * An Adornment is a flag belonging to a term or an atom that is propagated during Adn++
7   * (similar to the value of variables during chase). For consistent atoms, the atom's
       adornments correspond
8   * to the adornments of the atom's terms. While b-Adornments are only characterized only
       by their
9   * {@code AdornmentType}, f-Adornments reference a skolem function and include therefore
       the parameters of
10  * this function - a set of other adornments. Additionally, f-adornments are
       characterized by the TGD and the
11  * existentially quantified variable that led to their creation.
12  * The default {@code AdornmentType} is b.
13  *
14  * @author Andreas Goerres
15  */
16 public class Adornment {
17
18  private AdornmentType type=AdornmentType.b;
19  private int constraintindex=-1;
20  private String variable=null;
21  private HashSet<Adornment>skolemOf=null;
22
23  /**
24   * @return the existentially quantified variable characterizing the f-adornment, {@code
       null} for b-adornments
25   */
26  public String getVariable() {
27   return variable;
28  }
29
30  /**
31   * @param variable the variable to be set
32   */
33  public void setVariable(String variable) {
34   this.variable = variable;
35  }
36
37  /**
38   * @return the adornment type (f or b)
39   */
40  public AdornmentType getType() {
41   return type;
42  }
43
44  /**
45   * @param type the adornment type to be set
46   */
47  public void setType(AdornmentType type) {
```

```
48     this.type = type;
49   }
50
51   /**
52    * @return the id (index) of the TGD originally generating the f-adornment, -1 for b-
          adornments
53    */
54   public int getConstraintindex() {
55    return constraintindex;
56   }
57
58   /**
59    * @param constraintindex the constraint index to be set
60    */
61   public void setConstraintindex(int constraintindex) {
62    this.constraintindex = constraintindex;
63   }
64
65
66   /**
67    * @return the set of adornments that are the parameters of the associated skolem
          function
68    */
69   public HashSet<Adornment> getSkolemOf() {
70    return skolemOf;
71   }
72
73   /**
74    * @param skolemOf the adornments to be set as parameters of the skolem function
75    */
76   public void setSkolemOf(HashSet<Adornment> skolemOf) {
77    this.skolemOf = skolemOf;
78   }
79
80   /**
81    * constructor for b-adornments
82    */
83   public Adornment() {
84   }
85
86   /**
87    * constructor
88    */
89   public Adornment(AdornmentType type) {
90    if(type==AdornmentType.f) {
91     this.setType(type);
92     this.setConstraintindex(0);
93     this.setVariable("");
94     this.setSkolemOf(new HashSet<Adornment>());
95    }
96   }
97
98   /**
99    * constructor
100   */
101  public Adornment(int constraintindex, String variable, HashSet<Adornment>skolemOf) {
102   this.type=AdornmentType.f;
103   this.constraintindex=constraintindex;
104   this.variable=variable;
105   this.skolemOf=skolemOf;
106  }
```

```
107
108  /**
109   * @return {@code true} if this is a b-adornment, {@code false} otherwise
110   */
111  public boolean isB() {
112   return this.type==AdornmentType.b;
113  }
114
115  /**
116   * @return {@code true} if this is an f-adornment, {@code false} otherwise
117   */
118  public boolean isF() {
119   return this.type==AdornmentType.f;
120  }
121
122  public String toString() {
123   if(this.isB()) return "b";
124   String sko="";
125   for(Adornment a: this.getSkolemOf()) {
126    sko+=a.toString();
127   }
128   return "f^r"+this.getConstraintindex()+"_"+this.getVariable()+"("+sko+")";
129  }
130
131  @Override
132  public boolean equals(Object a1) {
133   if(a1 instanceof Adornment) {
134    Adornment a=(Adornment)a1;
135    if(this.isB()) {
136     return a.isB();
137    }
138    else {
139     return this.toString().equals(a.toString());
140    }
141   }
142   else return false;
143  }
144 }
```

Listing A.3: Klasse Adornment

```
1  package terminationtest;
2
3  /**
4   * Adornments can either be free (f-adornments) or bounded (b-adornments).
5   * While existentially quantified variables always carry f-Adornments,
6   * constants always carry b-adornments.
7   *
8   * @author Andreas Goerres
9   */
10 public enum AdornmentType {
11  f,
12  b;
13 }
```

Listing A.4: Enum AdornmentType

```
1  package terminationtest;
2
3  import java.util.ArrayList;
4  import atom.RelationalAtom;
5  import integrityConstraint.Tgd;
```

```
6
7  /**
8   * This class provides us with a general test for cyclicity in a graph. The algorithm
9   * can't be used to search for "special cycles" (which would be needed for the Weak
        Acyclicity criterion),
10  * so the algorithm in {@link terminationtest.WA} is a bit more complex than the one
        provided here.
11  * In theory, the algorithm could be used for general cycles, however I only implemented
        nodes for
12  * RelationalAtoms, because I need those for the Constraint-Rewriting-Algorithm. To
        search for cycles in
13  * a graph with a different node type, just extend the GeneralNode-class.
14  *
15  * @author Andreas Goerres
16  */
17 public class GraphUtilities {
18
19  private ArrayList<GeneralNode>graph=new ArrayList<GeneralNode>();
20  private ArrayList<GeneralNode>visited=new ArrayList<GeneralNode>();
21  private ArrayList<GeneralNode>finished=new ArrayList<GeneralNode>();
22
23  /**
24   * constructor
25   */
26  public GraphUtilities() {
27  }
28
29  public ArrayList<GeneralNode> getGraph() {
30   return graph;
31  }
32
33  /**
34   * @param graph the graph to be set
35   */
36  public void setGraph(ArrayList<GeneralNode> graph) {
37   this.graph = graph;
38  }
39
40  /**
41   * Prints all vertices and edges (links) of the graph.
42   *
43   */
44  public void showGraph() {
45   for(GeneralNode node: graph) {
46    System.out.println(node);
47    for(GeneralNode edge: node.linksTo) {
48     System.out.println("\tlinks to: "+edge);
49    }
50   }
51  }
52
53  /**
54   * Searching for a cycle in the graph using depth first search.
55   *
56   * @return {@code true} if a cycle has been found, {@code false} otherwise
57   */
58  public boolean sucheZyklus() {
59   finished=new ArrayList<GeneralNode>();
60   showGraph();
61   for(GeneralNode n1: this.graph) {
62    visited=new ArrayList<GeneralNode>();
```

```
63      if(n1 instanceof Node) {
64
65       Node n=(terminationtest.GraphUtilities.Node)n1;
66       if(sucheZyklus(n))return true;
67      }
68     }
69     System.out.println("kein Zyklus");
70     return false;
71    }
72
73    /**
74     * Recursive part of the method.
75     *
76     * @param node the current starting node
77     * @return {@code true} if a cycle has been detected, {@code false} otherwise
78     */
79    public boolean sucheZyklus(Node n) {
80     if (visited.contains(n)) {
81      return true;
82     }
83     if (finished.contains(n)) {
84      return false;
85     }
86     visited.add(n);
87     for(GeneralNode n1: n.getLinksTo()) {
88      if(sucheZyklus((Node)n1))return true;
89     }
90     finished.add(n);
91     return false;
92    }
93
94    private int universalIndex=1;
95
96    /**
97     * Get or create a node for an atom.
98     *
99     * @param atom the atom the required node should have
100    * @return a node with the required atom - if the node is not part of the graph yet,
101    * it will be created and inserted into the graph
102    */
103   public Node getNode(RelationalAtom atom) {
104    for(GeneralNode n1: this.graph) {
105     if(n1 instanceof Node) {
106      Node n=(terminationtest.GraphUtilities.Node)n1;
107      if(n.getAtom().getName().equals(atom.getName())&&(n.getAtom().getAdornments().equals(
           atom.getAdornments()))) return n;
108     }
109    }
110    Node node=new Node(atom);
111    node.index=universalIndex;
112    ++universalIndex;
113    this.graph.add(node);
114    return node;
115   }
116
117   /**
118    * The {@code getNode()} method gives no explanation if a new node was created. This
         method won't return
119    * the required node, but only checks if it already exists in the graph.
120    *
121    * @param atom the atom the required node should have
```

```
122     * @return {@code true} if a node with the atom is part of the graph, {@code false}
            otherwise
123     */
124    public boolean hasNode(RelationalAtom atom) {
125     for(GeneralNode n1: this.graph) {
126      if(n1 instanceof Node) {
127       Node n=(terminationtest.GraphUtilities.Node)n1;
128       if(n.getAtom().getName().equals(atom.getName())&&(n.getAtom().getAdornments().equals(
              atom.getAdornments()))) return true;
129      }
130     }
131     return false;
132    }
133
134    /**
135     * Inner class for abstract nodes.
136     */
137    public abstract class GeneralNode{
138     private ArrayList<GeneralNode>linksTo=new ArrayList<GeneralNode>();;
139
140     /**
141      * Creates an edge to an other node.
142      *
143      * @param node the node the edge is targeted at
144      */
145     public void addLink(GeneralNode node) {
146      if(!this.linksTo.contains(node))this.linksTo.add(node);
147     }
148
149     /**
150      * @return all nodes reachable with a single edge
151      */
152     public ArrayList<GeneralNode> getLinksTo() {
153      return linksTo;
154     }
155    }
156
157    /**
158     * Inner class for nodes containing atoms.
159     */
160    public class Node extends GeneralNode{
161     private RelationalAtom atom;
162     private Tgd creator=null;
163     public int index=0;
164
165     /**
166      * constructor
167      */
168     public Node(RelationalAtom atom) {
169      this.atom=atom;
170     }
171
172     public RelationalAtom getAtom() {
173      return atom;
174     }
175
176     /**
177      * @return the String representation of the nodes's atom including its adornments
178      */
179     public String toString() {
180      return index+": "+ConstraintRewriting.stringifyAtom(atom);
```

```
181    }
182
183    public Tgd getCreator() {
184     return creator;
185    }
186
187    public void setCreator(Tgd creator) {
188     this.creator = creator;
189    }
190
191    @Override
192    public boolean equals(Object o)
193    {
194     if(o instanceof Node) {
195      if(((Node)o).getAtom().getAdornments().equals(this.getAtom().getAdornments())&&
196        ((Node)o).getAtom().getName().equals(this.getAtom().getName()))return true;
197     }
198     return false;
199    }
200   }
201  }
```

Listing A.5: Klasse GraphUtilities

```
1  package terminationtest;
2
3  import java.io.BufferedReader;
4  import java.io.File;
5  import java.io.FileReader;
6  import java.io.IOException;
7  import java.util.ArrayList;
8  import java.util.HashSet;
9  import java.util.Set;
10 import atom.EqualityAtom;
11 import atom.RelationalAtom;
12 import integrityConstraint.Egd;
13 import integrityConstraint.IntegrityConstraint;
14 import integrityConstraint.Tgd;
15 import term.Term;
16 import term.Variable;
17 import term.VariableType;
18
19 /**
20  * This class provides a simple way to create integrity constraints from Strings (or text
        files).
21  * The format of the Strings could be Implication:
22  * R(x,y) -> S(x,Z)
23  * or Datalog (not actual datalog):
24  * S(x,Z):-R(x,y)
25  * Constraints end with a line break.
26  * Capitalized variables are existentially quantified  (but they don't have to be
       capitalized).
27  * Constants are e.g. 4, 4.2 or "forty-two".
28  * You can name a rule by adding a colon after the name:
29  * r1: R(x):-S(y).
30  * Thinks like space characters, tabs, empty lines etc. usually don't matter.
31  * Comments start with % ,@ or /* (but no multiline-comments are recognized).
32  *
33  * @author Andreas Goerres
34  */
35 public class Parser {
36
```

```
37  /**
38   * constructor
39   */
40  public Parser() {
41  }
42
43  /**
44   * Parse from a txt-file using the Implication format.
45   *
46   * @param filename the filename (including the path)
47   * @return the set of integrity constraints defined in the file
48   */
49  public static Set<IntegrityConstraint> parseFromFile(String filename) {
50   return parse(readFile(filename));
51  }
52
53  /**
54   * Parse from a txt-file using the Datalog format.
55   *
56   * @param filename the filename (including the path)
57   * @return the set of integrity constraints defined in the file
58   */
59  public static Set<IntegrityConstraint> parseDatalogFromFile(String filename) {
60   return parse(readFile(filename), true);
61  }
62
63  /**
64   * Parse from a txt-file using the Implication format, but generates a list of
         constraints.
65   *
66   * @param filename the filename (including the path)
67   * @return the list of integrity constraints defined in the file
68   */
69  public static Set<Tgd> parseFromFile2(String filename) {
70   return ConstraintRewriting.extractTGDs( parse(readFile(filename)));
71  }
72
73  /**
74   * Parse from a txt-file using the Datalog format, but generates a list of constraints.
75   *
76   * @param filename the filename (including the path)
77   * @return the list of integrity constraints defined in the file
78   */
79  public static Set<Tgd> parseDatalogFromFile2(String filename) {
80   return ConstraintRewriting.extractTGDs( parse(readFile(filename), true));
81  }
82
83  /**
84   * Parse from a String. If no second argument is given, Implication is the default
         format
85   *
86   * @param queryset the String containing constraints (divided by line breaks)
87   * @return the set of integrity constraints defined in the String
88   */
89  public static Set<IntegrityConstraint> parse(String queryset) {
90   return  parse(queryset, false);
91  }
92
93  /**
94   * Parse from a String.
95   *
```

```
96    * @param queryset the String containing constraints (divided by line breaks)
97    * @param datalog variable is {@code true} if the format of the String is Datalog,
98    * {@code false} otherwise
99    * @return the set of integrity constraints defined in the String
100   */
101  public static Set<IntegrityConstraint> parse(String queryset, boolean datalog) {
102   Set<IntegrityConstraint>tgds=new HashSet<IntegrityConstraint>();
103
104   String[]queries=queryset.split(System.lineSeparator());
105   int counter=0;
106   for(String query:queries) {
107    if(query.trim().isEmpty()||query.startsWith("%")||query.startsWith("@")||query.
         startsWith("/*"))continue;
108    ++counter;
109    HashSet<String>bodyvariables=new HashSet<String>();
110    String[] spl=null;
111    if(!datalog)spl=query.split("->");
112    else {
113     if(query.contains(":-"))spl=query.split(":-");
114     else {
115      spl= new String[2];
116      spl[0]=query;
117      spl[1]="";
118     }
119    }
120    String leftside="";
121    if(!datalog) leftside=spl[0].trim();
122    else leftside=spl[1].trim();
123    if(leftside.contains(":"))leftside=leftside.substring(leftside.indexOf(":")+1);
124    if(leftside.endsWith("."))leftside=leftside.substring(0, leftside.lastIndexOf("."));
125    String rightside="";
126    if(!datalog) rightside=spl[1].trim();
127    else rightside=spl[0].trim();
128    boolean isEGD=true;
129    if(rightside.contains("=")) {
130     isEGD=true;
131    }
132    else isEGD=false;
133    if(rightside.contains(":"))rightside=rightside.substring(rightside.indexOf(":")+1);
134    HashSet<RelationalAtom>links=new HashSet<RelationalAtom>();
135    HashSet<RelationalAtom>rechts2=new HashSet<RelationalAtom>();
136    HashSet<EqualityAtom>rechts3=new HashSet<EqualityAtom>();
137    String[]spl1=leftside.split("\\),");
138    String[]spl2=null;
139      if(isEGD)spl2=rightside.split(",");
140      else spl2=rightside.split("\\),");
141    for(String function1:spl1) {
142     String[]function11=function1.split("\\(");
143     String functionhead=function11[0].trim();
144     ArrayList<Term>terms=new ArrayList<Term>();
145     if(function11.length>1) {
146      if(function11[1].contains(")"))function11[1]=function11[1].substring(0, function11
           [1].lastIndexOf(")"));
147      String[]parameter=function11[1].split(",");
148      for(String p: parameter) {
149       if(p.trim().length()>0) {
150        Term t=null;
151        try {
152         int i=Integer.parseInt(p.trim());
153         t=new Term(i);
154        }catch(NumberFormatException e) {
```

```
155           try {
156            double j=Double.parseDouble(p.trim());
157            t=new Term(j);
158           }catch(NumberFormatException e2) {
159            if(p.startsWith("\"")) {
160             t=new Term(p.trim().replaceAll("\"", ""));
161            }
162            else if(p.toUpperCase().equals(p)) {
163             Variable v=new Variable(VariableType.E, p.trim(), counter);
164             t=new Term(v);
165            }
166            else {
167             bodyvariables.add(p.trim());
168             Variable v=new Variable(VariableType.V, p.trim(), counter);
169             t=new Term(v);
170            }
171           }
172          }
173         terms.add(t);
174         }
175        }
176       }
177       RelationalAtom ra=new RelationalAtom(functionhead, terms);
178       if(!functionhead.isEmpty())links.add(ra);
179      }
180      if(isEGD) {
181       for(String s:spl2) {
182        String[] parameter=s.split("=");
183        if(parameter.length==2) {
184         Term t=null;
185         try {
186          int i=Integer.parseInt(parameter[0].trim());
187          t=new Term(i);
188         }catch(NumberFormatException e) {
189          try {
190           double j=Double.parseDouble(parameter[0].trim());
191           t=new Term(j);
192          }catch(NumberFormatException e2) {
193           if(parameter[0].startsWith("\"")) {
194            t=new Term(parameter[0].trim().replaceAll("\"", ""));
195           }
196           else if((parameter[0].toUpperCase().equals(parameter[0]))||(!bodyvariables.
                  contains(parameter[0].trim()))) {
197            Variable v=new Variable(VariableType.E, parameter[0].trim(), counter);
198            t=new Term(v);
199           }
200           else {
201            Variable v=new Variable(VariableType.V, parameter[0].trim(), counter);
202            t=new Term(v);
203           }
204          }
205         }
206         Term t2=null;
207         try {
208          int i=Integer.parseInt(parameter[1].trim());
209          t2=new Term(i);
210         }catch(NumberFormatException e) {
211          try {
212           double j=Double.parseDouble(parameter[1].trim());
213           t2=new Term(j);
214          }catch(NumberFormatException e2) {
```

```
215          if(parameter[1].startsWith("\"")) {
216           t2=new Term(parameter[1].trim().replaceAll("\"", ""));
217          }
218          else if((parameter[1].toUpperCase().equals(parameter[1]))||(!bodyvariables.
                 contains(parameter[1].trim()))) {
219           Variable v=new Variable(VariableType.E, parameter[1].trim(), counter);
220           t2=new Term(v);
221          }
222          else {
223           Variable v=new Variable(VariableType.V, parameter[1].trim(), counter);
224           t2=new Term(v);
225          }
226         }
227        }
228        EqualityAtom ea =new EqualityAtom(t, t2);
229        if((t!=null)&&(t2!=null))rechts3.add(ea);
230       }
231      }
232     }
233     else {
234      for(String function1:spl2) {
235       String[]function11=function1.split("\\(");
236       String functionhead=function11[0].trim();
237       ArrayList<Term>terms=new ArrayList<Term>();
238       if(function11.length>1) {
239        if(function11[1].contains(")")) {
240         function11[1]=function11[1].substring(0, function11[1].lastIndexOf(")"));
241        }
242        if(function11[1].trim().length()>0) {
243         String[]parameter=function11[1].split(",");
244         for(String p: parameter) {
245          Term t=null;
246          try {
247           int i=Integer.parseInt(p.trim());
248           t=new Term(i);
249          }catch(NumberFormatException e) {
250           try {
251            double j=Double.parseDouble(p.trim());
252            t=new Term(j);
253           }catch(NumberFormatException e2) {
254            if(p.startsWith("\"")) {
255             t=new Term(p.trim().replaceAll("\"", ""));
256            }
257            else if((p.toUpperCase().equals(p))||(!bodyvariables.contains(p.trim()))) {
258             Variable v=new Variable(VariableType.E, p.trim(), counter);
259             t=new Term(v);
260            }
261            else {
262             Variable v=new Variable(VariableType.V, p.trim(), counter);
263             t=new Term(v);
264            }
265           }
266          }
267          terms.add(t);
268         }
269        }
270       }
271       RelationalAtom ra=new RelationalAtom(functionhead, terms);
272       if(!functionhead.isEmpty())rechts2.add(ra);
273      }
274     }
```

```
275     if(!isEGD)tgds.add(new Tgd(links, rechts2));
276     else tgds.add(new Egd(links, rechts3));
277    }
278    return tgds;
279   }
280
281   public static void showConstraints(Set<IntegrityConstraint>constraints) {
282    for(IntegrityConstraint t: constraints) {
283     System.out.println(t+System.lineSeparator());
284    }
285   }
286
287   /**
288    * Utility function that reads a file.
289    *
290    * @param filename path and name of the text file
291    * @return the content of the file as a String.
292    */
293   public static String readFile(String filename) {
294    BufferedReader breader=null;
295    String txt="";
296    try {
297     FileReader reader=new FileReader(new File(filename));
298     breader=new BufferedReader(reader);
299     String line="";
300     while((line=breader.readLine())!=null) {
301      txt+=line.trim()+" "+System.lineSeparator();
302     }
303    } catch (IOException e) {
304     e.printStackTrace();
305    }
306    finally{
307     try {
308      breader.close();
309     } catch (IOException e) {
310      e.printStackTrace();
311     }
312    }
313    return txt;
314   }
315  }
```

Listing A.6: Klasse Parser

```
1  package terminationtest;
2
3  import java.util.ArrayList;
4
5  /**
6   * This class represents a position in a relation. Therefore, objects of the class are
        characterized by the
7   * relation (that is, its name) and one of the relation's attributes (that is, its index
        number). Positions
8   * are part of a directed graph and can have to types of edges to other positions: normal
         edges and special
9   * edges. Normal edges copy value already present in the database into a newly created
        tuple, whereas special edges (might)
10  * (might) generate fresh null values in a newly created tuple.
11  * Positions are used not only for the weak acyclicity criterion, but also for rich
        acyclicity and safety. For
12  * the safety criterion, we differentiate between "affected" positions (positions that
        can carry null values) and
```

```
13  * "unaffected" null values (positions that can never carry null values).
14  *
15  * @author Andreas Goerres
16  */
17 public class Position {
18
19  private String relationName;
20  private int index;
21  private ArrayList<Position>edges=new ArrayList<Position>();
22  private ArrayList<Position>specialedges=new ArrayList<Position>();
23  private boolean affected=false;
24
25  /**
26   * @return can this position carry a null-value? Needed for the safety criterion, not
         for WA
27   */
28  public boolean isAffected() {
29   return affected;
30  }
31
32  /**
33   * @param affected makes the position affected ({@code true}) or unaffected ({@code
         false})
34   */
35  public void setAffected(boolean affected) {
36   this.affected = affected;
37  }
38
39  /**
40   * @return the name of the position's relation
41   */
42  public String getRelationName() {
43   return relationName;
44  }
45
46  public void setRelationName(String relationName) {
47   this.relationName = relationName;
48  }
49
50  /**
51   * @return the position's index (in a relation)
52   */
53  public int getIndex() {
54   return index;
55  }
56
57  /**
58   * @param index the index to be set
59   */
60  public void setIndex(int index) {
61   this.index = index;
62  }
63
64  /**
65   * @return the positions reachable with a single normal edge
66   */
67  public ArrayList<Position> getEdges() {
68   return edges;
69  }
70
71  /**
```

```
72     * @param edges the edges to be set
73     */
74    public void setEdges(ArrayList<Position> edges) {
75     this.edges = edges;
76    }
77
78    /**
79     * @return the positions reachable with a single special (generating) edge
80     */
81    public ArrayList<Position> getSpecialedges() {
82     return specialedges;
83    }
84
85    /**
86     * @param specialedges the special edges to be set
87     */
88    public void setSpecialedges(ArrayList<Position> specialedges) {
89     this.specialedges = specialedges;
90    }
91
92    /**
93     * Adds a normal edge to the node.
94     *
95     * @param p target position of the special edge
96     */
97    public void addEdges(Position p) {
98     if(!this.edges.contains(p))this.edges.add(p);
99    }
100
101   /**
102    * Adds a special edge to the node.
103    *
104    * @param p target position of the special edge
105    */
106   public void addSpecialedges(Position p) {
107    if(!this.specialedges.contains(p))this.specialedges.add(p);
108   }
109
110   /**
111    * Used for the safety criterion
112    */
113   public void removeAllEdges() {
114    this.edges=new ArrayList<Position>();
115    this.specialedges=new ArrayList<Position>();
116   }
117
118   /**
119    * Constructor
120    *
121    * @param relationName name from the atom used to create the position
122    * @param index the position of the attribute in the term-arraylist of the atom
123    */
124   public Position(String relationName, int index) {
125    this.relationName=relationName;
126    this.index=index;
127   }
128  }
```

Listing A.7: Klasse Parser

```
1  package terminationtest;
2
```

```
3  import java.util.ArrayList;
4  import java.util.HashMap;
5  import java.util.HashSet;
6  import java.util.Set;
7  import atom.EqualityAtom;
8  import atom.RelationalAtom;
9  import integrityConstraint.Egd;
10 import integrityConstraint.IntegrityConstraint;
11 import integrityConstraint.Tgd;
12 import term.Term;
13 import term.TermType;
14 import term.Variable;
15 import term.VariableType;
16
17 /**
18  * Most of the powerful termination tests (e.g. constraint-rewriting) work on TGDs only. If you want to test a
19  * set of integrity constraints containing both TGDs and EGDs, you can use the Substitution-free-Simulation (SfS)
20  * Algorithm to transforms the heterogeneous set of constraints into a set of TGDs with very similar behavior.
21  * To start this EGD-rewriting, call {@code SfS.doSfS(integrityconstraints)} with your {@code integrityconstraints}.
22  * Warning: EGDs might break an infinity loop of chase steps and thereby cause the termination of the chase. SfS
23  * does not simulate this behavior. However, if EGDs prevent the termination of the chase, chase on the rewritten
24  * set of TGDs won't terminate either.
25  * Warning: SfS does change the rewritten constraints - if you want to perform other tests on the original set
26  * of constraints or if you want to performe the actual chase, you should save those constraints or reload them.
27  *
28  *  @author Andreas Goerres
29  */
30 public class SfS {
31
32  /**
33   * Utility function that gives you the highest index number of all variables in a constraint, so you
34   * can introduce new variables without repeating yourself.
35   *
36   * @param constraint the EGD or TGD to be analyzed
37   * @return the highest index number of all variables in the constraint
38   */
39  public static int getMaxIndex(IntegrityConstraint constraint) {
40   int max=0;
41   for(RelationalAtom atom: constraint.getBody()) {
42    for(Term t: atom.getTerms()) {
43     if(t.getTermType()==TermType.Variable) {
44      Variable var=t.getTermValue();
45      if(var.getIndex()>max)max=var.getIndex();
46     }
47    }
48   }
49   if(constraint instanceof Tgd) {
50    Tgd tgd =(Tgd)constraint;
51    for(RelationalAtom atom: tgd.getHead()) {
52     for(Term t: atom.getTerms()) {
53      if(t.getTermType()==TermType.Variable) {
54       Variable var=t.getTermValue();
```

```
55          if(var.getIndex()>max)max=var.getIndex();
56         }
57        }
58       }
59      }
60      if(constraint instanceof Egd) {
61       Egd egd =(Egd)constraint;
62       for(EqualityAtom atom: egd.getHead()) {
63        Term t1= atom.getTerm1();
64        if(t1.getTermType()==TermType.Variable) {
65         Variable var1=t1.getTermValue();
66         if(var1.getIndex()>max)max=var1.getIndex();
67        }
68        Term t2= atom.getTerm2();
69        if(t2.getTermType()==TermType.Variable) {
70         Variable var2=t2.getTermValue();
71         if(var2.getIndex()>max)max=var2.getIndex();
72        }
73       }
74      }
75      return max;
76     }
77
78     /**
79      * We replace all EGDs with head t1=t2 by TGDs with head Eq(t1, t2) and hope that nobody
80      * will ever use "Eq" to name an actual table in the database.
81      *
82      * @param constraints set of integrity constraints to be rewritten
83      * @return rewritten set of TGDs
84      */
85     public static Set<Tgd> constraintsToTgds(Set<IntegrityConstraint> constraints){
86      Set<Tgd>tgds=new HashSet<Tgd>();
87      for(IntegrityConstraint constraint: constraints) {
88       tgds.add(constraintToTgd(constraint));
89      }
90      return tgds;
91     }
92
93     /**
94      * @param integrity constraint to be rewritten
95      * @return rewritten TGD
96      */
97     public static Tgd constraintToTgd(IntegrityConstraint constraint) {
98      if(constraint instanceof Tgd) return (Tgd)constraint;
99      Tgd tgd=null;
100     if(constraint instanceof Egd) {
101      Egd egd=(Egd)constraint;
102      HashSet<RelationalAtom>newhead=new HashSet<RelationalAtom>();
103      HashSet<EqualityAtom>oldhead=egd.getHead();
104      for(EqualityAtom oldheadatom: oldhead) {
105       ArrayList<Term>newheadterms=new ArrayList<Term>();
106       newheadterms.add(oldheadatom.getTerm1());
107       newheadterms.add(oldheadatom.getTerm2());
108       RelationalAtom newheadpart=new RelationalAtom("Eq", newheadterms);
109       newhead.add(newheadpart);
110      }
111      tgd=new Tgd(egd.getBody(), newhead);
112     }
113      return tgd;
114     }
115
```

```
116  /**
117   * Singularized integrity constraints substitute repeated variables (e.g. R(x,x)) and
          constants with fresh
118   * variables that are set equal to the original variable or constant. "Set equal" means
          they are added
119   * to the Eq-Relation:
120   * R(x,x,1) => R(x,y,z), Eq(x,y), Eq(1,z).
121   *
122   * @param constraints the integrity constraints to be singularized
123   * @return the singularized set of tgds
124   */
125  public static Set<Tgd>singularize(Set<IntegrityConstraint> constraints){
126   Set<Tgd>tgds=constraintsToTgds(constraints);
127   Set<Tgd>result=new HashSet<Tgd>();
128   for(Tgd t:tgds) {
129    result.add(singularization(t));
130   }
131   return result;
132  }
133
134  public static Tgd singularization(Tgd tgd) {
135   int maxindex=getMaxIndex(tgd);
136   for(RelationalAtom atom: tgd.getBody()) {
137    ArrayList<Term>newTerms=new ArrayList<Term>();
138    for(Term t: atom.getTerms()) {
139     if(t.getTermType()==TermType.Const) {
140      maxindex=maxindex+1;
141      Variable vc=new Variable(VariableType.V, t.toString(), maxindex);
142      Term t2=new Term(vc);
143      newTerms.add(t2);
144      ArrayList<Term>eqterms=new ArrayList<Term>();
145      eqterms.add(t);
146      eqterms.add(t2);
147      RelationalAtom eqatom=new RelationalAtom("Eq", eqterms);
148      tgd.addToBody(eqatom);
149     }
150     else newTerms.add(t);
151    }
152    atom.setTerms(newTerms);
153   }
154   Tgd tgd2=new Tgd(tgd.getBody(), tgd.getHead());
155   boolean notfirst=false;
156   ArrayList<RelationalAtom>zwischenspeicher=new ArrayList<RelationalAtom>();
157   for(RelationalAtom a0: tgd.getBody()) {
158    for(Term t0: a0.getTerms()) {
159     if(t0.getTermType()==TermType.Variable) {
160      Variable var0=t0.getTermValue();
161      notfirst=false;
162      for(RelationalAtom atom1: tgd2.getBody()) {
163       // we don't need to singularize the Eq-Atoms we just created using singularization
164       if(!atom1.getName().equals("Eq")) {
165        ArrayList<Term>newTerms=new ArrayList<Term>();
166        for(Term t1: atom1.getTerms()) {
167         if(t0.equals(t1)) {
168          if(notfirst) {
169           maxindex=maxindex+1;
170           Variable newVar=new Variable(var0.getVariableType(), var0.getIndexName(),
                 maxindex);
171           Term newTerm=new Term(newVar);
172           newTerms.add(newTerm);
173
```

```
174             ArrayList<Term>eqterms=new ArrayList<Term>();
175             eqterms.add(t0);
176             eqterms.add(newTerm);
177             RelationalAtom eqatom=new RelationalAtom("Eq", eqterms);
178             zwischenspeicher.add(eqatom);
179            }
180            else {
181             newTerms.add(t1);
182             notfirst=true;
183            }
184           }
185           else newTerms.add(t1);
186
187         }
188         atom1.setTerms(newTerms);
189        }
190       }
191      }
192     }
193    }
194    for(RelationalAtom a: zwischenspeicher) {
195     tgd2.addToBody(a);
196    }
197    return tgd2;
198   }
199
200
201   /**
202    * The Eq-relation is an equivalence relation, so it is reflexive, symmetric and
          transitive.
203    *
204    * @param constraints2 the integrity constraints to be rewritten
205    * @return rewritten set of TGDs
206    */
207   public static Set<Tgd> getEqAx(Set<IntegrityConstraint>constraints2){
208    Set<Tgd>constraints=constraintsToTgds(constraints2);
209    Set<Tgd>newconstraints=new HashSet<Tgd>();
210    Variable x1=new Variable(VariableType.V, "x", 1);
211    Variable y1=new Variable(VariableType.V, "y", 1);
212    Variable x2=new Variable(VariableType.V, "x", 2);
213    Variable y2=new Variable(VariableType.V, "y", 2);
214    Variable z2=new Variable(VariableType.V, "z", 2);
215    Term x_1=new Term(x1);
216    Term y_1 =new Term(y1);
217    Term x_2= new Term(x2);
218    Term y_2= new Term(y2);
219    Term z_2= new Term(z2);
220    ArrayList<Term> body1=new ArrayList<Term>();
221    body1.add(x_1);
222    body1.add(y_1);
223    HashSet<RelationalAtom>b1=new HashSet<RelationalAtom>();
224    b1.add(new RelationalAtom("Eq", body1));
225    ArrayList<Term> head1=new ArrayList<Term>();
226    head1.add(y_1);
227    head1.add(x_1);
228    HashSet<RelationalAtom>h1=new HashSet<RelationalAtom>();
229    h1.add(new RelationalAtom("Eq", head1));
230    Tgd symmetry =new Tgd(b1, h1);
231    ArrayList<Term> body2a=new ArrayList<Term>();
232    body2a.add(x_2);
233    body2a.add(y_2);
```

```
234     ArrayList<Term> body2b=new ArrayList<Term>();
235     body2b.add(y_2);
236     body2b.add(z_2);
237     HashSet<RelationalAtom>b2=new HashSet<RelationalAtom>();
238     b2.add(new RelationalAtom("Eq", body2a));
239     b2.add(new RelationalAtom("Eq", body2b));
240     ArrayList<Term> head2=new ArrayList<Term>();
241     head2.add(x_2);
242     head2.add(z_2);
243     HashSet<RelationalAtom>h2=new HashSet<RelationalAtom>();
244     h2.add(new RelationalAtom("Eq", head2));
245     Tgd transitivity =new Tgd(b2, h2);
246     newconstraints.add(symmetry);
247     newconstraints.add(transitivity);
248     HashMap<String, Integer>relations=new HashMap<String, Integer>();
249     for(IntegrityConstraint constr: constraints) {
250      for(RelationalAtom ra: constr.getBody()) {
251       relations.put(ra.getName(), ra.getTerms().size());
252      }
253      if(constr instanceof Tgd) {
254       Tgd t=(Tgd)constr;
255       for(RelationalAtom ra: t.getHead()) {
256        relations.put(ra.getName(), ra.getTerms().size());
257       }
258      }
259     }
260
261     Set<String>relnames=relations.keySet();
262     for(String relname: relnames) {
263      Tgd tgd =new Tgd(new HashSet<RelationalAtom>(), new HashSet<RelationalAtom>());
264      ArrayList<Term> body=new ArrayList<Term>();
265      int n=relations.get(relname);
266      for(int i=0;i<n;++i) {
267       Variable v=new Variable(VariableType.V, "v", i);
268       body.add(new Term(v));
269       ArrayList<Term> head=new ArrayList<Term>();
270       head.add(new Term(v));
271       head.add(new Term(v));
272       tgd.addToHead(new RelationalAtom("Eq", head));
273      }
274      tgd.addToBody(new RelationalAtom(relname, body));
275      newconstraints.add(tgd);
276     }
277     return newconstraints;
278    }
279
280    /**
281     * Static method that performs Simulation-free Substitution.
282     *
283     * @param constraints the set of integrity constraints to be rewritten
284     * @return return the rewritten set of TGDs
285     */
286    public static Set<Tgd>doSfS(Set<IntegrityConstraint>constraints){
287     Set<Tgd>set1=getEqAx(constraints);
288     Set<Tgd>set2=singularize(constraints);
289     set1.addAll(set2);
290     showSfS(set1);
291     return set1;
292    }
293
294    public static void showSfS(Set<Tgd>tgds) {
```

```
295     for(IntegrityConstraint c: tgds) {
296      System.out.println(c+System.lineSeparator());
297     }
298    }
299   }
```

Listing A.8: Klasse SfS

```
1   package terminationtest;
2
3
4   import java.awt.Dimension;
5   import java.awt.FlowLayout;
6   import java.awt.event.ActionEvent;
7   import java.awt.event.ActionListener;
8   import java.io.File;
9   import java.util.ArrayList;
10  import java.util.LinkedHashSet;
11  import java.util.Set;
12  import javax.swing.Box;
13  import javax.swing.JButton;
14  import javax.swing.JCheckBox;
15  import javax.swing.JFileChooser;
16  import javax.swing.JFrame;
17  import javax.swing.JLabel;
18  import javax.swing.JMenu;
19  import javax.swing.JMenuBar;
20  import javax.swing.JMenuItem;
21  import javax.swing.JPanel;
22  import javax.swing.JScrollPane;
23  import javax.swing.JTextPane;
24  import javax.swing.filechooser.FileNameExtensionFilter;
25  import chase.Chase;
26  import instance.Instance;
27  import instance.OriginTag;
28  import instance.Query;
29  import integrityConstraint.IntegrityConstraint;
30  import io.ChaseLog;
31  import io.InputReader;
32  import io.OutputWriter;
33  import io.SingleInput;
34
35  /**
36   *
37   * @author Andreas Goerres
38   */
39  public class TerminationTestGUI {
40    private File datei=null;
41    private Set<IntegrityConstraint>inputconstraints=null;
42    private SingleInput input;
43    private ChaseLog log = new ChaseLog();
44   //inputTypes: ChaTEAU (xml), Implication (txt, body -> head), Datalog (txt, head :- body
         )
45    private String inputType="ChaTEAU";
46
47   public static void main(String[] args) {
48    new TerminationTestGUI().darstellen();
49   }
50
51   /**
52    * Creates the GUI.
53    */
```

```
54  public  void darstellen(){
55   JFrame f=new JFrame("ChaTEAU");
56   f.setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
57   JTextPane jtp1 = new JTextPane();
58   jtp1.setText("");
59   JScrollPane scroll1 = new JScrollPane(jtp1);
60   JTextPane jtp2 = new JTextPane();
61   jtp1.setPreferredSize(new Dimension(300, 200));
62   jtp2.setText("");
63   jtp2.setPreferredSize(new Dimension(300, 200));
64   JScrollPane scroll2 = new JScrollPane(jtp2);
65   JPanel pan = new JPanel();
66   JPanel buttonbar=new JPanel(( new java.awt.FlowLayout() ));
67   JMenuBar menubar=new JMenuBar();
68   JMenu file=new JMenu("File");
69   JMenu options=new JMenu("Test Options");
70   menubar.add(file);
71   menubar.add(options);
72   JMenuItem openXML=new JMenuItem("Open ChaTEAU-File (.xml)");
73   JMenuItem openImplication=new JMenuItem("Open Implication-File (.txt)");
74   JMenuItem openDatalog=new JMenuItem("Open Datalog-File (.txt)");
75   file.add(openXML);
76   file.add(openImplication);
77   file.add(openDatalog);
78   JCheckBox RA=new JCheckBox("Rich Acyclicity");
79   JCheckBox WA=new JCheckBox("Weak Acyclicity");
80   JCheckBox SC=new JCheckBox("Safety");
81   JCheckBox AC=new JCheckBox("Acyclicity");
82   JCheckBox Sf=new JCheckBox("Acyclicity/EGD-Rewriting");
83   options.add(RA);
84   options.add(WA);
85  WA.setSelected(true);
86   options.add(SC);
87   options.add(AC);
88   options.add(Sf);
89   pan.setLayout( new javax.swing.BoxLayout(
90                 pan, javax.swing.BoxLayout.Y_AXIS ) );
91   JButton but = new JButton("Termination Test");
92   JButton but2 = new JButton("CHASE");
93   JLabel inputlabel=new JLabel("Input");
94   JLabel outputlabel=new JLabel("Output");
95   inputlabel.setLayout(new FlowLayout(FlowLayout.CENTER, 0, 0));
96   outputlabel.setLayout(new FlowLayout(FlowLayout.CENTER, 0, 0));
97   f.add(pan);
98   pan.add(menubar);
99   pan.add(inputlabel);
100  pan.add(scroll1);
101  pan.add(outputlabel);
102  pan.add(scroll2);
103  buttonbar.add(but);
104  buttonbar.add(but2);
105  buttonbar.add(Box.createHorizontalGlue());
106  pan.add(buttonbar);
107  f.setJMenuBar(menubar);
108  but.setEnabled(false);
109  but2.setEnabled(false);
110  f.setVisible(true);
111  f.pack();
112  f.setSize(new Dimension(500, 500));
113
114  // action listener of the TEST-button
```

```
115     but.addActionListener(new ActionListener() {
116      @Override
117      public void actionPerformed(ActionEvent arg0) {
118       //perform Test on TGDs
119       String result="Test results"+System.lineSeparator()+System.lineSeparator();
120       if(input!=null)inputconstraints=input.constraints;
121       if(inputconstraints==null)return;
122       ArrayList<Long>ra_values=new ArrayList<Long>();
123       ArrayList<Long>wa_values=new ArrayList<Long>();
124       ArrayList<Long>sc_values=new ArrayList<Long>();
125       ArrayList<Long>ac_values=new ArrayList<Long>();
126       ArrayList<Long>sf_values=new ArrayList<Long>();
127       // use the loop started by the following line for statistical tests
128       //for(int counter=0;counter<10;++counter) {
129       boolean nothingSelected=true;
130       if(RA.isSelected()) {
131        nothingSelected=false;
132        Long time1=System.currentTimeMillis();
133        boolean ra=new WA().checkRA(inputconstraints);
134        time1=System.currentTimeMillis()-time1;
135        ra_values.add(Long.valueOf(time1));
136        if(ra) result+="TGDs are not richly acyclic, Naive Oblivious Chase might not
               terminate. Time: "+time1+" ms"+System.lineSeparator();
137             else result+="TGDs are richly acyclic, even the Naive Oblivious Chase will
                    definitely terminate. Time: "+time1+" ms"+System.lineSeparator();
138       }
139       if(WA.isSelected()) {
140        nothingSelected=false;
141        Long time1=System.currentTimeMillis();
142           boolean wa=new WA().checkWA(inputconstraints);
143        time1=System.currentTimeMillis()-time1;
144        wa_values.add(Long.valueOf(time1));
145           if(wa)result+="TGDs are not weakly acyclic, Standard-Chase might not terminate.
                Time: "+time1+" ms"+System.lineSeparator();
146             else result+="TGDs are weakly acyclic, Standard-Chase will definitely
                    terminate. Time: "+time1+" ms"+System.lineSeparator();
147       }
148       if(SC.isSelected()) {
149        nothingSelected=false;
150        Long time1=System.currentTimeMillis();
151           boolean sc=new WA().checkSC(inputconstraints);
152        time1=System.currentTimeMillis()-time1;
153        sc_values.add(Long.valueOf(time1));
154           if(sc)result+="TGDs are not safe, Standard-Chase might not terminate. Time: "+
                time1+" ms"+System.lineSeparator();
155             else result+="TGDs are safe, Standard-Chase will definitely terminate. Time: "
                    +time1+" ms"+System.lineSeparator();
156       }
157       if(AC.isSelected()) {
158        nothingSelected=false;
159        Long time1=System.currentTimeMillis();
160        boolean ac=new ConstraintRewriting().prepareConstraintAdn(inputconstraints);
161        time1=System.currentTimeMillis()-time1;
162        ac_values.add(Long.valueOf(time1));
163        if(ac)result+="Constraint-Rewriting shows that TGDs are not acyclic. Standard-Chase
              will probably not terminate. Time: "+time1+" ms"+System.lineSeparator();
164             else result+="Constraint-Rewriting shows that TGDs are acyclic. Standard-Chase
                    will definitely terminate. Time: "+time1+" ms"+System.lineSeparator();
165       }
166       if(Sf.isSelected()) {
167        nothingSelected=false;
```

```
168       Long time1=System.currentTimeMillis();
169       boolean ac=new ConstraintRewriting().prepareAdn(SfS.doSfS(inputconstraints));
170       time1=System.currentTimeMillis()-time1;
171       sf_values.add(Long.valueOf(time1));
172       if(ac)result+="Constraint-Rewriting shows that TGDs/EGDs are not acyclic. Standard-
              Chase will probably not terminate. Time: "+time1+" ms"+System.lineSeparator();
173             else result+="Constraint-Rewriting shows that TGDs/EGDs are acyclic. Standard-
                    Chase will definitely terminate. Time: "+time1+" ms"+System.lineSeparator
                    ();
174
175             if(inputType.equals("ChaTEAU"))
176        try {
177         input = (new InputReader()).readFile(datei.getAbsolutePath());
178         inputconstraints=input.constraints;
179        } catch (Exception e) {
180         e.printStackTrace();
181        }
182       else if(inputType.equals("Implication"))inputconstraints=Parser.parseFromFile(datei.
              getAbsolutePath());
183             else if(inputType.equals("Datalog"))inputconstraints=Parser.
                    parseDatalogFromFile(datei.getAbsolutePath());
184     }
185
186     if(nothingSelected)result+="(Please select a termination criterion in the option menu
            !)";
187     // The following lines are part of the statistical tests, results are given in Latex
            code for a tabular
188     /*}
189     if(!ra_values.isEmpty())result+="&$"+getMean(ra_values)+" \\pm "+getStdDev(ra_values)
            +"$ ms "+System.lineSeparator();
190     if(!wa_values.isEmpty())result+="&$"+getMean(wa_values)+" \\pm "+getStdDev(wa_values)
            +"$ ms "+System.lineSeparator();
191     if(!sc_values.isEmpty())result+="&$"+getMean(sc_values)+" \\pm "+getStdDev(sc_values)
            +"$ ms "+System.lineSeparator();
192     if(!ac_values.isEmpty())result+="&$"+getMean(ac_values)+" \\pm "+getStdDev(ac_values)
            +"$ ms "+System.lineSeparator();
193     if(!sf_values.isEmpty())result+="&$"+getMean(sf_values)+" \\pm "+getStdDev(sf_values)
            +"$ ms "+System.lineSeparator();*/
194     jtp2.setText(result);
195     log.addEntry(result+System.lineSeparator()+System.lineSeparator());
196    }
197   });
198   // action listener for the CHASE-button
199   but2.addActionListener(new ActionListener() {
200    @Override
201    public void actionPerformed(ActionEvent ae) {
202     //perform Chase on Instanz/ Anfrage
203     jtp2.setText(chase());
204    }
205   });
206   // action listener for the open xml menu entry, includes test for st-tgd-consistency
207   openXML.addActionListener(new ActionListener() {
208    @Override
209    public void actionPerformed(ActionEvent ae) {
210     JFileChooser chooser = new JFileChooser();
211           FileNameExtensionFilter filter = new FileNameExtensionFilter(
212                   "XML Files", "xml");
213           chooser.setFileFilter(filter);
214           int returnVal = chooser.showOpenDialog(null);
215           if(returnVal!=0) return;
216           datei = chooser.getSelectedFile();
```

```
217         InputReader reader = new InputReader();
218             try {
219       input = reader.readFile(datei.getAbsolutePath());
220       inputType="ChaTEAU";
221       String result="";
222       Instance instance = input.intance;
223       LinkedHashSet<IntegrityConstraint> constraints = new LinkedHashSet<>();
224       for(IntegrityConstraint b : input.constraints)
225       {
226        constraints.add(b);
227       }
228       if(instance.getOriginTag() == OriginTag.i)
229       {
230        System.out.println("-Input (Instance):");
231        System.out.println(instance);
232              result+="Instance"+System.lineSeparator()+System.lineSeparator();
233              result+=instance+System.lineSeparator();
234       }
235       else
236       {
237        System.out.println("-Input (Query):");
238        System.out.println(instance.getQuery() + "\n");
239              result+="Query"+System.lineSeparator()+System.lineSeparator();
240              result+=instance.getQuery()+System.lineSeparator()+System.lineSeparator();
241       }
242       System.out.println("-Constraints:");
243       input.constraints.forEach(c -> System.out.println(c.getType() + ":\r\n" + c + "\r\n"
              ));
244             result+="Integrity Constraints"+System.lineSeparator()+System.lineSeparator();
245             for(IntegrityConstraint c: input.constraints) {
246        result+=c.getType()+": "+System.lineSeparator()+c + System.lineSeparator()+System.
              lineSeparator();
247       }
248             log.addEntry("Reading from " + datei.getAbsolutePath() + "\r\n");
249             log.addEntry("Check Constraints for incorrectly defined S-T TGDs. \r\n");
250             if(!instance.ConstraintCheck(constraints))
251             {
252              log.addEntry("Incorrectly defined S-T TGDs. (Check Source = S and Target = T
                  \"tag\" in schema..relation) \r\n");
253              log.addEntry("Stop without start the CHASE. \r\n");
254              System.out.println("-Output (Query):");
255              System.out.println("Incorrectly defined S-T TGDs. (Check Source = S and
                  Target = T \"tag\" in schema..relation) \r\n "
256                + "Stop without start the CHASE.");
257        jtp2.setText("Incorrectly defined S-T TGDs. (Check Source = S and Target = T \"tag
              \" in schema..relation) \r\n "
258                + "Stop without start the CHASE.");
259              but.setEnabled(false);
260        but2.setEnabled(false);
261             }
262             else
263             {
264              log.addEntry("Check succeeded. \r\n");
265              but.setEnabled(true);
266        but2.setEnabled(true);
267        jtp2.setText("");
268             }
269       jtp1.setText(result);
270       log = new ChaseLog();
271      } catch (Exception e) {
272       e.printStackTrace();
```

```
273       }
274      }
275     });
276     // action listener for the open Implication menu entry
277     openImplication.addActionListener(new ActionListener() {
278      @Override
279      public void actionPerformed(ActionEvent ae) {
280       JFileChooser chooser = new JFileChooser();
281             FileNameExtensionFilter filter = new FileNameExtensionFilter(
282                     "TXT Files", "txt");
283             chooser.setFileFilter(filter);
284             int returnVal = chooser.showOpenDialog(null);
285             if(returnVal!=0) return;
286             datei = chooser.getSelectedFile();
287             String result="";
288             inputconstraints=Parser.parseFromFile(datei.getAbsolutePath());
289          result+="Integrity Constraints"+System.lineSeparator()+System.lineSeparator();
290          inputconstraints.forEach(c -> System.out.println(c + "\r\n"));
291       for(IntegrityConstraint c: inputconstraints) {
292        result+=c + System.lineSeparator()+System.lineSeparator();
293       }
294       System.out.println("");
295       jtp1.setText(result);
296       jtp2.setText("");
297             input=null;
298             inputType="Implication";
299       but.setEnabled(true);
300       but2.setEnabled(false);
301       log = new ChaseLog();
302      }
303     });
304     // action listener for the open Datalog menu entry
305     openDatalog.addActionListener(new ActionListener() {
306      @Override
307      public void actionPerformed(ActionEvent ae) {
308       JFileChooser chooser = new JFileChooser();
309             FileNameExtensionFilter filter = new FileNameExtensionFilter(
310                     "TXT Files", "txt");
311             chooser.setFileFilter(filter);
312             int returnVal = chooser.showOpenDialog(null);
313             if(returnVal!=0) return;
314             datei = chooser.getSelectedFile();
315             String result="";
316             inputconstraints=Parser.parseDatalogFromFile(datei.getAbsolutePath());
317             result+="Integrity Constraints"+System.lineSeparator()+System.lineSeparator();
318             inputconstraints.forEach(c -> System.out.println(c + "\r\n"));
319       for(IntegrityConstraint c: inputconstraints) {
320        result+=c + System.lineSeparator()+System.lineSeparator();
321       }
322       System.out.println("");
323       jtp1.setText(result);
324       jtp2.setText("");
325             input=null;
326             inputType="Datalog";
327       but.setEnabled(true);
328       but2.setEnabled(false);
329       log = new ChaseLog();
330      }
331     });
332    }
333
```

```
334  /**
335   * Performs the actual chase. Code was copied from Jakob Zimmer.
336   *
337   * @return String representation of the Chase result
338   */
339  public String chase() {
340  OutputWriter writer = new OutputWriter();
341  String result="";
342  try
343  {
344   String folderPath = datei.getParent();
345         String name = datei.getName().substring(0, datei.getName().lastIndexOf('.'));
346         LinkedHashSet<IntegrityConstraint> constraints = new LinkedHashSet<>();
347      for(IntegrityConstraint b : input.constraints)
348      {
349       constraints.add(b);
350      }
351      //CHASE anwenden
352      Instance i = Chase.chase(input.intance, constraints, log);
353         log.addEntry("\r\nWriting Output to " + folderPath + "\\" + name + "Result.xml" +
               " and " + folderPath + "\\" + name + "Result.csv");
354         log.addEntry("Done");
355         if(i.equals(input.intance))
356         {
357          log.addEntry("\r\nThe CHASE yielded no new results");
358          result+="The CHASE yielded no new results"+System.lineSeparator()+System.
               lineSeparator();
359         }
360         writer.writeLog(log, folderPath + "\\" + name + "Log.txt");
361         if(i.getOriginTag() == OriginTag.i)
362         {
363          writer.writeXML(i, folderPath + "\\" + name + "Result.xml");
364          writer.writeCSV(i, folderPath + "\\" + name + "Result.csv");
365
366          System.out.println("-Output (Instance):");
367          System.out.println(i);
368          result+="Instance"+System.lineSeparator()+System.lineSeparator();
369          result+=i;
370         }
371         else if(i.getOriginTag() == OriginTag.q)
372         {
373          //wenn Instanz aus einer Query entstand, dann zurueckformen
374       Query resultQuery = i.getQuery();
375       writer.writeXML(i, resultQuery, folderPath + "\\" + name + "Result.xml");
376          writer.writeCSV(i, resultQuery, folderPath + "\\" + name + "Result.csv");
377          System.out.println("-Output (Query):");
378          System.out.println(resultQuery);
379          result+="Query"+System.lineSeparator()+System.lineSeparator();
380          result+=resultQuery;
381         }
382  } catch (Exception e)
383  {
384   e.printStackTrace();
385  }
386  return result;
387 }
388
389  /**
390   * Calculates the arithmetical mean of a sample of Long values.
391   *
392   * @param values the sample of Long values
```

```
393     * @return the arithmetical mean
394     */
395    public static double getMean(ArrayList<Long>values) {
396     double mean=0;
397     for(long v: values) {
398      mean+=(double)v;
399     }
400     return mean/(double)values.size();
401    }
402
403    /**
404     * Estimates the standard deviation of a population based on a sample.
405     *
406     * @param values the sample of Long values used to calculate the standard deviation
407     * @return the standard deviation
408     */
409    public static double getStdDev(ArrayList<Long>values) {
410     double mean=getMean(values);
411     double stddev=0;
412     for(long v: values) {
413      stddev+=((mean-v)*(mean-v));
414     }
415     return Math.sqrt(stddev/((double)values.size()-1.0));
416    }
417   }
```

Listing A.9: Klasse TerminationTestGUI

```
1   package term;
2
3   import terminationtest.Adornment;
4
5   /**
6    * The class defines terms that can either be Variables, Null values or Constants.
7    * In the case of Constants a term is represented by a basic data type. Variables and
         Nulls
8    * are represented and output by strings following the pattern
9    *
10   * <ul>
11   * <li>{@code #V_indexName_index} or {@code #E_indexName_index} for Variables and
12   * <li>{@code #N_indexName_index} for Nulls.
13   * </ul>
14   *
15   * The underscores are included in the strings. The {@code indexName} represents the name
16   * of the attribute for that the Variable or Null stands for. The {@code index} is the
         number
17   * of the corresponding Variable or Null.
18   *
19   * @author Martin Jurklies
20   */
21
22  public class Term
23  {
24   // type of the term
25   private TermType termType;
26
27   // type of the Constant (only has a value if termType == TermType.Const)
28   private ConstType constType;
29
30   // values of Constant terms
31   private int termValueInt;
32   private double termValueDouble;
```

```
33   private String termValueString;
34
35   // value of the term if it is a Null
36   private Null termValueNull;
37
38   // value of the term if it is a Variable
39   private Variable termValueVariable;
40
41   // adornments of terms and atoms are used for the constrain-rewriting termination check
42   private Adornment adornment=new Adornment();
43
44   /**
45    * Constructor based on another term. Variables will be the variables of the other term,
         not duplicates
46    */
47   public Term(Term term) {
48    this.termType=term.termType;
49    this.adornment=term.adornment;
50    if(this.termType==TermType.Const) {
51     this.constType=term.constType;
52     if(this.constType==ConstType.String) {
53      this.termValueString=term.termValueString;
54     }
55     else if(this.constType==ConstType.Double) {
56      this.termValueDouble=term.termValueDouble;
57     }
58     else if(this.constType==ConstType.Int) {
59      this.termValueInt=term.termValueInt;
60     }
61    }
62    else if(this.termType==TermType.Variable) {
63     this.termValueVariable=term.termValueVariable;
64    }
65    else if(this.termType==TermType.Null) {
66     this.termValueNull=term.termValueNull;
67    }
68   }
69
70   /**
71    * Constructor with an integer as Constant value.
72    */
73   public Term(int termValue)
74   {
75    this.termType = TermType.Const;
76    this.constType = ConstType.Int;
77    this.termValueInt = termValue;
78   }
79
80   /**
81    * Constructor with a double as Constant value.
82    */
83   public Term(double termValue)
84   {
85    this.termType = TermType.Const;
86    this.constType = ConstType.Double;
87    this.termValueDouble = termValue;
88   }
89
90   /**
91    * Constructor with either
92    *
```

```
93    *  <ul>
94    *  <li>a string as Constant value,
95    *  <li>a Variable represented as a string starting with "#V" or "#E" or
96    *  <li>a Null value represented as a string starting with "#N".
97    *  </ul>
98    */
99   public Term(String termValue)
100  {
101   if (termValue.startsWith("#N"))
102   {
103    int iFirst_ = termValue.indexOf("_");
104    int iSecond_ = termValue.substring(iFirst_ + 1).indexOf("_");
105    int nullIndex = Integer.parseInt(termValue.substring(iFirst_ + iSecond_ + 2)); //TODO
         try catch
106
107    this.termType = TermType.Null;
108
109    String nullName = termValue.substring(iFirst_ + 1, iFirst_ + iSecond_ + 1);
110    Null newNull = new Null(nullName, nullIndex);
111
112    this.termValueNull = newNull;
113   }
114
115   else if (termValue.startsWith("#V"))
116   {
117    int iFirst_ = termValue.indexOf("_");
118    int iSecond_ = termValue.substring(iFirst_ + 1).indexOf("_");
119    int variableIndex = Integer.parseInt(termValue.substring(iFirst_ + iSecond_ + 2)); //
         TODO try catch
120
121    this.termType = TermType.Variable;
122
123    String variableName = termValue.substring(iFirst_ + 1, iFirst_ + iSecond_ + 1);
124    Variable newVariable = new Variable(VariableType.V, variableName, variableIndex);
125
126    this.setTermValueVariable(newVariable);
127   }
128
129   else if (termValue.startsWith("#E"))
130   {
131    int iFirst_ = termValue.indexOf("_");
132    int iSecond_ = termValue.substring(iFirst_ + 1).indexOf("_");
133    int variableIndex = Integer.parseInt(termValue.substring(iFirst_ + iSecond_ + 2)); //
         TODO try catch
134
135    this.termType = TermType.Variable;
136
137    String variableName = termValue.substring(iFirst_ + 1, iFirst_ + iSecond_ + 1);
138    Variable newVariable = new Variable(VariableType.E, variableName, variableIndex);
139
140    this.setTermValueVariable(newVariable);
141   }
142
143   else
144   {
145    this.termType = TermType.Const;
146    this.constType = ConstType.String;
147    this.termValueString = termValue;
148   }
149  }
150
```

```
151   /**
152    * constructor
153    */
154   public Term(Null termValue)
155   {
156    this.termType = TermType.Null;
157    this.termValueNull = termValue;
158   }
159
160   /**
161    * constructor
162    */
163   public Term(Variable termValue)
164   {
165    this.termType = TermType.Variable;
166    this.setTermValueVariable(termValue);
167   }
168
169   /**
170    * @return the term type
171    */
172   public TermType getTermType()
173   {
174    return this.termType;
175   }
176
177   /**
178    * @param termType the term type to set
179    */
180   public void setTermType(TermType termType)
181   {
182    this.termType = termType;
183   }
184
185   /**
186    * @return the Constant type
187    */
188   public ConstType getConstType()
189   {
190    return this.constType;
191   }
192
193   /**
194    * @param constType the Constant type to set
195    */
196   public void setConstType(ConstType constType)
197   {
198    this.constType = constType;
199   }
200
201   /**
202    * @return the Adornment
203    */
204   public Adornment getAdornment() {
205    return adornment;
206   }
207
208   /**
209    * @param adornment the Adornment to set
210    */
211   public void setAdornment(Adornment adornment) {
```

```
212     this.adornment = adornment;
213   }
214
215   /**
216    * @return the term value dependent on the term type
217    */
218   @SuppressWarnings("unchecked")
219   public <Any> Any getTermValue()
220   {
221     // termType == Const
222     if (this.termType == TermType.Const)
223     {
224       // constType == Int
225       if (this.constType == ConstType.Int)
226       {
227         return (Any)(Integer)(int) this.termValueInt;
228       }
229
230       // constType == Double
231       else if (this.constType == ConstType.Double)
232       {
233         return (Any)(Double)(double) this.termValueDouble;
234       }
235
236       // constType == String
237       else
238       {
239         return (Any)(String) this.termValueString;
240       }
241     }
242
243     // termType == Null
244     else if (this.termType == TermType.Null)
245     {
246       return (Any)(Null) this.termValueNull;
247     }
248
249     // termType == Variable
250     else
251     {
252       return (Any)(Variable) this.getTermValueVariable();
253     }
254   }
255
256   /**
257    * Tests whether some other term "has a homomorphism to" this term. There can only be a
         homomorphism
258    * from term {@code t1} to term {@code t2}, also called "mapping" in the context of
         terms, if
259    *
260    * <ul>
261    * <li>{@code t1} is an existential quantified Variable,
262    * <li>{@code t1} is a given Variable and {@code t2} is the same Variable as {@code t1}
         or a Null or Constant,
263    * <li>{@code t1} is a Null value and {@code t2} is no Variable,
264    * <li>{@code t1} and {@code t2} are the same Constants.
265    * </ul>
266    *
267    * @param other the other term to where a homomorphism should exist
268    * @return {@code true} if there is a homomorphism from this term to the other, {@code
         false} otherwise
```

```
269     */
270    public boolean hasHomomorphismTo(Term other)
271    {
272     // "this" is a Variable
273     if (this.termType == TermType.Variable)
274     {
275      // "this" is an existential quantified Variable (#E)
276      if (this.termValueVariable.getVariableType() == VariableType.E)
277      {
278       if(other.getTermType() == TermType.Null)
279       {
280        return false;
281       }
282       else
283       {
284        return true;
285       }
286      }
287
288      // "this" is a given Variable (#V)
289      else
290      {
291       // "other" is a Variable
292       if (other.termType == TermType.Variable)
293       {
294        // true: "other" is the same Variable as "this"
295        // false: "this" and "other" are different Variables
296        return other.termValueVariable.equals(this.termValueVariable);
297       }
298
299       // "other" is a Null
300       else if (other.termType == TermType.Null)
301       {
302        return false;
303       }
304
305       // "other" is a Constant
306       else
307       {
308        return true;
309       }
310      }
311     }
312
313     // "this" is a Null
314     else if (this.termType == TermType.Null)
315     {
316      // true: "other" is not a Variable
317      // false: "other" is a Variable
318      return other.termType != TermType.Variable;
319     }
320
321     // "this" is a Constant
322     else
323     {
324      // both terms have not the same constType
325      if (this.constType != other.getConstType())
326      {
327       return false;
328      }
329
```

```
330     else
331     {
332      // both terms are Integers
333      if (this.constType == ConstType.Int)
334      {
335       return this.termValueInt == other.termValueInt;
336      }
337
338      // both terms are Doubles
339      else if (this.constType == ConstType.Double)
340      {
341       return this.termValueDouble == other.termValueDouble;
342      }
343
344      // both terms are Strings
345      else
346      {
347       return this.termValueString.equals(other.termValueString);
348      }
349     }
350    }
351   }
352
353
354   /* (non-Javadoc)
355    * @see java.lang.Object#toString()
356    */
357   @Override
358   public String toString()
359   {
360    StringBuilder builder = new StringBuilder();
361
362    // termType == Const
363    if (this.termType == TermType.Const)
364    {
365     // constType == Int
366     if (this.constType == ConstType.Int)
367     {
368      builder.append(this.termValueInt);
369     }
370
371     // constType == Double
372     else if (this.constType == ConstType.Double)
373     {
374      builder.append(this.termValueDouble);
375     }
376
377     // constType == String
378     else
379     {
380      builder.append(this.termValueString);
381     }
382    }
383
384    // termType == Null
385    else if (this.termType == TermType.Null)
386    {
387     builder.append(this.termValueNull.toString());
388    }
389
390    // termType == Variable
```

```
391     else
392     {
393      builder.append(this.getTermValueVariable().toString());
394     }
395
396     return builder.toString();
397    }
398
399    /* (non-Javadoc)
400     * @see java.lang.Object#hashCode()
401     */
402    @Override
403    public int hashCode()
404    {
405     final int prime = 31;
406     int result = 1;
407     result = prime * result + ((constType == null) ? 0 : constType.hashCode());
408     result = prime * result + ((termType == null) ? 0 : termType.hashCode());
409     long temp;
410     temp = Double.doubleToLongBits(termValueDouble);
411     result = prime * result + (int) (temp ^ (temp >>> 32));
412     result = prime * result + termValueInt;
413     result = prime * result + ((termValueNull == null) ? 0 : termValueNull.hashCode());
414     result = prime * result + ((termValueString == null) ? 0 : termValueString.hashCode());
415     result = prime * result + ((getTermValueVariable() == null) ? 0 : getTermValueVariable
            ().hashCode());
416     return result;
417    }
418
419    /* (non-Javadoc)
420     * @see java.lang.Object#equals(java.lang.Object)
421     */
422    @Override
423    public boolean equals(Object obj)
424    {
425     if (this == obj)
426     {
427      return true;
428     }
429
430     if (obj == null)
431     {
432      return false;
433     }
434
435     if (!(obj instanceof Term))
436     {
437      return false;
438     }
439
440     Term other = (Term) obj;
441
442     if (constType != other.constType)
443     {
444      return false;
445     }
446
447     if (termType != other.termType)
448     {
449      return false;
450     }
```

```
451
452   if (Double.doubleToLongBits(termValueDouble) != Double.doubleToLongBits(other.
          termValueDouble))
453   {
454    return false;
455   }
456
457   if (termValueInt != other.termValueInt)
458   {
459    return false;
460   }
461
462   if (termValueNull == null)
463   {
464    if (other.termValueNull != null)
465    {
466     return false;
467    }
468   }
469
470   else if (!termValueNull.equals(other.termValueNull))
471   {
472    return false;
473   }
474
475   if (termValueString == null)
476   {
477    if (other.termValueString != null)
478    {
479     return false;
480    }
481   }
482
483   else if (!termValueString.equals(other.termValueString))
484   {
485    return false;
486   }
487
488   if (getTermValueVariable() == null)
489   {
490    if (other.getTermValueVariable() != null)
491    {
492     return false;
493    }
494   }
495
496   else if (!getTermValueVariable().equals(other.getTermValueVariable()))
497   {
498    return false;
499   }
500
501   return true;
502  }
503
504  public Variable getTermValueVariable() {
505   return termValueVariable;
506  }
507
508  public Null getTermValueNull()
509  {
510   return termValueNull;
```

```
511   }
512
513   public void setTermValueVariable(Variable termValueVariable) {
514     this.termValueVariable = termValueVariable;
515   }
516 }
```

Listing A.10: Klasse Term

```
1  package atom;
2
3  import java.util.ArrayList;
4
5  import term.Term;
6  import terminationtest.Adornment;
7
8  /**
9   * The class defines relational atoms and consists of a relation name
10  * and a list of {@link term.Term} objects.
11  *
12  * @author Martin Jurklies
13  */
14 public class RelationalAtom extends Atom
15 {
16   private String relationName;
17   private ArrayList<Term> terms;
18   private ArrayList<Adornment>adornments;
19
20   /**
21    * empty constructor
22    */
23   public RelationalAtom()
24   {
25     this(null, null);
26   }
27
28   /**
29    * copy constructor
30    *
31    * @param atom the relational atom to copy
32    */
33   public RelationalAtom(RelationalAtom atom)
34   {
35     this(atom.getName(), atom.terms);
36   }
37
38   /**
39    * constructor
40    */
41   public RelationalAtom(String name)
42   {
43     this(name, null);
44   }
45
46   /**
47    * constructor
48    */
49   public RelationalAtom(String relationName, ArrayList<Term> terms)
50   {
51     if (relationName == null)
52     {
53       relationName = "";
```

```
54    }
55
56    if (terms == null)
57    {
58     terms = new ArrayList<Term>();
59    }
60
61    this.relationName = relationName;
62    this.terms = new ArrayList<Term>(terms);
63   }
64
65   /**
66    * @return the relation name
67    */
68   public String getName()
69   {
70    return this.getRelationName();
71   }
72
73   /**
74    * @param relationName the relation name to set
75    */
76   public void setName(String relationName)
77   {
78    this.relationName = relationName;
79   }
80
81   /**
82    * @return the list of terms
83    */
84   public ArrayList<Term> getTerms()
85   {
86    return this.terms;
87   }
88
89   /**
90    * @param terms the term list to set
91    */
92   public void setTerms(ArrayList<Term> terms)
93   {
94    this.terms = terms;
95   }
96
97   /**
98    * @return the list of adornments (corresponding to the list of terms)
99    */
100  public ArrayList<Adornment> getAdornments() {
101   return adornments;
102  }
103
104  /**
105   * @param adornments the adornment list to set
106   */
107  public void setAdornments(ArrayList<Adornment> adornments) {
108   this.adornments = adornments;
109  }
110
111  /**
112   * each Adornment in the adonments-list will be transfered to the corresponding term in
          the term-list
113   */
```

```
114   public void transferAdornments() {
115    if((this.adornments==null)||(this.terms==null)) return;
116    for(int i=0;i<this.terms.size();++i) {
117     if(i<this.adornments.size())this.getTerms().get(i).setAdornment(this.adornments.get(i)
          );
118    }
119   }
120
121   /**
122    * the adornments-list will be recreated with Adornments from terms of the term-list
123    */
124   public void receiveAdornments() {
125    if(this.terms==null) return;
126    this.setAdornments(new ArrayList<Adornment>());
127    for(int i=0;i<this.terms.size();++i) {
128     this.getAdornments().add(i, this.getTerms().get(i).getAdornment());
129    }
130   }
131
132   /**
133    * @return if Adornments from adornment-list are identical with Adornments from terms of
          the term-list
134    */
135   public boolean isConsistent() {
136    if((this.adornments==null)||(this.terms==null)) return true; // something is wrong, but
          the problem is not consistency
137    for(int i=0;i<this.terms.size();++i) {
138     if(i<this.adornments.size()) {
139      if(!this.getTerms().get(i).getAdornment().equals(this.adornments.get(i))) return
           false;
140     }
141    }
142    return true;
143   }
144
145   /**
146    * Checks if there exists a homomorphism from this object to
147    * another relational atom.
148    * @param other the other relational atom
149    * @return {@code true} if there is a homomorphism from this object to the other, {@code
          false} otherwise
150    */
151   public boolean hasHomomorphismTo(RelationalAtom other)
152   {
153    ArrayList<Term> thisTerms = this.terms;
154    ArrayList<Term> otherTerms = other.terms;
155
156    if (this.getRelationName().equals(other.getRelationName()) && thisTerms.size() ==
         otherTerms.size())
157    {
158     for (int i = 0; i < thisTerms.size(); i++)
159     {
160      if (thisTerms.get(i).hasHomomorphismTo(otherTerms.get(i)))
161      {
162       continue;
163      }
164
165      else
166      {
167       return false;
168      }
```

```
169     }
170
171     return true;
172    }
173
174    else
175    {
176     return false;
177    }
178   }
179
180
181   /* (non-Javadoc)
182    * @see java.lang.Object#toString()
183    */
184   @Override
185   public String toString()
186   {
187    StringBuilder builder = new StringBuilder();
188
189    String sepTerm = "";
190
191    builder.append(this.getRelationName());
192    builder.append("(");
193
194    ArrayList<Term> terms = this.terms;
195    int termSize = terms.size();
196
197    for (int i = 0; i < termSize; i++)
198    {
199     builder.append(sepTerm);
200     builder.append(terms.get(i).getTermValue().toString());
201     sepTerm = ", ";
202    }
203
204    builder.append(")");
205
206    return builder.toString();
207   }
208
209   /* (non-Javadoc)
210    * @see java.lang.Object#hashCode()
211    */
212   @Override
213   public int hashCode()
214   {
215    final int prime = 31;
216    int result = 1;
217    result = prime * result + ((getRelationName() == null) ? 0 : getRelationName().hashCode
           ());
218    result = prime * result + ((terms == null) ? 0 : terms.hashCode());
219    return result;
220   }
221
222   /* (non-Javadoc)
223    * @see java.lang.Object#equals(java.lang.Object)
224    */
225   @Override
226   public boolean equals(Object obj)
227   {
228    if (this == obj)
```

```
229     {
230      return true;
231     }
232 
233     if (obj == null)
234     {
235      return false;
236     }
237 
238     if (!(obj instanceof RelationalAtom))
239     {
240      return false;
241     }
242 
243     RelationalAtom other = (RelationalAtom) obj;
244 
245     if (getRelationName() == null)
246     {
247      if (other.getRelationName() != null)
248      {
249       return false;
250      }
251 
252     }
253 
254     else if (!getRelationName().equals(other.getRelationName()))
255     {
256      return false;
257     }
258 
259     if (terms == null)
260     {
261      if (other.terms != null)
262      {
263       return false;
264      }
265     }
266 
267     else if (!terms.equals(other.terms))
268     {
269      return false;
270     }
271 
272     return true;
273    }
274 
275    public String getRelationName() {
276     return relationName;
277    }
278 }
```

Listing A.11: Klasse RelationalAtom

```
1 package integrityConstraint;
2 
3 import java.util.HashSet;
4 
5 import atom.RelationalAtom;
6 import term.Term;
7 import term.TermType;
8 import term.Variable;
9 import term.VariableType;
```

```
10 import terminationtest.Adornment;
11
12 /**
13  * The class represents tgd's and inherits from {@link integrityConstraint.
        IntegrityConstraint}.
14  *
15  * @author Martin Jurklies
16  */
17 public class Tgd extends IntegrityConstraint
18 {
19  protected int index;
20
21  /**
22   * @return the tgd index
23   */
24  public int getIndex() {
25   return index;
26  }
27
28  /**
29   * @param index the tgd index to set
30   */
31  public void setIndex(int index) {
32   this.index = index;
33  }
34
35  /**
36   * @return the Adornments of all the universally quantified variables in the head
37   */
38  public HashSet<Adornment>getAdornmentsForSkolemization(){
39   HashSet<Adornment>skolemOf=new HashSet<Adornment>();
40   for(RelationalAtom a: this.getHead()) {
41    for(Term t: a.getTerms()) {
42     if(t.getTermType()==TermType.Variable) {
43      Variable v=(Variable)t.getTermValue();
44      if(v.getVariableType()==VariableType.V) skolemOf.add(t.getAdornment());
45     }
46    }
47   }
48   return skolemOf;
49  }
50
51  /**
52   * Existentially quantified variables (in ChaTEAU their terms) have an adornment that
        depends only on
53   * the adornments of the head's universally quantified variables. If the tgd is bodiless
54   * however, the adornment of other variables does not matter, it is simply b.
55   */
56  public void setAdornmentsForSkolemization(){
57   HashSet<Adornment>skolemOf=getAdornmentsForSkolemization();
58   for(RelationalAtom a: this.getHead()) {
59    for(Term t: a.getTerms()) {
60     if(t.getTermType()==TermType.Variable) {
61      Variable v=(Variable)t.getTermValue();
62      if(v.getVariableType()==VariableType.E) {
63       if(this.getBody().isEmpty())t.setAdornment(new Adornment());
64       else t.setAdornment(new Adornment(this.getIndex(), t.toString(), skolemOf));
65      }
66     }
67    }
68   }
```

```
69   }
70
71   /**
72    * constructor
73    */
74   public Tgd(HashSet<RelationalAtom> body, HashSet<RelationalAtom> head)
75   {
76    this.body = body;
77    this.head = head;
78   }
79
80   /**
81    * @return the tgd head
82    */
83   @SuppressWarnings("unchecked")
84   @Override
85   public HashSet<RelationalAtom> getHead()
86   {
87    return (HashSet<RelationalAtom>) this.head;
88   }
89
90   /**
91    * @param head the tgd head to set
92    *//*
93   public void setHead(HashSet<RelationalAtom> head)
94   {
95    this.head = head;
96   }*/
97
98   /**
99    * Adds a relational atom to the head of this tgd.
100  *
101  * @param atom the relational atom to add to the head
102  * @return {@code true} if {@code atom} was added to {@code head}, {@code false}
         otherwise
103  */
104  @SuppressWarnings("unchecked")
105  public boolean addToHead(RelationalAtom atom)
106  {
107   return ((HashSet<RelationalAtom>) this.head).add(atom);
108  }
109
110  /* (non-Javadoc)
111   * @see java.lang.Object#toString()
112   */
113  @SuppressWarnings("unchecked")
114  @Override
115  public String toString()
116  {
117   StringBuilder builder = new StringBuilder();
118   String sepAtom = "";
119
120   for (RelationalAtom atom : this.body)
121   {
122    builder.append(sepAtom);
123
124    builder.append(atom.toString());
125
126    sepAtom = ",\n";
127   }
128
```

```
129     builder.append("\n->\n");
130     sepAtom = "";
131
132     for (RelationalAtom atom : (HashSet<RelationalAtom>) this.head)
133     {
134       builder.append(sepAtom);
135
136       builder.append(atom.toString());
137
138       sepAtom = ",\n";
139     }
140
141     return builder.toString();
142   }
143 }
```

Listing A.12: Klasse Tgd

A.6. Aufbau des Datenträgers

Auf dem beigelegten Datenträger befinden sich die für diese Arbeit verwendeten Literaturquellen, das Programm ChaTEAU in drei verschiedenen Versionen, eine virtuelle Maschine, Testdateien sowie die Masterarbeit selbst. Im Folgenden wird der Aufbau des Datenträgers kurz beschrieben. Verwendete Überschriften entsprechen dabei den Ordnernamen im Wurzelverzeichnis des Datenträgers.

Literatur

Dieser Ordner enthält die verwendeten Literaturquellen in Form von PDF-Dokumenten. In der Arbeit zitierte Webseiten wurden ebenfalls als PDF-Dokumente bereitgestellt. In beiden Fällen entspricht der Dateiname dem im Literaturverzeichnis verwendeten Kürzel.
Hinweis: Die beigefügte Literatur dient der Nachvollziehbarkeit von Aussagen und darf nicht öffentlich bereitgestellt werden. Dies gilt speziell für Werke, welche durch Lizenzverträge der Universität Rostock durch die Universitätsbibliothek zur Verfügung gestellt wurden.

Masterarbeit

Dieser Ordner enthält die Masterarbeit als PDF-Dokument. Die bei der Anfertigung der Arbeit verwendeten TeX-Dateien sowie die BibTeX-Datei mit den Literaturquellen wurden ebenfalls bereitgestellt. Verwendete Bilder befinden sich im Unterordner „Bilder“.

Software

Dieser Ordner enthält drei Versionen des Programms ChaTEAU. Die erste Version im Ordner „Chateau_Renn“ enthält ChaTEAU in dem Zustand, der zu Beginn der Arbeit vorlag (letzte Änderung am 03.06.2019). Die zweite Version im Ordner „Chateau_Zimmer“ beinhaltet eine Ausführung von ChaTEAU, die parallel zur vorliegenden Arbeit im Rahmen der Bachelorarbeit von Jakob Zimmer erweitert wurde (letzte Änderung am 17.02.2020). Die dritte Version im Ordner „ChaTEAU_Goerres“ beinhaltet ChaTEAU in dem Zustand, in dem es sich zum Abschluss dieser Arbeit befindet. Von Jakob Zimmer vorgenommenen Erweiterungen wurden in diese dritte Version integriert. Alle drei Versionen liegen in Form von Eclipse-Projekten vor.

Virtuelle Maschine

Dieser Ordner enthält eine virtuelle Maschine im OVF-Format. Auf der virtuellen Maschine sind Llunatic 1.0.2, Llunatic 2.0, Graal, PDQ und ChaseTEQ installiert.

Testdateien

Dieser Ordner enthält die in Abschnitt 3.2 verwendeten Testdateien für Llunatic, Graal, PDQ und ChaseTEQ. Die Dateien befinden sich jeweils in einem Unterordner, der den Namen des entsprechenden Chase-Werkzeugs trägt. Der Unterordner „ChaTEAU“ enthält Testdateien für ChaTEAU, die unter anderem den Beispielen aus Unterabschnitt 3.1.1 und den Anwendungsfällen aus Abschnitt 5.1 entsprechen.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Masterarbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt 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 3.3.2020 _____