Universität Rostock Traditio et Innovatio

Bachelorarbeit

Erweiterung der ProSA-Pipeline um die Verarbeitung einzelner Schemaevolutionschritte

eingereicht von: Eric Maier

eingereicht am: 30.08.2022

Gutachter:innen: M. Sc. Tanja Auge
Dr.-Ing Holger Meyer

Zusammenfassung

Das Werkzeug ProSA, welches vom Lehrstuhl Datenbank- und Informationssysteme der Universität Rostock entwickelt wurde, bietet verschiedene Möglichkeiten zum Forschungsdatenmanagement an. Mit der Erstellung einer minimalen Teildatenbank soll die Reproduzierbarkeit von Forschungsergebnissen ermöglicht und gleichzeitig der Speicherbedarf reduziert werden. Dabei werden die Forschungsdatenbanken laufend mit neuen Daten erweitert und verändern sich in ihrer Struktur, wodurch die damalige Anfrage oft nicht die selben Ergebnisse liefert oder garnicht erst Ausführbar ist. Ziel dieser Arbeit ist Erweiterung der ProSA-Pipeline zur Implementierung von (Inverse-)Schemaevolutionen. Demnach sollen Änderungen an Datenbanken vorrübergehend Rückgängig gemacht werden, sodass wir die ursprüngliche Datenbank erhalten und die Anfrage erneut stellen können. Dazu sollen bereits implementierte Komponenten wie ChaTEAU und der Invertierer verwendet werden, um die Schemaevolutionsoperatoren anzuwenden und deren Inversen zu bilden. Vorherige Arbeiten der Universität Rostock bilden die Basis für die Grundlagen und das Konzept der Erweiterung.

Abstract

The tool ProSA, which was developed by the institute of database and information systems at the University of Rostock, offers various options for research data management. With the creation of a minimal sub-database we want to enhance the reproducibility of research results and reduce the storage requirements. At the same time, the research databases are constantly expanded with new data and change in their structure, which means that the query at that time often does not provide the same results or is not executable at all. The goal of this work is to extend the ProSA pipeline to implement (inverse) schema evolution. Accordingly, changes to databases should be undone temporarily, so that we can use the original database and process the request again. For this purpose already implemented components like ChaTEAU and the inverter will be used to execute the schema evolution operators and build their inverses. Previous work of the University of Rostock forms the basis for the fundamentals and the concept of the extension.

Inhaltsverzeichnis

1 Einleitung

Einleitend wollen wir klären, weshalb SMOs in ProSA integriert werden sollen und welches Ziel wir damit verfolgen. Außerdem wird ein Datensatz vorgestellt, an dem Beispiele vorgeführt werden können.

1.1 Motivation und Zielsetzung

Eine Aufgabe von ProSA ist die Reproduzierbarkeit von Anfrageergebnissen zu ermöglichen. Wird in ProSA eine Anfrage an eine Datenbank gestellt, so wird auf das Ergebnis die Inverse-Anfrage angewendet, um eine für die Anfrage passende minimale Teildatenbank zu generieren. Um die selbe Anfrage erneut zu stellen, benötigt man nur noch die Teildatenbank statt der ursprünglichen Datenbank. Dadurch wird der Speicherbedarf reduziert, während die Reproduzierbarkeit gewährleistet wird. Wenn wir jedoch eine Anfrage reproduzieren wollen, wofür keine minimale Teildatenbank existiert, dann muss die ursprüngliche Datenbank wiederhergestellt werden. Dazu wollen wir SMOs mit Hilfe eines geeigneten Aufrufes von ChaTEAU innerhalb von ProSA nutzen können. Da die Formulierungen als s-t tgds und deren Inversen bereits in [Aug22] ausführlich Vorgestellt wurden, wird nur noch ein Konzept und die Implementierung selbst benötigt. Dazu wollen wir die Datenbankkonfiguration um die Eingabe einer XML-File erweitern, welche eine oder mehrere SMOs als Menge von s-t tgds enthält. Die daraus resultierende Datenbankinstanz soll dann in einem separaten GUI-Tab ausgegeben werden.

1.2 Beispiel Datensatz

Im Folgenden werden vereinfachte Relationen aufgeführt, welche zur Veranschaulichung der Abhängigkeiten dienen sollen. Die Datensätze entstammen der in ProSA enthaltenen Beispiel Datenbank. Die Studenten Relation wurde um das Attribut Immatrikulation_Year erweitert, welches wir für ein späteres Beispiel benötigen werden. Die Participants Relation wurde aufgrund ihrer original Größe komprimiert.

MatricNo	LastName	FirstName	Course	Immatrikulation_Year
1	Fieber	Fabian	Computer Science for Lectureship	1999
2	Sonnenschein	Sarah	Mathematics	1999
3	Müller	Max	Electrical Engineering	2000
4	Müller	Mira	Computer Science	2001
5	Johansen	Johannes	Computer Science	2001
6	Miller	Mia	Computer Science	2001
7	Mustermann	Max	Electrical Engineering	2003
8	Johannes	Paul	ITTI	2004

1.2.1 Relation Students

ModuleNo	Title	Major
001	Databases III	Information Systems
002	Mathematics for Computer Scientists 1	
003	Pattern Recognition and Context Analysis	Smart Computing
004	Individual Knowledge Management	Information Systems
005	Data Warehouses and Business Intelligence	Business Informatics
006	Kognitive Systems	Smart Computing
007	Theory of relational Databases	Information Systems
008	Systems Biology	Models and Algorithms
009	New Developments in Computer Science	

1.2.2 Relation Modules

ModuleNo	MatricNo
001	1
001	4
001	5
002	4
002	5
002	7
003	1
004	3
004	2
005	4
005	7
006	4
006	5
007	1
007	3
007	5
008	3
009	1
009	4

1.2.3 Relation Participants

1.3 Aufbau der Arbeit

Vorerst wollen wir kurz den Aufbau dieser Arbeit und die Themen der einzelnen Kapitel betrachten.

Im 2. Kapitel sollen die Grundlagen für diese Arbeit vermittelt werden. Dazu werden im allgemeinen relationale Datenbanken und funktionale Abhängigkeiten betrachtet. Diese führen uns zum Chase-Algorithmus, welcher Abhängigkeiten in eine Instanz einbettet. Dazu werden egds, tgds und s-t tgds definiert und an Beispielen erklärt. Weiterhin werden Operatoren für Schemaevolutionen vorgestellt, welche sich auf Tabellen oder Spalten beziehen können.

Das 3. Kapitel beschäftigt sich mit dem aktuellen Stand der Technik. Dafür wird das Tool ChaTEAU vorgestellt, welches den Chase-Algorithmus implementiert. Dieser wird in ProSA für die Auswertung von Anfragen und zur Konstruktion der minimalen Teildatenbank benötigt. Anschließend wird der aktuelle Ablauf von ProSA grafisch dargestellt und beschrieben.

Im 4. Kapitel wird der aktuelle Stand der Forschung vorgestellt. Dazu werden SMOs definiert und deren Formulierung als s-t tgd vorgestellt.

Im 5. Kapitel wird das Konzept der Arbeit vorgestellt. Dazu wird die noch zu implementierende Eingabe von SMOs über XML-Files erklärt und Anhand von Beispielen gezeigt, worauf dabei zu achten ist. Darauffolgend wird beschrieben wie der Chase-Algorithmus die SMOs auf eine Datenbankinstanz anwenden soll und was wir als Ergebnis erwarten.

Das 6. Kapitel beschäftigt sich mit der eigentlichen Implementierung dieser Arbeit. Dazu wird die Eingabe und Konfiguration der SMOs implementiert. Anschließend sollen diese durch einen Aufruf von ChaTEAU angewendet werden und in einem eigenen Tab ausgegeben werden.

Letztendlich wird diese Arbeit kurz zusammengefasst und es werden mögliche Erweiterungen vorgestellt, welche in dieser Arbeit nicht umgesetzt werden konnten.

2 Grundlagen

Bevor wir uns dem aktuellen Stand der Technik (Kapitel 3) und der Forschung (Kapitel 4) widmen, wollen wir die theoretischen Grundlagen zusammentragen, welche für das Verständnis dieser Arbeit notwendig sind. Dazu werden allgemeine Kenntnisse zu Datenbanken (Siehe 2.1) benötigt, wobei wir uns relationale Datenbanken und funktionale Abhängigkeiten genauer anschauen werden, da diese essenziell für den Chase-Algorithmus (Siehe 2.2) sind. Dabei sind für uns vor allem die (s-t) tgds relevant, da diese für die Formalisierung von Schemaevolution verwendet werden sollen. Anschließend wollen wir klären, welche Operatoren es zur Darstellung von Schemaevolutionsschritten gibt und wofür diese verwendet werden (Siehe 2.3).

2.1 Datenbanken

Relationale Datenbanken

Als Eingabe für ProSA benötigen wir eine relationale Datenbank, welche eine Ansammlung von miteinander in Beziehung (Relation) stehenden Daten ist. Grundlegend sind die Relationsschemata, welche aus Mengen von Attributen bestehen und so gesehen Schablonen für die Relationen sind. Eine Relation wiederum besteht aus Tupeln mit Werten sowie einem Relationsnamen. In unserem Beispiel-Datensatz (Siehe Tabelle 1.2) besteht das Relationsschema aus den einzelnen Attributen `Student_ID`, `LastName`, `FirstName`, `Course` und `Immatrikulation_Year`, wobei `Student_ID` und `Immatrikulation_Year` der Wertebereich **integer** zugeordnet wird und den anderen Attributen der Wertebereich **string**. Im weiteren Verlauf der Arbeit, werden wir diese Wertebereiche auch als Domänen $D = \{D_1, ..., D_i | i \in \mathbb{N}\}$ bezeichnen.

Definition 2.1 (Relationsschemata und Relationen, nach [HSS18]). Eine Relation r über einem Relationsschema $R = \{A_1, ..., A_n\}$ mit $n \in \mathbb{N}^+$ ist eine endliche Menge von Abbildungen

$$t : R \rightarrow \bigcup_{i=1}^{\infty} D_i,$$

die Tupel t genannt werden. Damit ist unser Relationsschema definiert als

$$R_1 = \{\text{Student\_ID,LastName,FirstName,Course,Immatrikulation\_Year}\},$$

die Relation `Students` als $r_1 = \{t_1, ..., t_8\}$ und das erste Tupel als
$t_1 = (1,\text{Fieber,Fabian,Computer Science for Lectureship}, 1999)$.

Folglich können wir das Datenbankschema (Siehe Def. 2.2) und letztendlich die Datenbank (Siehe Def. 2.3) definieren.

Definition 2.2 (Datenbankschema, nach [HSS18]). Ein Datenbankschema ist eine Menge von Relationsschemata $\{R_1, ..., R_n\}$, für $n \in \mathbb{N}^+$.

Definition 2.3 (Datenbank, nach [HSS18]). Eine Datenbank ist eine Menge von Relationen $\{r_1, ..., r_n\}$ über einem Datenbankschema $S = \{R_1, ..., R_n\}$, wobei $r_i(R_i)$ für alle $1 \leq i \leq n$ gilt. Damit entstammt die Relation r_i dem Relationsschema R_i.

Funktionale Abhängigkeiten

Um Abhängigkeiten zwischen Attributen innerhalb einer Relation darzustellen, benutzen wir funktionale Abhängigkeiten (kurz: FDs, für `Functional Dependencys`), welche wir als $X \rightarrow Y$ bezeichnen. Dabei sind X und Y Attributmengen der Relation und der Implikationspfeil bedeutet X bestimmt Y funktional. Das heißt, wenn zwei Tupel innerhalb einer Relation die gleichen Werte in den Attributen aus X besitzen, dann haben diese auch die gleichen Werte für die Attribute aus Y. Als Beispiel definieren wir die funktionale Abhängigkeit:

$$\texttt{Student\_ID} \rightarrow \texttt{LastName}, \texttt{FirstName}, \texttt{Course}, \texttt{Immatrikulation\_Year}.$$

Angenommen wir hätten nun ein Tupel t_9 mit $t_9(\texttt{Student\_ID}) = 1$, dann folgt aus der funktionalen Abhängigkeit das ebenso gelten muss:

$$\begin{aligned} t_9(\texttt{LastName}) &= \text{Fieber}, \\ t_9(\texttt{FirstName}) &= \text{Fabian}, \\ t_9(\texttt{Course}) &= \text{Computer Science for Lectureship}, \\ t_9(\texttt{Immatrikulation\_Year}) &= 1999. \end{aligned}$$

Mit unserem Wissen über relationale Datenbanken und funktionalen Abhängigkeiten können wir uns nun dem Chase-Algorithmus widmen.

2.2 Chase-Algorithmus

Der Chase-Algorithmus wird in der Datenbankforschung verwendet, um Abhängigkeiten $\star$ in ein Objekt $\bigcirc$ einzuarbeiten:

$$Chase_\star(\bigcirc) = \circledast.$$

In ProSA ist das Objekt $\bigcirc$ eine Datenbankinstanz, welche Tupel aus Relationen enthält. Deshalb befassen wir uns lediglich mit dem Chase auf Instanzen. Die Abhängigkeiten $\star$ sind Mengen von eingebetteten Abhängigkeiten, welche wir wie folgt definieren:

Definition 2.4 (Eingebettete Abhängigkeiten, nach [GMS12]). Eine eingebettete Abhängigkeit (kurz: ED, für Embedded Dependency) ist ein logischer Ausdruck der Form:

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

wobei x, y, z Tupel von Variablen sind. Weiterhin sind $\phi(x, y)$ und $\psi(x, z)$ Konjunktionen über atomare Formeln, welche Rumpf und Kopf der Abhängigkeit genannt werden. Ein Spezialfall der ED ist die gleichheitserzeugende Abhängigkeit, welche genau ein Gleichheitsatom im Kopf besitzt.
Definition 2.5 (Gleichheitserzeugende Abhängigkeit (kurz: egd, für equality generating dependency), nach [GMS12]). Eine gleichheitserzeugende Abhängigkeit (egd) ist eine ED der Form:

$$\forall x : \phi(x) \rightarrow x_1 = x_2.$$

Diese werden im Chase-Algorithmus verwendet, um Nullwerte (η) funktional durch Konstanten oder andere Nullwerte zu ersetzen. Angenommen wir hätten nun ein Tupel $t_{10} = (6, \eta_1, \eta_2, \eta_3, \eta_4)$ in unserem Beispieldatensatz, dann können wir folgende egd (`Student_ID` bestimmt `LastName`) formulieren:

$$S(sid, ln_1, fn_1, c_1, imy_1) \wedge S(sid, ln_2, fn_2, c_2, imy_2) \rightarrow ln_1 = ln_2.$$

Analog verfahren wir mit `FirstName`,`Course` und `Immatrikulation_year`. Durch Anwendung dieser egds, würde das Tupel $t_{10} = (6,\text{Miller},\text{Mia},\text{Computer Science}, 2001)$ mit dem Tupel t_6 gleichgesetzt werden. Da der Chase mengenorientiert arbeitet, findet zudem eine automatische Duplikateliminierung statt.

Definition 2.6 (Tupelerzeugende Abhängigkeit (kurz tgd, für tuple generating dependency), nach [GMS12]). Eine tupelerzeugende Abhängigkeit (tgd) ist eine ED der Form:

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

Dabei ist $\psi(x)$ eine Konjunktion von atomaren Formeln mit Variablen aus x und $\psi(x, y)$ eine Konjunktion von atomaren Formeln mit Variablen aus x und y.
Wenn also ein Tupel existiert, welches in das Schema des Rumpfes $\phi(x)$ der tgd passt, wird ein für das Schema des Kopfes $\psi(x, y)$ passendes Tupel erzeugt. Dabei werden die existenzquantifizierten Variablen mit nummerierten Nullwerten η_n gefüllt. Zur Veranschaulichung folgendes Beispiel:

$$\forall ModuleNo, MatricNo : P(ModuleNo, MatricNo) \rightarrow \exists Title, Major : M(ModuleNo, Title, Major).$$

Diese tgd prüft für jedes Tupel der `Participants` Relation (Siehe 1.2), ob auch ein Tupel mit der selben `ModuleNo` in der `Modules` Relation (Siehe 1.2) existiert. Falls nicht, wird dieses Tupel erzeugt und mit Nullwerten für `Title` und `Major` aufgefüllt.

Schließlich können wir die s-t tgd definieren, welche für diese Arbeit die relevanteste Abhängigkeit sein wird.

Definition 2.7 (source-to-target tgd (kurz s-t tgd), nach [GMS12]). Eine s-t tgd ist eine tgd, bei der ϕ eine Konjunktion über einem Quellschema S und ψ eine Konjunktion über einem Zielschema T ist.

Statt nur neue Tupel zu erzeugen, werden bei einer s-t tgd Tupel aus dem Quellschema S in das Zielschema T überführt. Demnach existieren die Tupel aus dem Quellschema S nach Anwendung der s-t tgd nicht mehr. Angenommen wir wollen die `Student` Relation ohne die Attribute `LastName` und `FirstName` veröffentlichen. Dann können wir mit folgender s-t tgd alle relevanten Tupel der `Students` Relation in eine neue Relation `Students_Anonym` überführen:

$$\forall mano, ln, fn, c, imy : S(mano, ln, fn, c, imy) \rightarrow S_A(mano, c, imy).$$

Dies wäre bereits ein Beispiel des Schemaevolutionsoperators `DROP Column`, mit dem wir Attribute eines Relationsschemas entfernen können.
Nachdem wir nun die im Chase genutzten Abhängigkeiten kennen, müssen wir uns zum Verständnis des Algorithmus noch die Homomorphismen und Trigger aneignen.

Definition 2.8 (Homomorphismus für Instanzen, nach [Han22]). Seien I_1 und I_2 Instanzen über einem Schema A, $Const_i$ die Menge aller Konstanten und Var_i die Menge aller Variablen der jeweiligen Instanz I_i. Eine Funktion

$$h : Const_1 \cup Var_1 \rightarrow Const_2 \cup Var_2$$

heißt Homomorphismus genau dann, wenn:

- $\forall c \in Const_1 : h(c) = c$
- $\forall R \in A, \forall (a_1, ..., a_n) \in R^{I_1} : (h(a_1), ...h(a_n)) \in R^{I_2}$ (R^I : Instanz I über der Relation R)

$I_1 \rightarrow I_2$ beschreibt dann einen Homomorphismus von der Instanz I_1 nach I_2.

Definition 2.9 ((Aktiver) Trigger, nach [Sch21]). Sei I eine Instanz und $b \in B$ eine (s-t) tgd oder egd. Ein Trigger für b in I ist ein Homomorphismus $h : \phi(x) \rightarrow I$, wobei $\phi(x)$ den Rumpf von b darstellt. Ein Trigger gilt als aktiv genau dann, wenn gilt:

- Falls b eine (s-t) tgd ist, existiert keine Erweiterung von h zu einem Homomorphismus von $\psi(x, y) \rightarrow I$.
- Falls b eine egd ist, dann gilt $h(x_i) \neq h(x_j)$ für mindestens ein Gleichheitsatom $(x_i = x_j)$ im Kopf der egd.

Algorithm 1 Chase auf Instanzen mit angepasster Notation, aus [Han22]

```
for each Trigger t für eine Abhängigkeit g ∈ ⋆ do
    if t ist aktiv then
        if g hat Form ∀x : φ(x) → ∃y : ψ(x, y) (tgd) then
            ψ anwenden, neue Tupel in I aufnehmen
        else if g hat Form ∀x : φ(x) → x_1 = x_2 (egd) then
            if x_1, x_2 sind Konstanten und x_1 ≠ x_2 then
                Chase schlägt fehl
            else if x_1 ist Konstante then
                x_2 ← x_1
            else if x_2 ist Konstante oder x_1, x_2 sind Nullwerte then
                x_1 ← x_2
            end if
        end if
    end if
end for
```

Der Chase auf Instanzen wendet solange Homomorphismen an, bis keine aktiven Trigger t mehr existieren. Dabei wird für den Rumpf der Abhängigkeit g geprüft, ob dieser homomorph auf ein Atom der Instanz I abgebildet werden kann und somit einen Trigger t generiert. Wenn zudem kein Atom der Instanz I zum Kopf der Abhängigkeit g homomorph ist, ist der Trigger t aktiv und die die Abhängigkeit g kann in die Instanz I eingearbeitet werden. Als Beispiel durchlaufen wir den Algorithmus mit der bereits eingeführten s-t tgd, erweitert um die Konstante $S(course) =$ Mathematics:

$$\forall mano, ln, fn, imy : S(mano, ln, fn, \text{'Mathematics'}, imy) \rightarrow S_A(mano, \text{'Mathematics'}, imy)$$

Dazu muss dem Algorithmus das Quellschema S, das Zielschema S_A, die s-t tgd und der Datensatz als Instanz übergeben werden.

1. Existiert ein Homomorphismus zwischen dem Quellschema und einem Tupel der Instanz?
   Ja : S(2,Sonnenschein,Sarah,Mathematics,1999) → erstelle Trigger.
2. Würde durch Anwendung der Abhängigkeit g ein neues Tupel entstehen?
   Ja → der Trigger ist aktiv.
3. Ist die Abhängigkeit g eine tgd?
   Ja → wende den Kopf von g an und nehme neues Tupel in die Instanz auf.
4. Existiert ein Homomorphismus zwischen dem Quellschema und einem Tupel der Instanz?
   Ja : S(2,Sonnenschein,Sarah,Mathematics,1999) → erstelle Trigger.
5. Würde durch Anwendung der Abhängigkeit g ein neues Tupel entstehen?
   Nein → der Trigger ist nicht aktiv.
6. Existiert ein Homomorphismus zwischen dem Quellschema und einem Tupel der Instanz?
   Nein
7. Existiert eine weitere Abhängigkeit $g \in \star$?
   Nein
8. Der Chase Terminiert.

Somit erhalten wir eine Instanz mit lediglich einer Relation S(2,Mathematics,1999) und dem dazugehörigem Schema als Ausgabe des Chase.

2.3 Schemaevolution

Zunächst wollen wir die Schemavolution im allgemeinen kennen lernen und anschließend die jeweiligen Operatoren betrachten. Angenommen wir haben eine Menge von Datenbankschemata S_t zum Zeitpunkt t, dann können wir eine Schemavolution formalisieren als:

$$E = (S_t, S_{t+1}, \Sigma),$$

wobei Σ die funktionale Abhängigkeit ist, welche das Schema S_t zum Zeitpunkt t in das Schema S_{t+1} zum Zeitpunkt t+1 überführt. Ebenfalls können wir die Inverse Schemaevolution beschreiben als:

$$E^{-1} = (S_{t+1}, S_t, \Sigma^{-1}),$$

wobei Σ^{-1} die invertierte funktionale Abhängigkeit ist.

Mit Schemaevolutionsoperatoren (kurz SMO, für Schema Modification Operator) wollen wir Veränderungen an Datenbanken realisieren. Diese können auf Tabellen selbst oder auf Spalten einer Tabelle angewendet werden. Die gängigsten SMOs werden in folgender Tabelle aufgelistet und kurz beschrieben. Die jeweiligen Umformungen zu s-t tgds, welche wir für die Ausführung des Chase benötigen, werden dann in Kapitel 4 beschrieben.

SMO	**Beschreibung**
`COPY TABLE`	dupliziert eine existierende Tabelle
`CREATE TABLE`	erstellt eine neue, leere Tabelle
`DECOMPOSE TABLE`	anhand einer Quelltabelle, werden zwei neue Tabellen generiert
`DROP TABLE`	entfernt eine existierende Tabelle
`JOIN TABLE`	erstellt über dem Verbundattribut zweier Quelltabellen eine neue Tabelle
`MERGE TABLE`	nimmt zwei Quelltabellen mit dem selben Schema und erstellt eine Tabelle als Vereinigung
`PARTITION TABLE`	anhand einer Quelltabelle, werden zwei neue Tabellen generiert, welche bestimmte Bedingungen erfüllen
`RENAME TABLE`	ändert den Namen einer Tabelle
`ADD COLUMN`	erstellt für eine Tabelle eine neue Spalte
`COPY COLUMN`	kopiert eine Spalte einer Tabelle und fügt diese einer anderen Tabelle hinzu
`DROP COLUMN`	löscht eine Spalte aus einer Tabelle
`MOVE COLUMN`	siehe `COPY COLUMN` mit anschließendem löschen der Originalspalte
`RENAME COLUMN`	ändert den Namen einer Spalte

2.3.1 Schema Modification Operators (SMOs)
entnommen aus [Aug22]

Klassifikation relevanter SMOs

Um die Relevanz von den jeweiligen SMOs zu bestimmen, wurde in [CTMZ08] die Datenbankentwicklung von Wikipedia analysiert. Dabei wurde die Häufigkeit des Auftretens jedes SMO aus insgesamt 269 Evolutionen bestimmt. Daraus ergeben sich die Relevanten Operatoren als `ADD Column`(38.7%), `DROP Column`(26.4%), `RENAME Column`(16.0%), `CREATE Table`(8.9%) und `DROP Table`(3.3%). In der Arbeit [Man20] hingegen, wurde die Anzahl der Schema-Änderungen der Forschungsdatenbank des Leibniz-Institut für Ostseeforschung Warnemünde (IOW) untersucht. Das Resultat bestand dabei zu 75% aus `ADD Column` und zu 22% aus `DROP Column` Operatoren. Weiterhin wurden in den Arbeiten [Man20] und [Aug22] zwei neue Operatoren eingeführt, welche im Bereich des Forschungsdatenmanagements häufiger auftreten.

`MERGE Column`: Hierbei wird eine neue Spalte einer Tabelle funktional erstellt mit anschließenden löschen der "Quell" Spalten. Als Beispiel betrachten wir das verschmelzen der Spalten `firstName` und `lastName` aus Tabelle 1.2 zu einer Spalte, welche wir `fullName` nennen wollen:

1. `ADD Column`: erstellen der Spalte `fullName` funktional als:
   $func := CONCAT(firstName, ' ', lastName)$
2. `DROP Column`: löschen der Spalte `firstName`
3. `DROP Column`: löschen der Spalte `lastName`

`SPLIT Column`: Dieser Operator wird in [Man20] als inverse von `MERGE Column` definiert. Demnach werden mehrere Spalten funktional erstellt und anschließend die "Quell" Spalte gelöscht. Bleiben wir bei unserem Beispiel und zerlegen die erstellte Spalte `fullName` zu `firstName` und `lastName`.

1. `ADD Column`: erstellen der Spalte `firstName` funktional als:
   $func := SUBSTRING(fullName, 1, CHARINDEX(' ', fullName))$
2. `ADD Column`: erstellen der Spalte `lastName` funktional als:
   $func := SUBSTRING(fullName, CHARINDEX(' ', fullName), LENGTH(fullName))$
3. `DROP Column`: löschen der Spalte `fullName`

Mit diesen beiden Operatoren lassen sich mehrere Evolutionsschritte auf einzelne Schritte reduzieren.

3 Aktueller Stand der Technik

In diesem Kapitel wollen wir den aktuellen Stand von ProSA und das darin enthaltene Tool ChaTEAU betrachten. Dabei schauen wir uns den Aufbau und den generellen Ablauf von ProSA (Siehe 3.1.2) und ChaTEAU (Siehe 3.2) genauer an. Anschließend werfen wir einen Blick auf ein bereits existierenden Prototypen (Siehe 3.3) eines Datenbankmanagement Systems zur Anwendung von Schemaevolution.

3.1 ProSA

ProSA ist ein Forschungsdatenmanagement-Tool zur Analyse, Auswertung und Archivierung von Datenmengen. Ziel ist es, eine minimale Teildatenbank der Originaldatenbank zu erzeugen, um die Reproduzierbarkeit und Nachvollziehbarkeit von Anfrageergebnissen zu ermöglichen und die zu speichernde Menge an Daten zu verringern [Aug20]. Die aktuellste Version von ProSA ist unter
`https://git.informatik.uni-rostock.de/ta093/prosa` zu finden.

3.1.1 Aufbau von ProSA

Wir wollen uns die GUI von ProSA zu beginn dieser Arbeit ansehen, welche im Projekt [WWO+21] entstanden ist, im laufe der Bachelorarbeit [Han22] angepasst wurde und in dieser Arbeit für Evolution erweitert werden soll. Diese wurde mit dem plattformübergreifenden Framework JavaFX spezifiziert und in einer FXML-Konfigurationsdatei festgehalten. Dadurch kann die GUI per Hand oder aber auch über Scene Builder erweitert werden. Dabei wird jedem GUI-Element eine eindeutige ID zugeordnet, sodass diese in Java als Objekte importiert werden können. Ebenso existieren für Event aktivierende GUI-Elemente jeweilige Referenzen, um deren gefeuerten Events zu behandeln. Die einzelnen Tabs sind dabei nach dem linke Seite Eingabe und rechte Seite Ausgabe Schema aufgebaut. Die Erweiterung der GUI wird dann in Kapitel 5 spezifiziert und in Kapitel 6 umgesetzt.

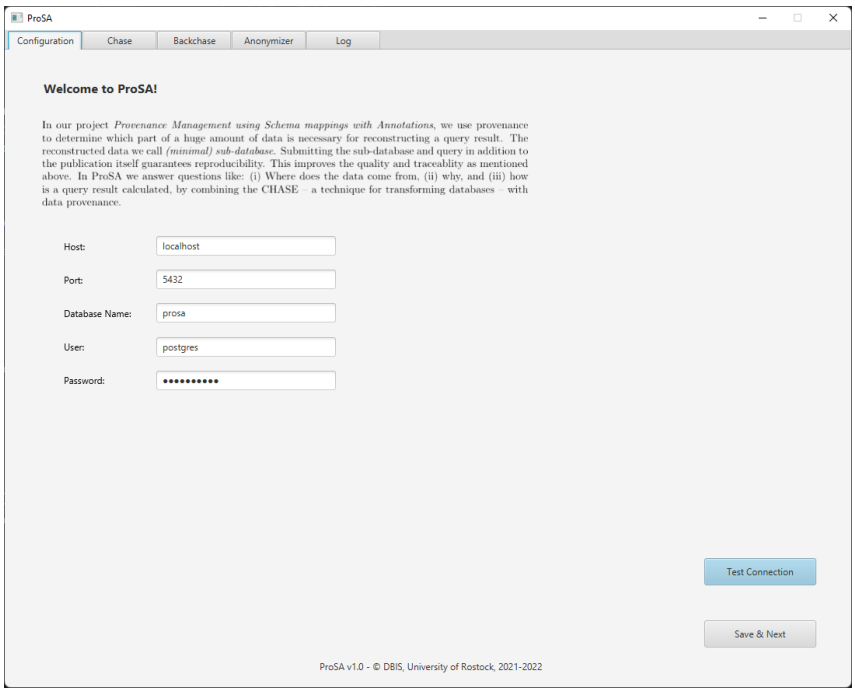

Abbildung 3.1: Configuration-Tab in der ProSA-GUI

In Abbildung 3.1 sehen wir das Configuration-Tab, welches zur Konfigurierung der verwendeten PostgreSQL-Datenbank dient. Die einzelnen Textfelder werden mit den Werten aus einer config.json-Datei automatisch gefüllt und können nach Bedarf angepasst werden. Anschließend kann mit dem Button **Test Connection** die Verbindung getestet werden und man gelangt durch **Save & Next** zum nächsten Tab, dem Chase-Tab (Siehe 3.2).

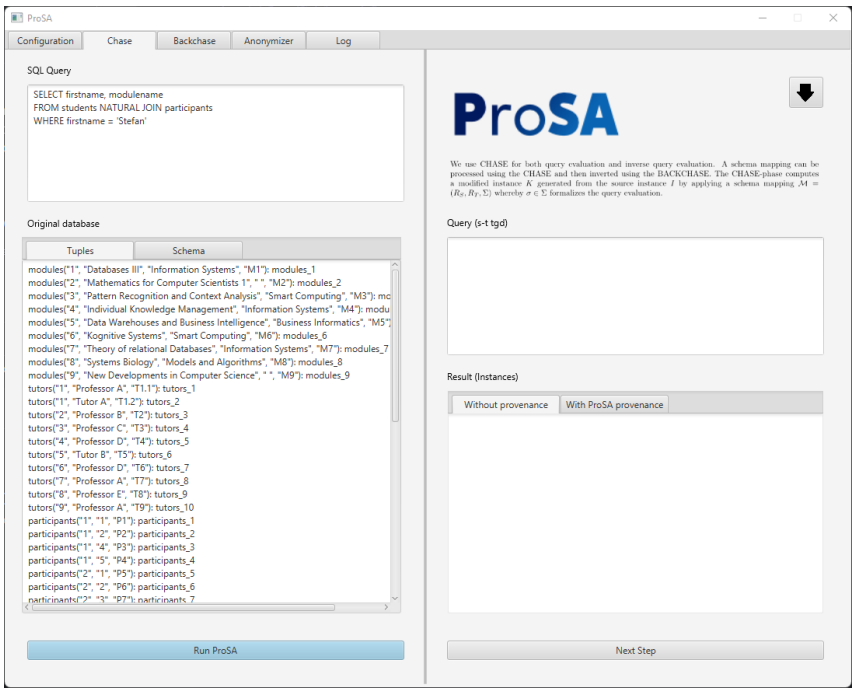

Abbildung 3.2: Chase-Tab

In Abbildung 3.2 sehen wir das Chase-Tab. Im unteren linken Teil wird die Original Datenbank bestehend aus einer Menge von Schemata und Tupeln ausgegeben, welche aus der angegebenen PostgreSQL-Datenbank stammen. Darüber befindet sich ein Textbereich, in welches wir die auszuführende SQL-Anfrage eingeben können. Durch betätigen des **Run ProSA** Buttons, wird die SQL-Anfrage als s-t tgd umgeformt und auf der rechten Seite ausgegeben. Ebenso wird die s-t tgd durch einen Aufruf von ChaTEAU auf die Datenbankinstanz angewendet und das daraus resultierende Ergebnis mit und ohne zusätzlicher Provenance ausgegeben. Im Hintergrund wird ebenfalls die s-t tgd invertiert und durch eine Backchase-Phase [Ros20] erneut angewendet. Das Ergebnis des Backchase wird im separaten Backchase-Tab ausgegeben. Weiterhin kann die Eingabe, die Ausführung und das Ergebnis über den oberen rechten Button als Text-Datei gespeichert werden.

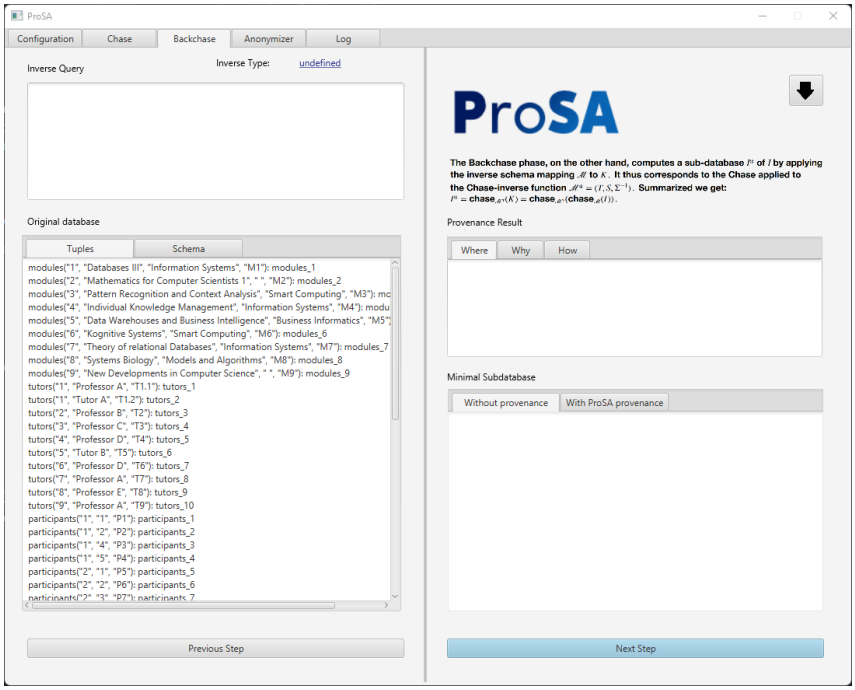

Abbildung 3.3: Backchase-Tab

In Abbildung 3.3 sehen wir das Backchase-Tab, welches nach **Run ProSA** gefüllt wird. Im **Inverse Query** Textbereich wird die Inverse Anfrage als s-t tgd gezeigt, welche mit dem implementierten Invertierer [Spo22] automatisch aus der ursprünglichen SQL-Anfrage generiert wird. Auf der rechten Seite wird das Provenance Ergebnis unterteilt in where, why and how ausgegeben (beschrieben in [Han22] Kapitel 2.2). Darunter erhalten wir die durch den Backchase berechnete minimale Teildatenbank jeweils mit und ohne Provenance.
Damit haben wir die relevanten GUI-Elemente kennengelernt, welche im Laufe dieser Arbeit angepasst und erweitert werden müssen.

3.1.2 Ablauf von ProSA

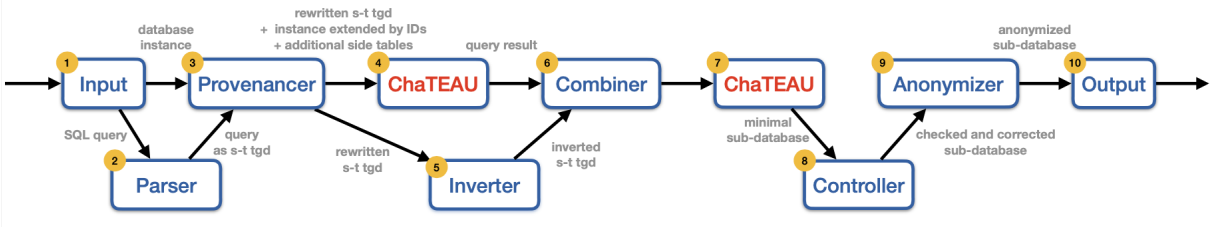

Abbildung 3.4: Ablauf von ProSA aus [Aug22]

In Abbildung 3.4 wird der generelle Ablauf von ProSA grafisch dargestellt. Der Input (1) besteht aus der PostgreSQL-Datenbank und der eingebenen SQL-Anfrage. Die SQL-Anfrage wird dann durch einen Parser (2) zu einer s-t tgd umgeformt und an den Provenancer (3) übergeben. Die s-t tgd und die Datenbankinstanz werden dann als XML-File an ChaTEAU (4) für den Chase-Schritt übergeben und das Ergebnis gelangt zusammen mit der invertierten (5) Anfrage zum Kombinierer (6), welcher ein XML-Datei für den Backchase-Schritt (7) generiert. Dadurch erhalten wir die minimale Teildatenbank, welche anschließend anonymisiert (9) werden kann. Somit erhalten wir als Output (10) eine (anonymisierte) minimale Teildatenbank. [AHH22]

3.2 ChaTEAU

ChaTEAU ist ein an der Universität Rostock entwickeltes Tool, welches den Chase-Algorithmus implementiert. Bislang wird ChaTEAU für den Chase und den Backchase verwendet. Die aktuellste Version von ChaTEAU ist unter `https://git.informatik.uni-rostock.de/ta093/chateau` zu finden. Dazu wird eine XML-Datei mit Datenbankschema, Abhängigkeiten und Datenbankinstanz übergeben, welche wir im Kapitel Grundlagen kennengelernt haben. Ein Beispiel für eine s-t tgd als `COPY Table` Evolutionsoperator einer `Students` Relation, ist in Abbildung 3.5 gegeben. Dieses enthält die `Students` Relation jeweils als **source** und **target** Schema, sowie die `StudensCopy` Relation als **target** Schema. Dabei werden die **source** Schemata in den **Body** und die **target** Schemata in den **Head** der s-t tgd geschrieben. Die Instanz enthält dabei lediglich zwei Tupel. Durch die Ausführung des Chase-Algorithmus erhalten wir dann folgende Instanz:

Students(1, "Muster", "Max", "Electrical engineering")
Students(2, "Johansen", "Johannes", "Computer Science")
StudentsCopy(1, "Muster", "Max", "Electrical engineering")
StudentsCopy(2, "Johansen", "Johannes", "Computer Science")

```
<input>
  <schema>
    <relations>
      <relation name="Students" tag="S">
        <attribute name="student_id" type="int" />
        <attribute name="lastname" type="string" />
        <attribute name="firstname" type="string" />
        <attribute name="course" type="string" />
      </relation>
      <relation name="Students" tag="T">
        <attribute name="student_id" type="int" />
        <attribute name="lastname" type="string" />
        <attribute name="firstname" type="string" />
        <attribute name="course" type="string" />
      </relation>
      <relation name="StudentsCopy" tag="T">
        <attribute name="student_id" type="int" />
        <attribute name="lastname" type="string" />
        <attribute name="firstname" type="string" />
        <attribute name="course" type="string" />
      </relation>
    </relations>
    <dependencies>
      <sttgd>
        <body>
          <atom name="Students">
            <variable name="student_id" type="V" index="1" />
            <variable name="lastname" type="V" index="1" />
            <variable name="firstname" type="V" index="1" />
            <variable name="course" type="V" index="1" />
          </atom>
        </body>
        <head>
          <atom name="Students">
            <variable name="student_id" type="V" index="1" />
            <variable name="lastname" type="V" index="1" />
            <variable name="firstname" type="V" index="1" />
            <variable name="course" type="V" index="1" />
          </atom>
          <atom name="StudentsCopy">
            <variable name="student_id" type="V" index="1" />
            <variable name="lastname" type="V" index="1" />
            <variable name="firstname" type="V" index="1" />
            <variable name="course" type="V" index="1" />
          </atom>
        </head>
      </sttgd>
    </dependencies>
  </schema>
  <instance>
    <atom name="Students">
      <constant name="student_id" value="1" />
      <constant name="lastname" value="Muster" />
      <constant name="firstname" value="Max" />
      <constant name="course" value="Electrical engineering" />
    </atom>
    <atom name="Students">
      <constant name="student_id" value="2" />
      <constant name="lastname" value="Johansen" />
      <constant name="firstname" value="Johannes" />
      <constant name="course" value="Computer Science" />
    </atom>
  </instance>
</input>
```

Abbildung 3.5: COPY Table für Students Relation

3.3 PRISM

PRISM [CMHZ09] ist ein von Carlo A. Curino, Hyun J. Moon und Carlo Zaniolo entwickeltes Datenbankmanagementsystem, welches Schemaevolution durch Disjunctive Embedded Dependencys (DEDs) ermöglicht. Während unser Ziel das temporäre Zurücksetzen von Evolutionen ist, um damalige Anfragen erneut an die Datenbank zu stellen, hat PRISM das Ziel Evolution für den Datenbank Administrator intuitiver zu gestalten. Hauptsächlich werden dabei fünf Anforderungen genannt, welche an PRISM gestellt werden:

1. Sprache für SMOs, zum Ausdruck komplexer Schema-Änderungen.
2. Werkzeuge zur Auswertung von Effekten der Schema-Änderungen.
3. Optimierung von alten Anfragen, welche auch auf neuen Schemaversionen ausgeführt werden sollen.
4. Automatische Daten Migration.
5. Dokumentation der Änderungen zur Unterstützung von:
    - Data Provenance
    - Datenbankwiederherstellung
    - Historisierung von Anfragen

Der PRISM-Prototyp wurde 2009 in einer Live-Demo [CMHZ09] vorgestellt und war zu der Zeit das am weitesten entwickelte System zur Unterstüzung von relationaler Schema Evolution. Das Werkzeug basiert dabei auf AJAX als Frontend, Java als Backend und der Verbindung zu einer MySQL Datenbank.
Die Durchführung der Evolution ist dabei in fünf Schritte unterteilt.

1. Konfiguration: Herstellen einer Verbindung zur MySQL Datenbank.
2. SMO Design: Spezifizierung der SMO's mit Assistenten für Syntax, Sicherung von Informationen und Tests auf Redundanzen.
3. Inverse Design: Bilden der inversen SMOs, welche für die Optimierung alter Anfragen benötigt werden.
4. Validierung: Gibt eine Übersicht der Schema-Mappings und bietet das Testen der Anfrageoptimierung an.
5. Deployment: Das System generiert ein SQL-Datenmigrationsskript, um das Schema zu evolutionieren und die Daten zu migrieren.

4 Aktueller Stand der Forschung

In Abschnitt 2.3 haben wir bereits die für uns relevanten 15 SMOs kennengelernt. Nun wollen wir uns anschauen, wie diese Operatoren als s-t tgds formalisiert werden, damit diese durch den Chase realisert werden können. Dazu werden die Erkenntnisse aus [Aug22] und [CMZ08] im Abschnitt 4.1 zusammengetragen und anschließend die Invertierung dieser Operatoren nach [Spo22] in Abschnitt 4.2 gezeigt.

4.1 SMO als s-t tgds

SMO	**s-t tgd** : Σ
COPY Table	$R(a,b,c) \rightarrow R'(a,b,c) \wedge V(a,b,c)$
	$R(a,b,c) \rightarrow R'(a,b,c)$, $R(a,b,c) \rightarrow V(a,b,c)$
CREATE Table	$\emptyset \rightarrow \exists A,B,C : R(A,B,C)$
DECOMPOSE Table	$R(a,b,c) \rightarrow T(a,b) \wedge V(b,c)$
	$R(a,b,c) \rightarrow T(a,b)$, $R(a,b,c) \rightarrow V(b,c)$
DROP Table	$R(a,b,c) \rightarrow \emptyset$
JOIN Table	$R(a,b) \wedge V(a,c) \rightarrow T(a,b,c)$
MERGE Table	$R(a,b,c) \rightarrow T(a,b,c)$, $V(a,b,c) \rightarrow T(a,b,c)$
PARTITION Table	$R(a,b,c) \wedge \texttt{cond}_A \rightarrow V(a,b,c)$, $R(a,b,c) \wedge \neg\texttt{cond}_A \rightarrow T(a,b,c)$
RENAME Table	$R(a,b,c) \rightarrow V(a,b,c)$
ADD Column	$R(a,b,c) \rightarrow V(a,b,c,\texttt{const}\|f(a,b,c))$
	$R(a,b,c) \rightarrow \exists D : V(a,b,c,D)$
COPY Column	$R(a,b,c) \wedge V(d,e) \wedge \texttt{cond}_A \rightarrow R'(a,b,c) \wedge T(c,d,e)$, $R(a,b,c) \wedge V(d,e) \wedge \neg\texttt{cond}_A \rightarrow \exists C : R'(a,b,c) \wedge T(C,d,e)$
	$R(a,b,c) \rightarrow T(a,b,c,c)$
DROP Column	$R(a,b,c,d) \rightarrow T(a,b,c)$
MERGE Column	$R(a,b,c) \rightarrow T(b,f(a,c))$
MOVE Column	$R(a,b,c) \wedge V(d,e) \wedge \texttt{cond}_A \rightarrow R'(a,b) \wedge T(c,d,e)$
	$R(a,b,c) \wedge V(d,e) \wedge \texttt{cond}_A \rightarrow R'(a,b) \wedge T(c,d,e)$, $R(a,b,c) \wedge V(d,e) \wedge \neg\texttt{cond}_A \rightarrow \exists C R'(a,b) \wedge T(C,d,e)$
RENAME Column	$R(a,b,c) \rightarrow V(a,b,c)$
SPLIT Column	$R(a,b) \rightarrow T(f_1(a),b,f_2(a))$

4.1.1 SMO als s-t tgd entnommen aus [Aug22]

In Tabelle 4.1 sind unsere SMOs und deren Umformungen als Menge von s-t tgds zu finden. Diese stammen aus [Aug22] und werden für die spätere Notation in der XML-Datei benötigt. Hierbei werden klein geschriebene Variablen als allquantifiziert und groß geschriebene Variablen als existenzquantifiziert aufgefasst. R, R', V und T sind Relationen über den jeweiligen Attributen. Die Bedindungen `cond`, ermöglichen einen Vergleich zwischen Konstanten oder zwischen Konstanten und Variablen. Die Funktionen f hingegen, können im aktuellen Stand von ChaTEAU noch nicht verarbeitet werden.

4.2 Invertierung von s-t tgds

Um das spätere Konzept umzusetzen, wird die Invertierung von s-t tgds benötigt, welche im Invertierer von ProSA implementiert sind. Dieser wurde in der Masterarbeit [Spo22] entwickelt und soll nun für die Invertierung der Evolution verwendet werden. Der folgende Algorithmus ist dabei eine vereinfachte Version des implementierten Invertierers und bezieht sich lediglich auf eine einzelne s-t tgd. Bei der Invertierung von SMOs, welche aus zwei s-t tgds bestehen, muss stattdessen der **Algorithmus 5: Invertierung** aus [Spo22] verwendet werden.

Algorithm 2 Invertierung, aus [Spo22]

Input: Schemaabbildung $\mathcal{M} = (S, T, \Sigma)$, wobei Σ eine s-t tgd entspricht.
Output: Invertierte Schemaabbildung $\mathcal{M}' = (T, S, \Omega)$ von $\mathcal{M}$, wobei Ω einer invertierten s-t tgd entspricht.

1. (Invertierung der s-t tgd) Invertiere die s-t tgd, indem alle s-t tgds der Form $\alpha \rightarrow \beta$ durch $\beta \rightarrow \alpha$ ersetzt werden.
2. (Anwendung der Gleichheitsprädikate) Ersetze bei einer s-t tgd der Form $\beta \rightarrow \alpha \wedge x = a$ alle Variablen x durch den Wert a bzw. durch die Variable a und entferne das Gleichheitsprädikat $x = a$.
3. (Anpassung der Quantoren) Alle Variablen einer s-t tgd, die im Kopf aber nicht im Rumpf vorkommen, werden existenzquantifiziert. Ebenfalls werden alle Variablen die durch einen konkreten Wert ersetzt wurden nicht quantifiziert. Die restlichen Variablen werden allquantifiziert.

Return: $\mathcal{M}' = (T, S, \Omega)$.

Als Bespiel bestrachten wir die s-t tgd, welche wir in Definition 2.7 bereits gesehen haben.

$$\forall mano, ln, fn, c, imy : S(mano, ln, fn, c, imy) \rightarrow S_A(mano, c, imy).$$

Schritt 1: Wir invertieren die s-t tgd, indem wir den Rumpf mit dem Kopf vertauschen und vorerst die Quantoren entfernen:

$$S_A(mano, c, imy) \rightarrow S(mano, ln, fn, c, imy).$$

Den 2. Schritt brauchen wir in den Fällen unserer SMOs nicht beachten, da dort keine Gleichheitsprädikate vorkommen.
Schritt 3: Alle im Kopf vorkommenden Variablen, welche nicht im Rumpf vorkommen, werden existenzquantifiziert:

$$S_A(mano, c, imy) \rightarrow \exists ln, fn : S(mano, ln, fn, c, imy).$$

Die restlichen Variablen werden allquantifiziert:

$$\forall mano, c, imy : S_A(mano, c, imy) \rightarrow \exists ln, fn : S(mano, ln, fn, c, imy).$$

SMO	s-t tgd : Σ	s-t tgd : Σ^{-1}	SMO
COPY Table	$R(a,b,c) \rightarrow R'(a,b,c) \wedge V(a,b,c)$	$R'(a,b,c) \wedge V(a,b,c) \rightarrow R(a,b,c)$	MERGE Table
	$R(a,b,c) \rightarrow R'(a,b,c)$, $R(a,b,c) \rightarrow V(a,b,c)$	$R'(a,b,c) \rightarrow R(a,b,c)$, $V(a,b,c) \rightarrow R(a,b,c)$	
CREATE Table	$\emptyset \rightarrow \exists A,B,C : R(A,B,C)$	$R(a,b,c) \rightarrow \emptyset$	DROP Table
DECOMPOSE Table	$R(a,b,c) \rightarrow T(a,b) \wedge V(b,c)$	$T(a,b) \wedge V(b,c) \rightarrow R(a,b,c)$	JOIN Table
	$R(a,b,c) \rightarrow T(a,b)$, $R(a,b,c) \rightarrow V(b,c)$	$T(a,b) \rightarrow R(a,b,c)$, $V(b,c) \rightarrow R(a,b,c)$	
DROP Table	$R(a,b,c) \rightarrow \emptyset$	$\emptyset \rightarrow \exists A,B,C : R(A,B,C)$	CREATE Table
JOIN Table	$R(a,b) \wedge V(a,c) \rightarrow T(a,b,c)$	$T(a,b,c) \rightarrow R(a,b) \wedge V(a,c)$	DECOMPOSE Table
MERGE Table	$R(a,b,c) \rightarrow T(a,b,c)$, $V(a,b,c) \rightarrow T(a,b,c)$	$T(a,b,c) \rightarrow R(a,b,c)$, $T(a,b,c) \rightarrow V(a,b,c)$	COPY Table
PARTITION Table	$R(a,b,c) \wedge \texttt{cond}_A \rightarrow V(a,b,c)$, $R(a,b,c) \wedge \neg\texttt{cond}_A \rightarrow T(a,b,c)$	$V(a,b,c) \rightarrow R(a,b,c)$, $T(a,b,c) \rightarrow R(a,b,c)$	MERGE Table
RENAME Table	$R(a,b,c) \rightarrow V(a,b,c)$	$V(a,b,c) \rightarrow R(a,b,c)$	RENAME Table
ADD Column	$R(a,b,c) \rightarrow V(a,b,c,\texttt{const}\|f(a,b,c))$	$V(a,b,c,\texttt{const}\|f(a,b,c)) \rightarrow R(a,b,c)$	DROP Column
	$R(a,b,c) \rightarrow \exists D : V(a,b,c,D)$	$V(a,b,c,d) \rightarrow R(a,b,c)$	
COPY Column	$R(a,b,c) \wedge V(d,e) \wedge \texttt{cond}_A \rightarrow R'(a,b,c) \wedge T(c,d,e)$, $R(a,b,c) \wedge V(d,e) \wedge \neg\texttt{cond}_A \rightarrow \exists C : R'(a,b,c) \wedge T(C,d,e)$	$R'(a,b,c) \rightarrow R(a,b,c)$ $T(c,d,e) \rightarrow V(d,e)$	DROP Column
	$R(a,b,c) \rightarrow T(a,b,c,c)$	$T(a,b,c,c) \rightarrow R(a,b,c)$	
DROP Column	$R(a,b,c,d) \rightarrow T(a,b,c)$	$T(a,b,c) \rightarrow \exists D : R(a,b,c,D)$	ADD Column
MERGE Column	$R(a,b,c) \rightarrow T(b,f(a,c))$	$T(b,f(a,c)) \rightarrow \exists D,E : R(D,b,E)$	SPLIT Column
MOVE Column	$R(a,b,c) \wedge V(d,e) \wedge \texttt{cond}_A \rightarrow R'(a,b) \wedge T(c,d,e)$	$R'(a,b) \wedge T(c,d,e) \wedge \texttt{cond}_A \rightarrow R(a,b,c) \wedge V(d,e)$	MOVE Column
	$R(a,b,c) \wedge V(d,e) \wedge \texttt{cond}_A \rightarrow R'(a,b) \wedge T(c,d,e)$, $R(a,b,c) \wedge V(d,e) \wedge \neg\texttt{cond}_A \rightarrow \exists C R'(a,b) \wedge T(C,d,e)$	$R'(a,b) \wedge T(c,d,e) \rightarrow R(a,b,c)$, $T(c,d,e) \rightarrow V(d,e)$	
RENAME Column	$R(a,b,c) \rightarrow V(a,b,c)$	$V(a,b,c) \rightarrow R(a,b,c)$	RENAME Column
SPLIT Column	$R(a,b) \rightarrow T(f_1(a),b,f_2(a))$	$T(f_1(a),b,f_2(a)) \rightarrow \exists C : R(C,b)$	MERGE Column

4.2.1 SMO als s-t tgd und deren Inversen entnommen aus [Aug22]

5 Konzept

In diesem Kapitel wird das Konzept zur Erweiterung von ProSA vorgestellt. Dazu schauen wir uns im einzelnen die Eingabe, die Verarbeitung und die Ausgabe für Schemaevolution genauer an. In Abschnitt 5.1 wird die Eingabe über eine XML-Datei vorgestellt. Dazu wird für jeden Schemaevolutionsoperator ein Beispiel gegeben. In Abschnitt 5.2 wird geklärt, wann und wie ChaTEAU innerhalb von ProSA aufgerufen werden soll, um die Schemaevolution anzuwenden. Anschließend wird in Abschnitt 5.3 ein Beispiel gegeben, welches für das Verständnis des Konzeptes beitragen soll.

5.1 Eingabe über XML-Datei

Die Eingabe von Schemaevolutionen werden im XML-Format realisiert. Dazu soll in ProSA eine Möglichkeit zum Einlesen einer Datei im XML-Format implementiert werden. Das Format entspricht dem aus Abbildung 3.5, wobei wir lediglich das Schema mit Relationen und Abhängigkeiten benötigen, da die Instanz aus der Datenbank gelesen wird. Dabei müssen alle Relationen, welche für die Ausführung der Abhängigkeit benötigt werden, mit dem Tag "S" für Source versehen werden. Weiterhin müssen alle Relationen, welche wir nach Anwendung des Chase erhalten wollen, mit den Tag "T" für Target markiert werden. Im Folgenden wird jeder Operator kurz erklärt und jeweils eine Beispieleingabe vorgezeigt. Ebenfalls existieren bei manchen Operatoren bestimmte Bedingungen, welche dann genannt werden. Für jeden Operator wird ebenfalls dies zugehörige s-t tgd angegeben, wobei Relations- und Attributnamen zu Initialen abgekürzt werden.

COPY Table nutzen wir, um bereits existierende Schemata zu duplizieren. Ein Beispiel zum Kopieren der Relation `students` ist in Anhang 8.1 gegeben. Dazu wird die `students` Relation mit dem Tag "S" versehen und die zwei neuen Relationen `studentsOrig` und `studentsCopy` mit dem Tag "T". Anschließend muss lediglich die Source-Relation in den Rumpf und die Target-Relationen in den Kopf der s-t tgd geschrieben werden. Die auszuführende s-t tgd sieht dabei wie folgt aus:

$$\forall ma, ln, fn, sc, id : S(ma, ln, fn, sc, id_s) \rightarrow SO(ma, ln, fn, sc, id_s) \wedge SC(ma, ln, fn, sc, id_s)$$

CREATE Table nutzen wir, um ein neues Schema einzufügen. Ein Beispiel zum Erstellen einer Relation `grades` ist in Anhang 8.2 gegeben. Dabei darf der Rumpf der Abhängigkeit nicht leer sein, da sonst keine Trigger gefunden werden können. Deswegen wird eine Source-Relation eingeführt, welche mit dem Namen `trigger` versehen wurde. Dadurch wird für jedes Tupel in der Instanz ein Tupel für die Relation `grades` erstellt und entsprechend mit Nullwerten aufgefüllt. Konkret wird folgende s-t tgd verwendet:

$$\forall tr : T(tr) \rightarrow \exists mo, ma, se, gr, id_g : G(mo, ma, se, gr, id_g)$$

DECOMPOSE Table nutzen wir, um ein Schema in zwei Schemata zu teilen. Als Beispiel betrachten wir die Trennung der Relation `students`, um das Attribut `studycourse` separat vom Namen zu speichern (Anhang 8.3). Hier teilen wir das Schema über das Attribut `matricno`. Dabei ist eine Trennung auch ohne ein gemeinsames Attribut möglich, jedoch ist anschließend das Umkehren dieser Operation nicht mehr eindeutig. Die entsprechende s-t tgd ist definiert als:

$$\forall ma, ln, fn, sc, id_s : S(ma, ln, fn, sc, id_s) \rightarrow SN(ma, ln, fn) \wedge SC(ma, sc)$$

DROP Table nutzen wir um ein Schema zu löschen. Da alle Schemata gelöscht werden, welche nicht das Tag "T" besitzen, kann ein Schema durch weglassen in der XML-Datei entfernt werden. Möchten wir zum Beispiel das Schema `tutors` löschen, dann übernehmen wir alle anderen Schemata und fügen das Tag "T" hinzu. Die s-t tgd hat in diesem Fall einen leeren Rumpf und Kopf. Intuitiv würde man das Schema als Source hinzufügen und die s-t tgd als $R(a, b, c) \rightarrow \emptyset$ definieren, jedoch wird der Chase dabei keine aktiven Trigger finden, da keine neuen Tupel entstehen.

JOIN Table nutzen wir, um aus zwei Schemata mit einem gemeinsamen Attribut, ein Verbund-Schema zu bilden. Als Beispiel bilden wir das Schema `participantsWithTitle` als Verbund über `modules` und `participants` (Anhang 8.4).

$$\forall mo, ti, mj, id_m, ma, id_p : M(mo, ti, mj, id_m) \wedge P(mo, ma, id_p) \rightarrow PWT(mo, ti, ma)$$

MERGE Table nutzen wir, um zwei Datenbankinstanzen über dem selben Schema zu vereinigen. Als Beispiel vereinigen wir die Schemata `students1` und `students2` zu `students` (Anhang 8.5). Wir wenden folgende Menge von s-t tgds an:

$$\begin{gathered}\forall s_{id}, ln, fn, co : S1(s_{id}, ln, fn, co) \rightarrow S(s_{id}, ln, fn, co), \\ S2(s_{id}, ln, fn, co) \rightarrow S(s_{id}, ln, fn, co)\end{gathered}$$

PARTITION Table (mit ChaTEAU 1.1 nicht unterstützt) nutzen wir, um ein Schema durch Bedingungen in zwei Schemata zu teilen. Als Beispiel wird die Relation `grades` zerlegt in `gradesBestanden` und `gradesNichtBestanden`. Dazu werden alle Tupel mit `grade` kleiner als 5 in das Schema `gradesBestanden` überführt und alle Tupel, welche diese Bedingung nicht erfüllen, in das Schema `gradesNichtBestanden` überführt. Die Menge an s-t tgds wird dann wie folgt definiert:

$$\begin{gathered}\forall mo, ma, se, gr, id_g : G(mo, ma, se, gr, id_g) \wedge gr < 5 \rightarrow GB(mo, ma, se, gr, id_g), \\ G(mo, ma, se, gr, id_g) \wedge \neg gr < 5 \rightarrow GNB(mo, ma, se, gr, id_g)\end{gathered}$$

RENAME Table nutzen wir, um den Namen eines Schemas zu ändern. Als Beispiel benennen wir das Schema `students` zu `studenten` (Anhang 8.6). Dafür fügen wir lediglich die beiden Schemata mit den jeweiligen Tags hinzu, schreiben das Schema `students` in den Body und schreiben das Schema `studenten` in den Head der s-t tgd.

$$\forall ma, ln, fn, sc, id_s : S_{eng}(ma, ln, fn, sc, id_s) \rightarrow S_{deu}(ma, ln, fn, sc, id_s)$$

ADD Column nutzen wir, um ein Schema durch ein neues Attribut zu erweitern. Als Beispiel erweitern wir das Schema `students` um das Attribut `semester` (Anhang 8.7). Dafür wird das Schema `students` als Source markiert und in den Body der s-t tgd geschrieben. Das Schema `studentsAdded` markieren wir als Target und schreiben dieses in den Head der s-t tgd.

$$\forall ma, ln, fn, sc, id_s : S(ma, ln, fn, sc, id_s) \rightarrow \exists se : SA(ma, ln, fn, sc, se, id_s)$$

COPY Column nutzen wir, um ein Attribut eines Schemas zu kopieren. Als Beispiel kopieren wir das Attribut `title` aus `modules` in das Schema `tutors` (Anhang 8.8). Dafür fügen wir die Schemata `modules` und `tutors` jeweils als Source hinzu und schreiben diese in den Body der s-t tgd. Das resultierende Schema `tutorsWithTitle` markieren wir als Target und schreiben dieses in den Head der s-t tgd.

$$\forall mo, ti, mj, id_m, tu, id_t : M(mo, ti, mj, id_m) \wedge T(mo, tu, id_t) \rightarrow MN(mo, ti, mj, id_m) \wedge TWT(mo, ti, tu, id_t)$$

DROP Column nutzen wir, um ein Attribut eines Schemas zu entfernen. Als Beispiel entfernen wir das Attribut `studycourse` aus `students` (Anhang 8.9). Hierbei muss ebenfalls der Name des Schemas angepasst werden, da ChaTEAU(Version 1.1) während des Chase momentan nicht zwischen Source und Target unterscheidet.

$$\forall ma, ln, fn, sc, id_s : S(ma, ln, fn, sc, id_s) \rightarrow SD(ma, ln, fn, id_s)$$

MERGE Column (mit ChaTEAU 1.1 nicht unterstützt) nutzen wir, um ein neues Attribut zu einem Schema funktional hinzuzufügen. Als Beispiel betrachten wir das verschmelzen der Attribute `firstname` und `lastname` zu einem neuen Attribut `name`. Die s-t tgd ist dann wie folgt definiert:

$$\forall ma, ln, fn, sc, id_s : S(ma, ln, fn, sc, id_s) \rightarrow \exists f : S_{new}(ma, f(fn, ln), sc, id_s)$$

MOVE Column nutzen wir, um ein Attribut eines Schemas in ein anderes Schema zu verschieben. Als Beispiel verschieben wir das Attribut `tutor` von `tutors` zu `modules` (Anhang 8.10). Dafür fügen wir `modules` und `tutors` als Source hinzu und schreiben diese in den Body der s-t tgd. Das resultierende Schema `modulesWithTutor` markieren wir als Target und schreiben dieses in den Head der s-t tgd.

$$\forall mo, ti, mj, id_m, tu, id_t : M(mo, ti, mj, id_m) \wedge T(mo, tu, id_t) \rightarrow MWT(mo, ti, mj, tu, id_m)$$

RENAME Column (mit ChaTEAU 1.1 nicht unterstützt) nutzen wir, um den Namen eines Attributs zu ändern. Um diesen Operator ausführen zu können, werden funktionale Abbildungen benötigt. Dazu würde ein neues Schema angelegt werden, welches das Attribut mit neuem Namen einführt und funktional mit den Werten des alten Attributs befüllt.

SPLIT Column (mit ChaTEAU 1.1 nicht unterstützt) nutzen wir, um aus einem Attribut funktional zwei neue Attribute zu erstellen. Ein Beispiel für eine s-t tgd, welche aus dem durch `MERGE Column` erstellten Attribut `name` die Attribute `firstname` und `lastname` erstellt, wäre wie folgt definiert:

$$\forall ma, na, sc, id_s : S(ma, na, sc, id_s) \rightarrow \exists f_1, f_2 : S_{new}(ma, f_1(na), f_2(na), sc, id_s)$$

5.2 Geplanter Ablauf für ProSA mit Evolution

In der Abbildung 3.4 haben wir uns bereits den Ablauf von ProSA zu Beginn dieser Arbeit angesehen. Im Folgenden wollen wir die Grafik für Evolution erweitern und anhand dessen das Konzept verstehen.

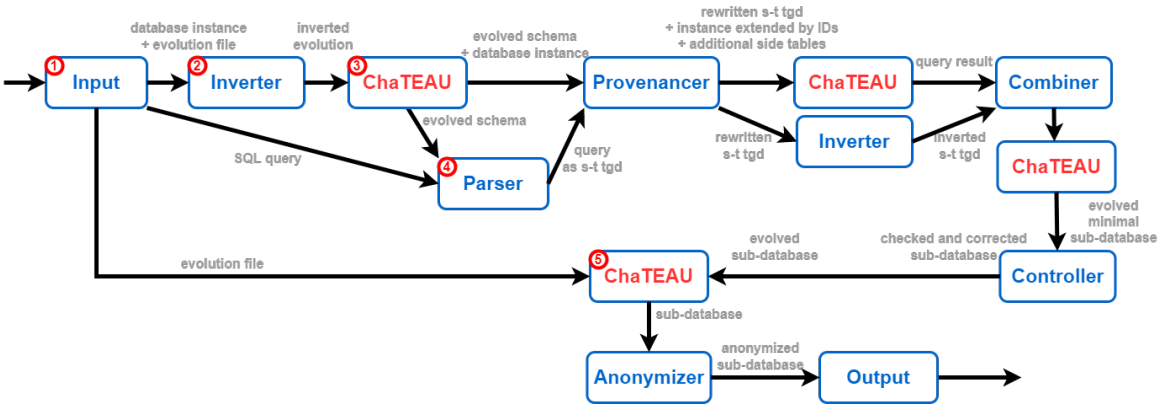

Abbildung 5.1: Ablauf von ProSA mit Evolution

Der Zweck zur Implementierung von Evolution ist das Umkehren von Evolutionsschritten um damalige Anfragen erneut zu stellen. Der Invertierer soll deswegen direkt beim Einlesen der Evolution aufgerufen werden. Dadurch muss die Evolution (S_t, S_{t+1}, Σ) lediglich so eingegeben werden, wie sie definiert und ausgeführt wurde und muss nicht per Hand invertiert und eingegeben werden. Konkret wird am Anfang die inverse Evolution $(S_{t+1}, S_t, \Sigma^{-1})$ ausgeführt, welche die eingelesene Datenbank auf einen vorherigen Stand bringt. Anschließend wird ProSA wie gewohnt ausgeführt und wir erhalten eine (anonymisierte) minimale Teildatenbank. Diese wird mit einem erneuten Aufruf von ChaTEAU mit der ürsprünglich eingegebenen Evolution wieder auf den aktuellen Stand gebracht. Bei den Überlegungen an welchen Stellen die Evolution eingebracht werden soll, wurde sich für den Anfang und das Ende des Ablaufes von ProSA entschieden. Der erste Aufruf der Evolution muss vor der Anfragestellung geschehen und ist somit die einzige Positionierungsmöglichkeit. Der zweite Aufruf der Evolution ist optimalerweise nach dem Backchase und kann vorher nicht angewendet werden, da der Backchase nur mit dem Schema funktioniert, welches nach der Anwendung der SQL query enstand. Würden wir also nach der Anfrage evolutionieren, könnten wir keine minimale Teildatenbank durch den Backchase generieren. Der Anonymisierungsschritt ist optional, jedoch werden wir die zweite Evolution vor dem Anonymisieren anwenden, da die Anonymisierung das Schema (Datentypen) ändern kann, sodass eine Evolution nicht mehr möglich ist.

1. Input: Der Input wird wie in Abschnitt 5.1 beschrieben, um eine Eingabe für eine XML-Datei erweitert. Diese XML-Datei enthält die Evolution, welche invertiert und angewendet werden soll. Dazu wird das Configuration-Tab um eine Dateiauswahl erweitert.

2. Inverter: Der Invertierer soll die eingelesenen Mengen von s-t tgds der Evolutionsdatei invertieren und die Source- und Target-Tags der Schemata anpassen.

3. ChaTEAU (erste Evolution): Der neu hinzugefügte Aufruf von ChaTEAU soll die erste Evolution ausführen. Dazu wird die invertierte Evolution und die Datenbankinstanz verwendet. Wir erhalten daraus die evolutionierte Datenbank.

4. Parser: Der Parser erhält die SQL-Anfrage aus dem Input und das evolvierte Schema von ChaTEAU. Da der Parser die Anfrage als s-t tgd anhand der Datenbankverbindung erstellt, muss der Parser um eine alternative Eingabe erweitert werden. Dazu soll der Parser ein Schema übergeben bekommen, falls eine

Evolution stattgefunden hat und mit diesem die s-t tgd erstellen. Falls keine Evolution stattgefunden hat, soll der Parser wie gewohnt das Schema der Datenbank benutzen.

5. ChaTEAU(zweite Evolution): Hier wird die ursprünglich eingegebe Evolution angewendet, um die minimale Teildatenbank an den aktuellen Stand (S_{t+1}) der eingelesenen Datenbank anzupassen. Dazu wird der Inhalt der Evolutionsdatei in ein Java Objekt gespeichert, sodass die Datei nicht erneut ausgelesen oder ein zweites mal invertiert werden muss.

5.3 Beispiel Eingabe und erwartetes Ergebnis

Der konkret geplante Ablauf in ProSA sieht wie folgt aus:

1. Invertierung der Evolution ($E \rightarrow E^{-1}$)
2. Anwendung des Chase ($E^{-1} : D_{t+1} \rightarrow D_t$)
3. Auswertung der Anfrage Q auf D_t
4. Invertierung der Anfrage Q mit anschließendem Backchase
5. Evolution der minimalen Teildatenbank ($E : D_t \rightarrow D_{t+1}$)
6. (Optional) Anonymisierung der minimalen Teildatenbank

Gegeben seien als Eingabe:

- eine Datenbank D_{t+1} bestehend aus den Tabellen `grades`, `tutors`, `participants`, `students` und `modules`
- die Evolution $E = (S_t, S_{t+1}, \Sigma)$ von D_t nach D_{t+1} mit:
  $\Sigma = \forall mo, ti, mj, id_m, tu, id_t : MN(mo, ti, mj, id_m) \wedge TWT(mo, ti, tu, id_t) \rightarrow$
  $M(mo, ti, mj, id_m) \wedge T(mo, tu, id_t)$
- die Anfrage Q:
  SELECT title, tutor
  FROM tutorsWithTitle

Dabei ist D_{t+1} die aktuell vorhandene Datenbank, welche wir auf das Schema von D_t zurücksetzen wollen, um die Anfrage Q stellen zu können.

Als Ergebnis erwarten wir eine (anonymisierte) minimale Teildatenbank $D_{t+1_{min}}$ von D_{t+1} zur Anfrage Q. Dabei besteht $D_{t+1_{min}}$ lediglich aus den Tabellen `modules` und `tutors`. Dadurch kann die Anfrage Q mit Hilfe der Evolution E erneut an die Datenbank D gestellt werden, ohne die Originaldatenbank zu benötigen.

6 Implementierung

In diesem Kapitel wird die Implementierung zur Erweiterung von ProSA um Schemaevolution gezeigt und erläutert. Dazu wird in Abschnitt 6.1 vorerst die Erweiterung der GUI vorgestellt. In Abschnitt 6.2 werden dann die einzelnen Komponenten von ProSA vorgestellt, welche für Evolution erweitert wurden. Abschließend wird in Abschnitt 6.3 die Evolution an einem Beispiel vorgezeigt.

6.1 Erweiterte GUI

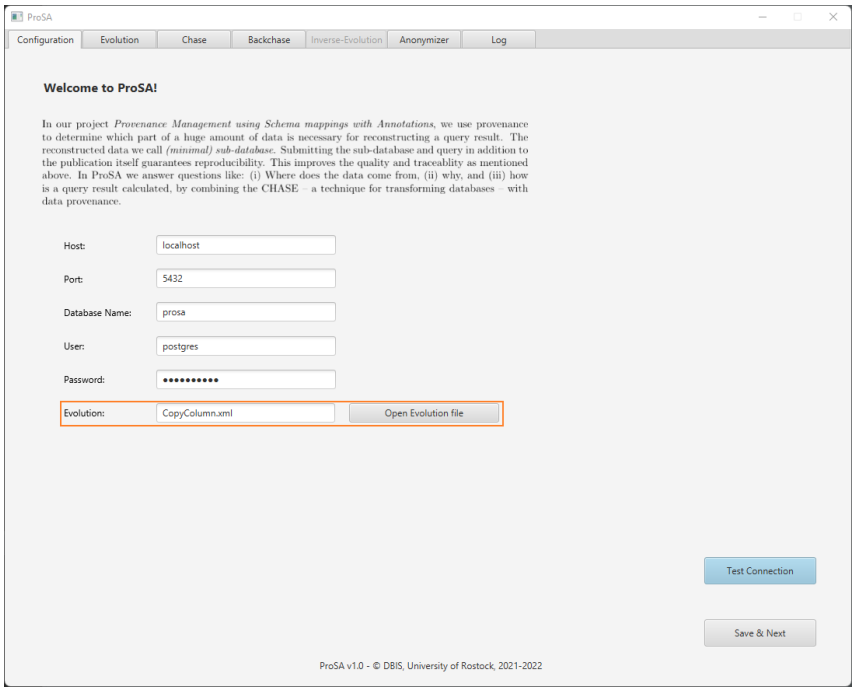

Abbildung 6.1: Configuration-Tab

Das Configuration-Tab (Abbildung 6.1) wurde um die Eingabe einer XML-Datei erweitert. Dazu wird über den **Open Evolution file** Button ein Dateiauswahlfenster geöffnet und der Dateipfad der ausgewählten Datei gespeichert. Mit dem Erhalt des Pfades, wird der Evolution-Tab freigeschaltet. Der Button **Save & Next** führt uns dann zum Evolution-Tab. Falls kein Pfad angegeben wurde, gelangen wir stattdessen zum Chase-Tab und können dann ProSA ohne Evolution nutzen.

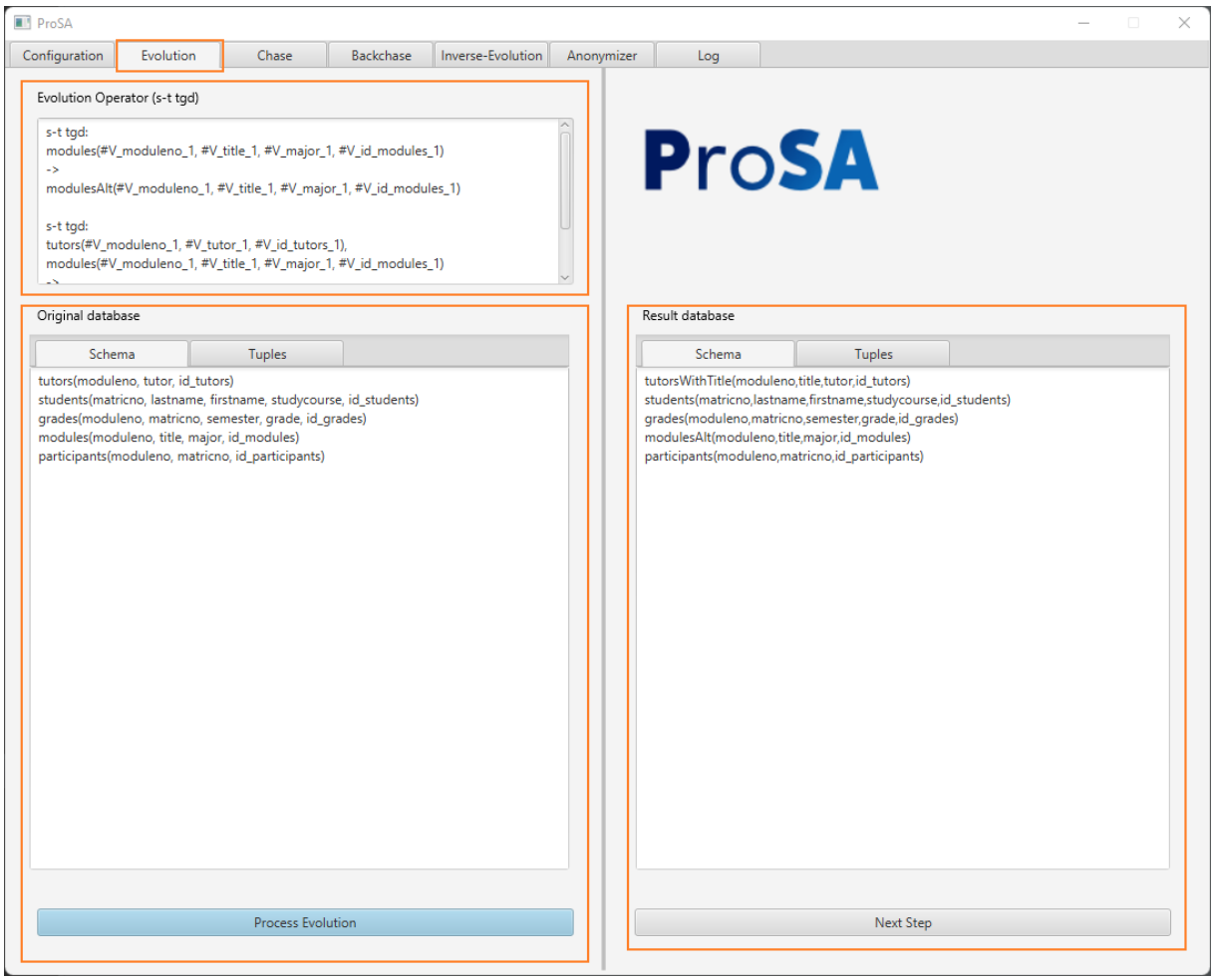

Abbildung 6.2: Evolution-Tab

Im Evolution-Tab (Abbildung 6.2) angelangt, erhält man eine Übersicht der auszuführenden Evolution. Oben Links wird der Evolutionsoperator als Menge von s-t tgds angezeigt, welcher durch den Invertierer erzeugt wurde. Unten Links befindet sich die Datenbank, welche aus der Datenbankverbindung ausgelesen wurde. Dazu werden alle Schemata der Datenbank unter **Schema** angezeigt und alle Tupel der Datenbankinstanz werden unter **Tuples** ausgegeben. Mit dem **Process Evolution** Button, wird diese Evolution ausgeführt und die resultierenden Schemata und Tupel werden auf der rechten Seiten ausgegeben. Ebenfalls wird der Inverse-Evolution-Tab freigeschaltet.

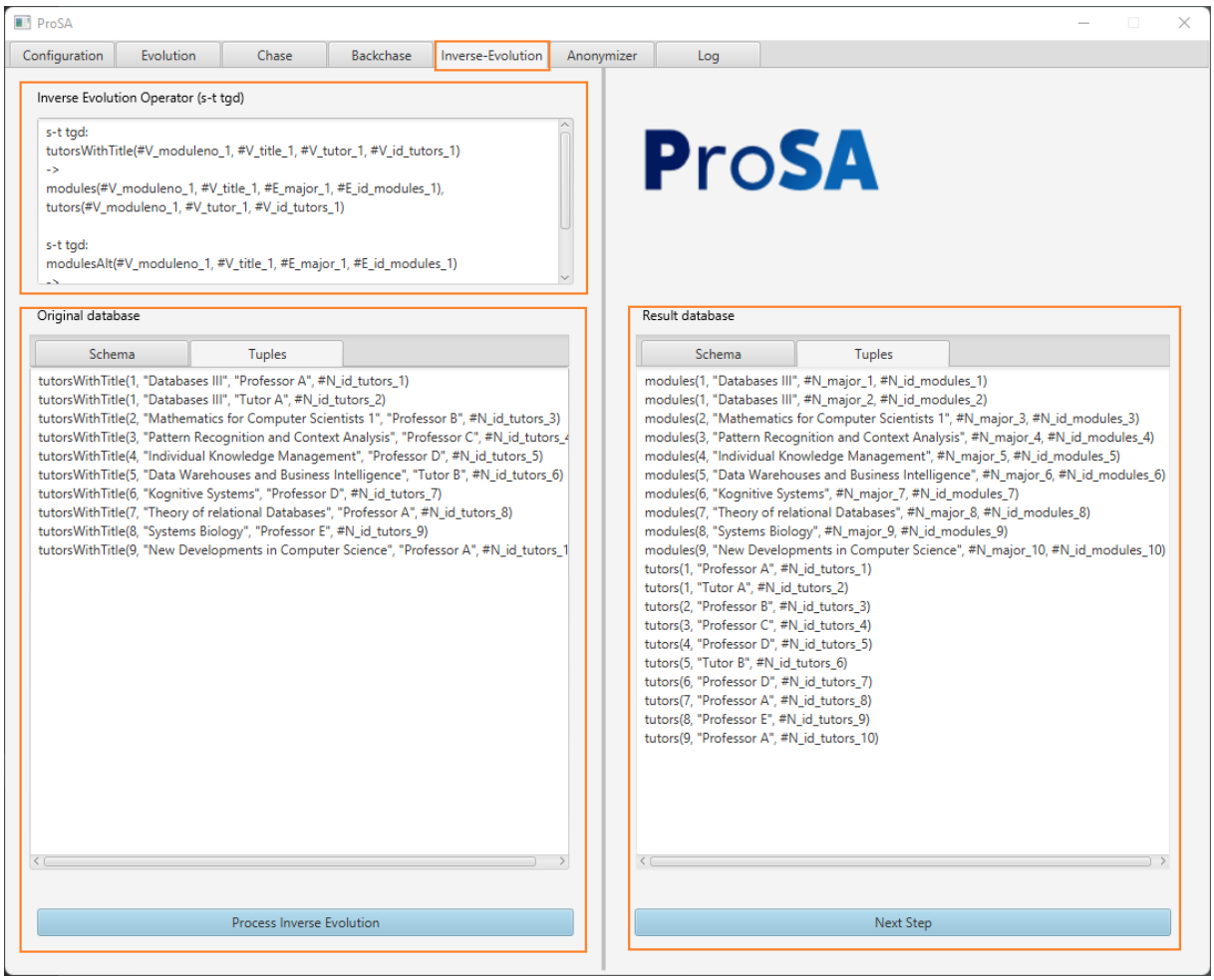

Abbildung 6.3: Inverse-Evolution-Tab

Der Inverse-Evolution-Tab (Abbildung 6.3) ist wie der Evolution-Tab (Abbildung 6.2) aufgebaut. Oben Links wird der ursprünglich in der XML-Datei eingebene Evolutionsoperator als Menge von s-t tgds angezeigt. Unter **Original database** befindet sich die aus dem Backchase erzeugte minimale Teildatenbank, auf welche der Evolutionsoperator angewendet werden soll. Durch **Process Inverse Evolution** wird die Evolution ausgeführt und auf der rechten Seite unter **Result database** angezeigt. Die nicht in der minimalen Teildatenbank enthaltenen Attribute, werden entsprechend mit Nullwerten aufgefüllt. Mit dem Button **Next Step** gelangt man zum optionalen Schritt, dem Anonymisierer.

6.2 Implementierung der Evolution

Für die Implementierung der Evolution wurden im Wesentlichen zwei Komponenten von ProSA angepasst. Dazu schauen wir uns den **MainGuiController** Abschnitt 6.2.1 an, welcher die Aktivierung der hinzugefügten Buttons behandelt und gegebenenfalls die Benutzeroberfläche aktualisiert. In Abschnitt 6.2.2 wird die Erweiterung des Parser vorgestellt, welcher für Evolution angepasst werden musste.

6.2.1 MainGuiController.java

Der **MainGuiController** ist für die Behandlung von Benutzereingaben und dem Anpassen der Benutzeroberfläche zuständig. Konkret wurden die Methoden **openEvolutionFile**, **handleProcessEvolution** und **handleInverseEvolution** hinzugefügt, welche folgend genauer erklärt werden.

openEvolutionFile (Abbildung 6.4)

Die Betätigung des **Open Evolution file** Buttons öffnet einen **File Chooser**. Mit diesem kann das System nach einer Datei durchsucht werden, wobei lediglich Dateien mit der Endung .xml akzeptiert werden. Nachdem eine XML-Datei eingelesen wurde, wird die Methode **openEvolutionFile** aufgerufen, welche als Parameter die eingelesene Datei übergeben bekommt.

Zeile 4-13: Die Datei kann dann von dem **InputReader** zu einem **Instance** Objekt konvertiert werden, welches als Eingabe für ChaTEAU benötigt wird. Dazu speichern wir die Instanz global als **evolutionInput**, welche wir für die Invertierung und den ersten Evolutionsschritt benötigen. Ebenso wird eine Kopie als **origEvolutionInput** gespeichert, welche wir später für die zweite Evolution benötigen werden. In der Theorie könnte die invertierte Evolution auch ein zweites mal invertiert werden, um die ursprüngliche Evolution zurückzuerhalten. Jedoch genügt die Speicherung als globale Variable und wir sparen somit den erneuten Aufruf des Invertierers ein.

Zeile 16-28: Hier wird zunächst die Datenbank ausgelesen und als **SingleInput** Objekt gespeichert, welches wir anschließend um die Schemata und gegebenenfalls um die Tupel der Evolutions XML-Datei erweitern. Dabei werden konkret die **source** Schemata benötigt, welche nach Invertierung der Evolution zu **target** Schemata werden. Diese sind demnach die zu erzeugenden Schemata und stehen nach der Invertierung im Kopf der s-t tgd.

Zeile 30-51: Hier wird der Evolutionsoperator als Menge von s-t tgds invertiert. Für spätere Erweiterungen wurde auch die Invertierung von tgds berücksichtig, welche bei unseren Eingaben jedoch nicht benötigt werden. Zur Invertierung werden konkret die **Constraints** des Inputs abgebildet und an **InvertingTechniques.invert** übergeben. Der **Combiner** kombiniert anschließend die Instanz, bestehend aus Schemata und Tupel, mit dem invertierten Evolutionsoperator.

Zeile 57-63: Abschließend wird die Anzeige der **Original database** des Evolution-Tabs mit der ausgelesenen Datenbank aktualisiert.

```
1  private void openEvolutionFile(File file) throws NoColumnsException, SQLException,
   TransformationException,
2      JAXBException, ParsingException, Exception {
3
4    InputReader reader = new InputReader();
5
6    try {
7      evolutionInput = reader.readFile(file);
8    } catch (JDOMException | IOException e) {
9      e.printStackTrace();
10     return;
11   }
12
13   origEvolutionInput = evolutionInput;
14
15   // get database schema and instance
16   Connection dbConn = mainController.getDBController().getDbConn();
17   ParserDefaultService parser = new ParserDefaultService();
18   ByteArrayOutputStream parsedByteStream = parser.generateEvolutionXml(dbConn,
19       mainController.getDBController().getAllTables(dbConn));
20   InputReader inputReader = new InputReader();
21   SingleInput input = inputReader.readStream(parsedByteStream);
22
23   // add target schema, schematags and instance tuples to the instance (from
24   // evolution.xml)
25   input.getInstance().getSchema().putAll(evolutionInput.getInstance().getSchema());
26   input.getInstance().getSchemaTags().putAll(evolutionInput.getInstance().getSchemaTags
     ());
27   input.getInstance().getRelationalAtoms().addAll(evolutionInput.getInstance().
     getRelationalAtoms());
28
29   // stream sttgds and tgs for the inverter
30   LinkedHashSet<STTgd> sttgds = new LinkedHashSet<>();
31   sttgds.addAll(this.evolutionInput.getConstraints().stream().filter(con -> (con
     instanceof STTgd))
32       .map(sttgd -> (STTgd) sttgd).collect(Collectors.toSet()));
33
34   LinkedHashSet<Tgd> tgds = new LinkedHashSet<>();
35   tgds.addAll(this.evolutionInput.getConstraints().stream().filter(con -> (con
     instanceof Tgd))
36       .map(tgd -> (Tgd) tgd).collect(Collectors.toSet()));
37
38   Pair<LinkedHashSet<STTgd>, Optional<Tgd>> inverse = InvertingTechniques.invert(sttgds
     , tgds);
39   LinkedHashSet<IntegrityConstraint> constraints = new LinkedHashSet<>();
40   constraints.addAll(inverse.getLeft());
41
42   Optional<Tgd> tgd = inverse.getRight();
43   if (tgd.isEmpty()) {
44     logger.log(Level.FINE, "Inverse does not contain a tgd, skipping...");
45   } else {
46     logger.log(Level.FINE, "Adding tgd to inverse...");
47     constraints.add(tgd.get());
48   }
49
50   // combine schema, tupel and inverted constraints
51   evolutionInput = Combiner.combine(input.getInstance(), constraints, logger);
52
53   IOUtils.saveSingleInput(evolutionInput, Constants.ACTUALS.EVOLUTION_INPUT.toString(),
54       "Could not save evolution input to XML.", logger);
55
56   // generate inverted constraint output in Evolution-Tab
57   String constraintsString = "";
58   System.out.println("-Constraints:");
59   evolutionInput.getConstraints().forEach(c -> System.out.println(c.getTypeName() + ":"
      + SLS + c + SLS));
60   for (IntegrityConstraint c : evolutionInput.getConstraints()) {
61     constraintsString += c.getTypeName() + ": " + SLS + c + SLS + SLS;
62   }
63   evolutionTabShowOperatorTextArea.setText(constraintsString);
64
65   this.file = file;
66 }
```

Abbildung 6.4: openEvolutionFile

handleProcessEvolution (Abbildung 6.5)

Diese Methode wird aufgerufen sobald der Button **Process Evolution** betätigt wird.

Zeile 5-6: Die Abhängigkeiten des evolutionInput werden vorerst durch den **ChaseService** auf Terminierung getestet, wobei in unserem Falle mit s-t tgds kein Test stattfindet, da diese immer terminieren. Für den Fall das auch tgds für Evolution benutzt werden sollen, benötigen wir jedoch die Terminierungstests. Anschließend wird ein **Chase** Objekt mit dem **evolutionInput** als Parametern angelegt. Mit **chase.chase()** wird nun der Chase-Algorithmus ausgeführt und wir erhalten als Ergebnis wieder ein **Instance** Objekt.

Zeile 11-31: Hier wird lediglich die Anzeige für **result Database** des Evolution-Tabs und **Original database** des Chase-Tabs generiert. Dabei werden nur die Schemata ausgegeben, welche das Tag **target** besitzen, da nur diese Tupel enthalten auf denen wir eine Anfrage stellen können. Die **source** Schemata benötigen wir jeodch später für die zweite Evolution.

Zeile 35-37: Für die weitere Nutzung der evolutionierten Datenbankinstanz, werden globale Variablen gesetzt, welche wir später für den Parser benötigen. Dieser arbeitete vorher nur mit der Datenbankverbindung und musste um weitere Angaben angepasst werden. Dazu wurde ein **boolean withEvolution** angelegt, welcher standardmäßig auf **false** gesetzt ist. Falls jedoch eine Evolution stattgefunden hat, dann wird dieser auf **true** gesetzt. Ebenso wird die evolutionierte Datenbankinstanz global gespeichert und das Inverse-Evolution-Tab aktiviert.

Zeile 39-50: Abschließend zur Evolution wird das **Inverse-Evolution Operator (s-t tgd)** Textfeld des Inverse-Evolution-Tabs mit der ursprünglich eingegebenen Evolution gefüllt. Diese ist sogesehen die Inverse der ersten Evolution und setzt die Schemata der Datenbankinstanz wieder auf den aktuellen Stand. Die **evolutionOutput** Instanz wird dann als Zwischenschritt XML gespeichert und der Logger erweitert.

```
1   @FXML
2   void handleProcessEvolution(ActionEvent event) throws Exception {
3
4     // run chase with extended schema Instance and dependencies from evolution.xml
5     ChaseService.runTests(evolutionInput.getInstance(), evolutionInput.getConstraints());
6     Chase chase = new Chase(evolutionInput.getInstance(), evolutionInput.getConstraints()
      );
7
8     Instance evolutionOutput = chase.chase();
9
10    // build string for schema output
11    HashMap<String, SchemaTag> schemaTags = evolutionOutput.getSchemaTags();
12    String schemaOutput = "";
13    for (Entry<String, LinkedHashMap<String, String>> schema : evolutionOutput.getSchema
      ().entrySet()) {
14      if (schemaTags.get(schema.getKey()).getToken().equals("T")) {
15        schemaOutput += schema.getKey() + "(";
16        for (Entry<String, String> attribute : schema.getValue().entrySet()) {
17          schemaOutput += attribute.getKey() + ",";
18        }
19        // remove "," after last attribute
20        schemaOutput = schemaOutput.substring(0, schemaOutput.length() - 1);
21        schemaOutput += ")" + SLS;
22      }
23    }
24
25    // set evolution output in the evolution tab
26    evolutionTabResultSchema.setText(schemaOutput);
27    evolutionTabResultTuples.setText(evolutionOutput.toString());
28
29    // set evolution output in the chase tab
30    chaseTabOrigDatabaseTuplesTextField.setText(evolutionOutput.toString());
31    chaseTabOrigDatabaseSchemaTextField.setText(schemaOutput);
32
33    // preparation for next steps. (withEvolution switch to tell the parser which
34    // schema to use)
35    setWithEvolution(true);
36    this.evolutionOutput = evolutionOutput;
37    invEvolutionTab.setDisable(false);
38
39    String constraintsString = "";
40    System.out.println("-Constraints:");
41    evolutionInput.getConstraints().forEach(c -> System.out.println(c.getTypeName() + ":"
       + SLS + c + SLS));
42    for (IntegrityConstraint c : origEvolutionInput.getConstraints()) {
43      constraintsString += c.getTypeName() + ": " + SLS + c + SLS + SLS;
44    }
45
46    invEvolutionTabOperatorTextfield.setText(constraintsString);
47
48    // save evolution result into src/main/resources/pipelineSteps
49    IOUtils.saveInstanceToXMLWithoutProv(evolutionOutput, Constants.ACTUALS.
      EVOLUTION_RESULT.toString(),
50        "Could not save evolution result to XML.", logger);
51
52  }
```

Abbildung 6.5: handleProcessEvolution

handleInverseEvolution (Abbildung 6.6)

Diese Methode wird aufgerufen, sobald der Button **Process Inverse Evolution** betätigt wird.

Zeile 4-6: Zunächst wird die Eingabe der zweiten Evolution als Zwischenschritt XML gespeichert und der Logger erweitert.

Zeile 8-12: Es wird wieder ein **Chase** Objekt mit **backEvolutionInputInstance** und der Menge an s-t tgds aus **origEvolutionInput** als Parametern angelegt. Mit **chase.chase()** erhalten wir dann das Ergebnis der zweiten Evolution als **Instance** Objekt. Dieses kann das für den optionalen Anonymisierer verwendet werden und wird als Zwischenschritt XML gespeichert und dem Logger hinzugefügt.

Zeile 15-31: Abschließend wird die Ausgabe der Evolution mit Schemata und Tupel generiert und die entsprechenden Textfelder aktualisiert.

```
1   @FXML
2   void handleInverseEvolution(ActionEvent event) throws Exception {
3
4     IOUtils.saveInstanceToXMLWithoutProv(backEvolutionInputInstance,
5         Constants.ACTUALS.BACK_EVOLUTION_INPUT.toString(), "Could not save back evolution
          input to XML.",
6         logger);
7
8     Chase chase = new Chase(backEvolutionInputInstance, origEvolutionInput.getConstraints
      ());
9     Instance backEvolutionOutput = chase.chase();
10
11    IOUtils.saveInstanceToXMLWithoutProv(backEvolutionOutput, Constants.ACTUALS.
      BACK_EVOLUTION_RESULT.toString(),
12        "Could not save back evolution result to XML.", logger);
13
14    // build string for schema output
15    HashMap<String, SchemaTag> schemaTags = evolutionOutput.getSchemaTags();
16    String schemaOutput = "";
17    for (Entry<String, LinkedHashMap<String, String>> schema : backEvolutionOutput.
      getSchema().entrySet()) {
18      if (schemaTags.get(schema.getKey()).getToken().equals("S")) {
19        schemaOutput += schema.getKey() + "(";
20        for (Entry<String, String> attribute : schema.getValue().entrySet()) {
21          schemaOutput += attribute.getKey() + ",";
22        }
23        // remove "," after last attribute
24        schemaOutput = schemaOutput.substring(0, schemaOutput.length() - 1);
25        schemaOutput += ")" + SLS;
26      }
27    }
28
29    invEvolutionTabResultTuplesTextField.setText(backEvolutionOutput.toString());
30
31    invEvolutionTabResultSchemaTextField.setText(schemaOutput);
32
33  }
```

Abbildung 6.6: handleInverseEvolution

6.2.2 ParserDefaultService.java

Für das Auslesen der Datenbank mit generierung der XML und die Anfrageauswertung auf der evolutionierten Datenbankinstanz musste der Parser angepasst werden. Der Parser hat Methoden zum auslesen einer Datenbank implementiert.

generateEvolutionXML (Abbildung 6.7)

Dabei werden die Objekte **dbSchema**, **instance** und **dependencies** angelegt. Da die Methoden jedoch für SQL Anfragen angelegt wurden, müssen wir eine Anfrage angeben, um die **dependencies** anzulegen. Dazu wird vorübergehend die Anfrage **SELECT matricno FROM students** verwendet, welche nur für die aktuelle Datenbankverbindung funktioniert von ProSA. Dies ist jedoch nur eine Zwischenlösung und wird im Laufe der Implementierung noch angepasst.
Als Ergebnis der Methode erhalten wir eine XML-Datei mit Schemata, Tupel und der SqlQuery als Abhängigkeit. Wir benötigen für die Evolution jedoch nur die Schemata und Tupel.

```
1  public ByteArrayOutputStream generateEvolutionXml(Connection dbConn, List<String>
   tables)
2      throws NoColumnsException, SQLException, TransformationException, JAXBException,
       ParsingException {
3
4
5    Select select = SqlQueryParser.parse("SELECT matricno \n FROM students");
6
7    DatabaseReader dbReader = new DatabaseReader(dbConn);
8    DBSchema dbSchema = dbReader.readSchema(DEFAULT_SCHEMA, tables);
9    Instance instance = dbReader.readInstance(DEFAULT_SCHEMA, tables);
10
11   Dependencies dependencies = Transformation.transform(select, dbSchema);
12   Input xmlRootElement = Compositor.buildXml(dependencies, dbSchema, instance);
13
14   return XmlSerializer.marshal(xmlRootElement);
15 }
```

Abbildung 6.7: generateEvolutionXML

generateXmlAfterEvolution (Abbildung 6.8)

Nachdem die Datenbankinstanz evolutioniert wurde, muss der Parser diese statt der Datenbankinstanz der Datenbankverbindung für die Anfrageauswertung verwenden. Um die Methoden **Transformation.transform** und **Compositor.buildXML** weiter zu verwenden, muss die Datenbankinstanz vorerst transformiert werden. Dazu wurden die Methoden **conertInstance** (Abbildung 6.10) und **extractTargetRelationsFromInstance** (Abbildung 6.9) implementiert. Wir erhalten dadaurch eine XML Datei, welche die Schemata und Tupel der evolutionerten Datenbankinstanz und die eingebene Anfrage als Menge von Abhängigkeiten enthält. Diese wird dann für den Chase verwendet und stellt die Anfrageauswertung dar.

```
1  public ByteArrayOutputStream generateXmlAfterEvolution(String query,
2      instance.Instance evolutionOutput)
3      throws NoColumnsException, SQLException, TransformationException, JAXBException,
       ParsingException {
4
5    Select select = SqlQueryParser.parse(query);
6
7    DBSchema dbSchema = new DBSchema();
8    Instance instance = convertInstance(evolutionOutput);
9
10   // add evolutionOutput schema to dbSchema
11   extractTargetRelationsFromInstance(dbSchema, evolutionOutput);
12
13   Dependencies dependencies = Transformation.transform(select, dbSchema);
14   Input xmlRootElement = Compositor.buildXml(dependencies, dbSchema, instance);
15
16   return XmlSerializer.marshal(xmlRootElement);
17 }
```

Abbildung 6.8: generateXmlAfterEvolution

extractTargetRelationsFromInstance (Abbildung 6.9)

Der Parser erwartet als Datenbankinstanz eine Datenbankverbindung. Da wir jedoch lediglich ein **Instance** Objekt der evolutionierten Datenbank besitzen, müssen wir diese als **DBSchema** Objekt darstellen. Dazu legen wir für jedes Schema ein **Relation** Objekt an und kopieren den Namen, das Tag und die einzelnen Attribute. Anschließend wird die angelegte Relation zum **DBSchema** hinzugefügt und kann somit für die bereits implementierten Methoden verwendet werden.

```
1  public DBSchema extractTargetRelationsFromInstance(DBSchema dbSchema, instance.Instance
    instance) {
2    HashMap<String, SchemaTag> schemaTags = instance.getSchemaTags();
3    for (Entry<String, LinkedHashMap<String, String>> schema : instance.getSchema().
     entrySet()) {
4
5      // Only use Target tagged schema (others are empty anyways)
6      if (schemaTags.get(schema.getKey()).getToken().equals("T")) {
7
8        // create Relation, needed for DBSchema
9        Relation rel = new Relation();
10       rel.setName(schema.getKey());
11       rel.setTag(Tag.fromValue("S"));
12
13       // extract attributes and convert them for relation
14       List<Attribute> attributes = rel.getAttribute();
15
16       for (Entry<String, String> attribute : schema.getValue().entrySet()) {
17         Attribute newAttribute = new Attribute();
18         newAttribute.setName(attribute.getKey());
19         newAttribute.setType(AttributeType.fromValue(attribute.getValue()));
20         attributes.add(newAttribute);
21       }
22       dbSchema.putTable(rel);
23     }
24   }
25
26   return dbSchema;
27 }
```

Abbildung 6.9: extractTargetRelationsFromInstance

convertInstance (Abbildung 6.10)

Nun müssen die Tupel der evolutionierten Datenbankinstanz von **instance.instance** zu **xml.instance** konvertiert werden. Dazu wird ein **xml.instance** Objekt angelegt und die benötigten Werte aus dem **evolutionOutput** kopiert. Durch die Konvertierung kann nun die bereits implementierte Methode **Compositor.buildXml** verwendet werden.

```
1   public xml.Instance convertInstance(instance.Instance evolutionOutput) {
2
3   xml.Instance convertedInstance = new xml.Instance();
4   int i = 1;
5   for (RelationalAtom relAtom : evolutionOutput.getRelationalAtoms()) {
6
7     InstanceAtom convertedAtom = new InstanceAtom();
8     convertedAtom.setName(relAtom.getRelationName());
9     convertedAtom.setId(relAtom.getRelationName() + "_" + i);
10
11    for(Term term : relAtom.getTerms()) {
12
13        xml.Constant xmlConstant = new xml.Constant();
14
15        xmlConstant.setName(term.getName());
16        xmlConstant.setValue(term.toString().replace("\"", ""));
17
18        convertedAtom.getConstant().add(xmlConstant);
19    }
20
21    convertedInstance.getAtom().add(convertedAtom);
22    i++;
23  }
24
25
26  return convertedInstance;
27
28 }
```

Abbildung 6.10: convertInstance

Somit wurden folgende Schritte zur Implementierung der Evolution abgearbeitet:

1. Anpassung der Benutzeroberfläche
2. Einlesen der XML-Datei
3. Invertierung der Menge an s-t tgds
4. Anpassung der Eingabe und Aufruf von ChaTEAU für die erste Evolution
5. Ausgabe der ersten Evolution
6. Übergabe an den Parser zur Anfrageauswertung
7. Anpassung der Eingabe und Aufruf von ChaTEAU für die zweite Evolution
8. Ausgabe der zweiten Evolution

6.3 Beispiel Durchlauf

Im folgenden Beispiel durchlaufen wir die Evolution in ProSA. Dafür verwenden wir eine Datenbank D_{t+1}, eine Evolution E und eine Anfrage Q. Die schreibweise der Schemata und der s-t tgds entstammen dabei der Ausgabe von ProSA. Gegeben seien:

- Die Evolution $E = (S_t, S_{t+1}, \Sigma)$ als CopyColumn.xml (Anhang 8.11).
  Dabei ist Σ eine Menge von s-t tgds und wie folgt definiert:
  $\Sigma = tutorsWithTitle(\#V\_moduleno\_1, \#V\_title\_1, \#V\_tutor\_1, \#V\_id\_tutors\_1) \rightarrow$
  $modules(\#V\_moduleno\_1, \#V\_title\_1, \#E\_major\_1, \#E\_id\_modules\_1) \wedge$
  $tutors(\#V\_moduleno\_1, \#V\_tutor\_1, \#V\_id\_tutors\_1),$

  $modulesAlt(\#V\_moduleno\_1, \#V\_title\_1, \#E\_major\_1, \#E\_id\_modules\_1) \rightarrow$
  $modules(\#V\_moduleno\_1, \#V\_title\_1, \#E\_major\_1, \#E\_id\_modules\_1)$
- Die Datenbank D_{t+1} mit dem Schema S_{t+1} bestehend aus:
  $tutors(moduleno, tutor, id\_modules)$
  $students(matricno, lastname, firstname, studycourse, id\_students)$
  $grades(moduleno, matricno, semester, grade, id\_grades)$
  $modules(moduleno, title, major, id\_modules)$
  $participants(moduleno, matricno, id\_participants)$
- Die Anfrage Q zur Datenbank D_t mit Schema S_t:
  SELECT moduleno, title, tutor
  FROM tutorsWithTitle

Ziel ist es nun die Anfrage Q erneut stellen zu können und für diese eine minimale Teildatenbank von D_{t+1} zu erstellen.

1. **Eingabe**: Wir benutzen die Datenbankverbindung von D_{t+1} und die Evolution $E = (S_t, S_{t+1}, \Sigma)$ (Siehe 8.11) als Eingabe im Configuration-Tab. Wir erhalten eine Übersicht der Evolution im Evolution-Tab. Diese besteht aus der invertierten Evolution $E^{-1} = (S_{t+1}, S_t, \Sigma^{-1})$, den Schemata und den Tupeln der Datenbank D_{t+1}. Dabei sieht Σ^{-1} wir folgt aus:
   $\Sigma^{-1} = modules(\#V\_moduleno\_1, \#V\_title\_1, \#V\_major\_1, \#V\_id\_modules\_1) \wedge$
   $tutors(\#V\_moduleno\_1, \#V\_tutor\_1, \#V\_id\_tutors\_1) \rightarrow$
   $tutorsWithTitle(\#V\_moduleno\_1, \#V\_title\_1, \#V\_tutor\_1, \#V\_id\_tutors\_1),$

   $modules(\#V\_moduleno\_1, \#V\_title\_1, \#V\_major\_1, \#V\_id\_modules\_1) \rightarrow$
   $modulesAlt(\#V\_moduleno\_1, \#V\_title\_1, \#V\_major\_1, \#V\_id\_modules\_1)$

2. **Evolution**: Mit dem Button **Process Evolution** können wir nun die Evolution E^{-1} auf die Datenbank D_{t+1} anwenden und erhalten die Datenbank D_t mit dem Schema S_t bestehend aus:
   $tutorsWithTitle(moduleno, title, tutor, id\_modules)$
   $students(matricno, lastname, firstname, studycourse, id\_students)$
   $grades(moduleno, matricno, semester, grade, id\_grades)$
   $modulesAlt(moduleno, title, major, id\_modules)$
   $participants(moduleno, matricno, id\_participants)$

3. **Chase**: Da wir nun die Datenbank D_t mit dem Schema S_t erzeugt haben, können wir die Anfrage Q stellen. Die Anfrage Q wird dabei als s-t tgd definiert und sieht wie folgt aus:
   $tutorsWithTitle(\#V\_moduleno\_1, \#V\_title\_1, \#V\_tutor\_1, \#V\_id\_tutors\_1) \rightarrow$
   $Result(\#V\_moduleno\_1, \#V\_title\_1, \#V\_tutor\_1)$
   Dadurch erhalten wir eine Instanz I über dem Schema S_Q :
   $Result(moduleno, title, tutor)$

4. **Backchase**: Der Backchase invertiert nun die Anfrage Q und stellt die resultierende Anfrage Q^{-1} an die Instanz I. Dabei sieht die s-t tgd von Q^{-1} wie folgt aus:
   $Result(\#V\_moduleno\_1, \#V\_title\_1, \#V\_tutor\_1) \rightarrow$
   $tutorsWithTitle(\#V\_moduleno\_1, \#V\_title\_1, \#V\_tutor\_1, \#V\_id\_tutors\_1)$
   Wir erhalten eine minimale Teildatenbank $D_{t_{min}}$ der Datenbank D_t zur Anfrage Q.

5. **Inverse-Evolution**: Wir wenden nun mit dem Button **Process Inverse Evolution** die ursprünglich eingegebene Evolution E auf die erzeugte minimale Teildatenbank $D_{t_{min}}$ an. Die Menge an s-t tgds enspricht dabei dem vorher definierten Σ. Dadurch erhalten wir eine minimale Teildatenbank $D_{t+1_{min}}$ über dem Schema:
   $tutors(moduleno, tutor, id\_tutors)$
   $modules(moduleno, title, major, id\_modules)$

6. **Ergebnis**: Als Ergebnis haben wir nun:

   - Eine minimale Teildatenbank $D_{t+1_{min}}$ der Datenbank D_{t+1}
   - Die Evolution E
   - Die Anfrage Q
   - Das Ergebnis der Anfrage Q zur Datenbank D_t

Somit können wir das Ergebnis der Anfrage Q jederzeit Reproduzieren, ohne die Datenbank D_t oder D_{t+1} zu besitzen. Dazu muss die Eingabe von ProSA lediglich auf die Datenbank $D_{t+1_{min}}$ statt der vorher genutzten Datenbank D_{t+1} gesetzt werden.

7 Zusammenfassung und Ausblick

In diesem Kapitel wird eine inhaltliche Zusammenfassung (Abschnitt 7.1) dieser Arbeit gegeben. Abschließend werden im Ausblick (Abschnitt 7.2) mögliche Erweiterungen der Evolution vorgezeigt, welche nicht Bestandteil dieser Arbeit waren oder noch nicht umgesetzt werden konnten.

7.1 Zusammenfassung

In dieser Arbeit wurde die Erweiterung der ProSA-Pipeline um die Verarbeitung einzelner Schemaevolutionschritte vorgestellt. Dazu haben wir uns bereits existierende Techniken und Forschungen zum Thema Evolution durch den Chase-Algorithmus angesehen. Daraus erhielten wir die Darstellung der SMOs als Mengen von s-t tgds (Siehe 4.1) und deren Inversen (Siehe 4.2). Dadurch konnten einzelne SMOs in ChaTEAU getestet werden und erhielten somit die innerhalb von ProSA zu generierende Eingabe für ChaTEAU (Siehe 5.1). Nachdem dann der geplante Ablauf für ProSA mit Evolution (Siehe 3.4) erweitert wurde, stand die Theorie zur Implementierung (Siehe 6). In Abschnitt 6.1 wurde zunächst die angepasste Benutzeroberfläche vorgestellt. Anschließend wurden in Abschnitt 6.2 die erweiterten Komponenten von ProSA gezeigt und die einzelnen Schritte genauer erklärt. Abschließend wurde die implementierte Schemavolution anhand eines Anwendungsbeispiels (Siehe 6.3) vorgestellt.

7.2 Ausblick

Hinzufügen weiterer Schritte der Evolution: In dieser Arbeit wurde lediglich die Anwendung einer SMO als XML-Datei betrachtet. Für die Verarbeitung weiterer Schritte, müsste die Eingabe für mehrere Dateien ermöglicht werden. Dabei muss eine bestimmte Reihenfolge zur Anwendung der SMOs eingehalten werden, da sonst verschiedene Ergebnisse oder Fehler auftreten können. Neben der Invertierung der einzelnen SMOs, muss auch die Reihenfolge der Anwendung dieser angepasst werden.

Evolution mit Vergleichen und Funktionen: In Abschnitt 5.1 wurden die Operatoren `PARTITION Table`, `MERGE Column`, `RENAME Column` und `SPLIT Column` vorgestellt. Diese enthalten Vergleiche und Funktionen, welche in ChaTEAU (Version 1.1) nicht implementiert sind. Demnach konnten die Operatoren in dieser Arbeit nicht getestet werden. Ob die Implementierung der Evolution nach Versionsänderungen von ChaTEAU angepasst werden muss, ist jedoch noch nicht eindeutig absehbar.

8 Anhang

```
<input>
  <schema>
    <relations>
      <relation name="students" tag="S">
        <attribute name="matricno" type="int" />
        <attribute name="lastname" type="string" />
        <attribute name="firstname" type="string" />
        <attribute name="studycourse" type="string" />
        <attribute name="id_students" type="string" />
      </relation>
      <relation name="studentsOrig" tag="T">
        <attribute name="matricno" type="int" />
        <attribute name="lastname" type="string" />
        <attribute name="firstname" type="string" />
        <attribute name="studycourse" type="string" />
        <attribute name="id_students" type="string" />
      </relation>
      <relation name="studentsCopy" tag="T">
        <attribute name="matricno" type="int" />
        <attribute name="lastname" type="string" />
        <attribute name="firstname" type="string" />
        <attribute name="studycourse" type="string" />
        <attribute name="id_students" type="string" />
      </relation>
    </relations>
    <dependencies>
      <sttgd>
        <body>
          <atom name="students">
            <variable name="matricno" type="V" index="1" />
            <variable name="lastname" type="V" index="1" />
            <variable name="firstname" type="V" index="1" />
            <variable name="studycourse" type="V" index="1" />
            <variable name="id_students" type="V" index="1" />
          </atom>
        </body>
        <head>
          <atom name="studentsOrig">
            <variable name="matricno" type="V" index="1" />
            <variable name="lastname" type="V" index="1" />
            <variable name="firstname" type="V" index="1" />
            <variable name="studycourse" type="V" index="1" />
            <variable name="id_students" type="V" index="1" />
          </atom>
          <atom name="studentsCopy">
            <variable name="matricno" type="V" index="1" />
            <variable name="lastname" type="V" index="1" />
            <variable name="firstname" type="V" index="1" />
            <variable name="studycourse" type="V" index="1" />
            <variable name="id_students" type="V" index="1" />
          </atom>
        </head>
      </sttgd>
    </dependencies>
  </schema>
  <instance></instance>
</input>
```

Abbildung 8.1: COPY Table für Students Relation

```
<input>
  <schema>
    <relations>
      <relation name="trigger" tag="S">
        <attribute name="trigger" type="int" />
      </relation>
      <relation name="grades" tag="T">
        <attribute name="moduleno" type="int" />
        <attribute name="matricno" type="int" />
        <attribute name="semester" type="string" />
        <attribute name="grade" type="double" />
        <attribute name="id_grades" type="string" />
      </relation>
    </relations>
    <dependencies>
      <sttgd>
        <body>
          <atom name="trigger">
            <variable name="trigger" type="V" index="1" />
          </atom>
        </body>
        <head>
          <atom name="grades">
            <variable name="moduleno" type="E" index="1" />
            <variable name="matricno" type="E" index="1" />
            <variable name="semester" type="E" index="1" />
            <variable name="grade" type="E" index="1" />
            <variable name="id_grades" type="E" index="1" />
          </atom>
        </head>
      </sttgd>
    </dependencies>
  </schema>
  <instance>
    <atom name="trigger">
      <constant name="trigger" value="1" />
    </atom>
  </instance>
</input>
```

Abbildung 8.2: CREATE Table für grades Relation

```
<input>
  <schema>
    <relations>
      <relation name="students" tag="S">
        <attribute name="matricno" type="int" />
        <attribute name="lastname" type="string" />
        <attribute name="firstname" type="string" />
        <attribute name="studycourse" type="string" />
        <attribute name="id_students" type="string" />
      </relation>
      <relation name="studentsName" tag="T">
        <attribute name="matricno" type="int" />
        <attribute name="lastname" type="string" />
        <attribute name="firstname" type="string" />
      </relation>
      <relation name="studentsCourse" tag="T">
        <attribute name="matricno" type="int" />
        <attribute name="studycourse" type="string" />
      </relation>
    </relations>
    <dependencies>
      <sttgd>
        <body>
          <atom name="students">
            <variable name="matricno" type="V" index="1" />
            <variable name="lastname" type="V" index="1" />
            <variable name="firstname" type="V" index="1" />
            <variable name="studycourse" type="V" index="1" />
            <variable name="id_students" type="V" index="1" />
          </atom>
        </body>
        <head>
          <atom name="studentsName">
            <variable name="matricno" type="V" index="1" />
            <variable name="lastname" type="V" index="1" />
            <variable name="firstname" type="V" index="1" />
          </atom>
          <atom name="studentsCourse">
            <variable name="matricno" type="V" index="1" />
            <variable name="studycourse" type="V" index="1" />
          </atom>
        </head>
      </sttgd>
    </dependencies>
  </schema>
  <instance></instance>
</input>
```

Abbildung 8.3: DECOMPOSE Table auf students Relation

```
<input>
  <schema>
    <relations>
      <relation name="modules" tag="S">
        <attribute name="moduleno" type="int" />
        <attribute name="title" type="string" />
        <attribute name="major" type="string" />
        <attribute name="id_modules" type="string" />
      </relation>
      <relation name="participants" tag="S">
        <attribute name="moduleno" type="int" />
        <attribute name="matricno" type="int" />
        <attribute name="id_participants" type="string" />
      </relation>
      <relation name="participantsWithTitle" tag="T">
        <attribute name="moduleno" type="int" />
        <attribute name="title" type="string" />
        <attribute name="matricno" type="int" />
      </relation>
    </relations>
    <dependencies>
      <sttgd>
        <body>
          <atom name="modules">
            <variable name="moduleno" type="V" index="1" />
            <variable name="title" type="V" index="1" />
            <variable name="major" type="V" index="1" />
            <variable name="id_modules" type="V" index="1" />
          </atom>
          <atom name="participants">
            <variable name="moduleno" type="V" index="1" />
            <variable name="matricno" type="V" index="1" />
            <variable name="id_participants" type="V" index="1" />
          </atom>
        </body>
        <head>
          <atom name="participantsWithTitle">
            <variable name="moduleno" type="V" index="1" />
            <variable name="title" type="V" index="1" />
            <variable name="matricno" type="V" index="1" />
          </atom>
        </head>
      </sttgd>
    </dependencies>
  </schema>
  <instance>
  </instance>
</input>
```

Abbildung 8.4: JOIN Table auf modules und participants Relationen

```
<input>
  <schema>
    <relations>
      <relation name="Students1" tag="S">
        <attribute name="student_id" type="int" />
        <attribute name="lastname" type="string" />
        <attribute name="firstname" type="string" />
        <attribute name="course" type="string" />
      </relation>
      <relation name="Students2" tag="S">
        <attribute name="student_id" type="int" />
        <attribute name="lastname" type="string" />
        <attribute name="firstname" type="string" />
        <attribute name="course" type="string" />
      </relation>
      <relation name="Students" tag="T">
        <attribute name="student_id" type="int" />
        <attribute name="lastname" type="string" />
        <attribute name="firstname" type="string" />
        <attribute name="course" type="string" />
      </relation>
    </relations>
    <dependencies>
      <sttgd>
        <body>
          <atom name="Students1">
            <variable name="student_id" type="V" index="1" />
            <variable name="lastname" type="V" index="1" />
            <variable name="firstname" type="V" index="1" />
            <variable name="course" type="V" index="1" />
          </atom>
        </body>
        <head>
          <atom name="Students">
            <variable name="student_id" type="V" index="1" />
            <variable name="lastname" type="V" index="1" />
            <variable name="firstname" type="V" index="1" />
            <variable name="course" type="V" index="1" />
          </atom>
        </head>
      </sttgd>
      <sttgd>
        <body>
          <atom name="Students2">
            <variable name="student_id" type="V" index="1" />
            <variable name="lastname" type="V" index="1" />
            <variable name="firstname" type="V" index="1" />
            <variable name="course" type="V" index="1" />
          </atom>
        </body>
        <head>
          <atom name="Students">
            <variable name="student_id" type="V" index="1" />
            <variable name="lastname" type="V" index="1" />
            <variable name="firstname" type="V" index="1" />
            <variable name="course" type="V" index="1" />
          </atom>
        </head>
      </sttgd>
    </dependencies>
  </schema>
  <instance></instance>
</input>
```

Abbildung 8.5: MERGE Table auf Students Relationen

```
<input>
  <schema>
    <relations>
      <relation name="students" tag="S">
        <attribute name="matricno" type="int" />
        <attribute name="lastname" type="string" />
        <attribute name="firstname" type="string" />
        <attribute name="studycourse" type="string" />
        <attribute name="id_students" type="string" />
      </relation>
      <relation name="studenten" tag="T">
        <attribute name="matricno" type="int" />
        <attribute name="lastname" type="string" />
        <attribute name="firstname" type="string" />
        <attribute name="studycourse" type="string" />
        <attribute name="id_students" type="string" />
      </relation>
    </relations>
    <dependencies>
      <sttgd>
        <body>
          <atom name="students">
            <variable name="matricno" type="V" index="1" />
            <variable name="lastname" type="V" index="1" />
            <variable name="firstname" type="V" index="1" />
            <variable name="studycourse" type="V" index="1" />
            <variable name="id_students" type="V" index="1" />
          </atom>
        </body>
        <head>
          <atom name="studenten">
            <variable name="matricno" type="V" index="1" />
            <variable name="lastname" type="V" index="1" />
            <variable name="firstname" type="V" index="1" />
            <variable name="studycourse" type="V" index="1" />
            <variable name="id_students" type="V" index="1" />
          </atom>
        </head>
      </sttgd>
    </dependencies>
  </schema>
  <instance>
  </instance>
</input>
```

Abbildung 8.6: RENAME Table auf Students Relation zu Studenten Relation

```
<input>
  <schema>
    <relations>
      <relation name="students" tag="S">
        <attribute name="matricno" type="int" />
        <attribute name="lastname" type="string" />
        <attribute name="firstname" type="string" />
        <attribute name="studycourse" type="string" />
        <attribute name="id_students" type="string" />
      </relation>
      <relation name="studentsAdded" tag="T">
        <attribute name="matricno" type="int" />
        <attribute name="lastname" type="string" />
        <attribute name="firstname" type="string" />
        <attribute name="studycourse" type="string" />
        <attribute name="semester" type="int" />
        <attribute name="id_students" type="string" />
      </relation>
    </relations>
    <dependencies>
      <sttgd>
        <body>
          <atom name="students">
            <variable name="matricno" type="V" index="1" />
            <variable name="lastname" type="V" index="1" />
            <variable name="firstname" type="V" index="1" />
            <variable name="studycourse" type="V" index="1" />
            <variable name="id_students" type="V" index="1" />
          </atom>
        </body>
        <head>
          <atom name="studentsAdded">
            <variable name="matricno" type="V" index="1" />
            <variable name="lastname" type="V" index="1" />
            <variable name="firstname" type="V" index="1" />
            <variable name="studycourse" type="V" index="1" />
            <variable name="semester" type="E" index="1" />
            <variable name="id_students" type="V" index="1" />
          </atom>
        </head>
      </sttgd>
    </dependencies>
  </schema>
  <instance></instance>
</input>
```

Abbildung 8.7: ADD Column zur Erweiterung der Students Relation um semester

```
<input>
  <schema>
    <relations>
      <relation name="tutors" tag="S">
        <attribute name="moduleno" type="int" />
        <attribute name="tutor" type="string" />
        <attribute name="id_tutors" type="string" />
      </relation>
      <relation name="modules" tag="S">
        <attribute name="moduleno" type="int" />
        <attribute name="title" type="string" />
        <attribute name="major" type="string" />
        <attribute name="id_modules" type="string" />
      </relation>
      <relation name="modulesNew" tag="T">
        <attribute name="moduleno" type="int" />
        <attribute name="title" type="string" />
        <attribute name="major" type="string" />
        <attribute name="id_modules" type="string" />
      </relation>
      <relation name="tutorsWithTitle" tag="T">
        <attribute name="moduleno" type="int" />
        <attribute name="title" type="string" />
        <attribute name="tutor" type="string" />
        <attribute name="id_tutors" type="string" />
      </relation>
    </relations>
    <dependencies>
      <sttgd>
        <body>
          <atom name="modules">
            <variable name="moduleno" type="V" index="1" />
            <variable name="title" type="V" index="1" />
            <variable name="major" type="V" index="1" />
            <variable name="id_modules" type="V" index="1" />
          </atom>
          <atom name="tutors">
            <variable name="moduleno" type="V" index="1" />
            <variable name="tutor" type="V" index="1" />
            <variable name="id_tutors" type="V" index="1" />
          </atom>
        </body>
        <head>
            <atom name="modulesNew">
            <variable name="moduleno" type="V" index="1" />
            <variable name="title" type="V" index="1" />
            <variable name="major" type="V" index="1" />
            <variable name="id_modules" type="V" index="1" />
          </atom>
          <atom name="tutorsWithTitle">
            <variable name="moduleno" type="V" index="1" />
            <variable name="title" type="V" index="1" />
            <variable name="tutor" type="V" index="1" />
            <variable name="id_tutors" type="V" index="1" />
          </atom>
        </head>
      </sttgd>
    </dependencies>
  </schema>
  <instance></instance>
</input>
```

Abbildung 8.8: COPY Column auf title von Modules zu Tutors

```
<input>
  <schema>
    <relations>
      <relation name="students" tag="S">
        <attribute name="matricno" type="int" />
        <attribute name="lastname" type="string" />
        <attribute name="firstname" type="string" />
        <attribute name="studycourse" type="string" />
        <attribute name="id_students" type="string" />
      </relation>
      <relation name="studentsDropped" tag="T">
        <attribute name="matricno" type="int" />
        <attribute name="lastname" type="string" />
        <attribute name="firstname" type="string" />
        <attribute name="id_students" type="string" />
      </relation>
    </relations>
    <dependencies>
      <sttgd>
        <body>
          <atom name="students">
            <variable name="matricno" type="V" index="1" />
            <variable name="lastname" type="V" index="1" />
            <variable name="firstname" type="V" index="1" />
            <variable name="studycourse" type="V" index="1" />
            <variable name="id_students" type="V" index="1" />
          </atom>
        </body>
        <head>
          <atom name="studentsDropped">
            <variable name="matricno" type="V" index="1" />
            <variable name="lastname" type="V" index="1" />
            <variable name="firstname" type="V" index="1" />
            <variable name="id_students" type="V" index="1" />
          </atom>
        </head>
      </sttgd>
    </dependencies>
  </schema>
  <instance></instance>
</input>
```

Abbildung 8.9: DROP Column auf studycourse in Students

```
<input>
  <schema>
    <relations>
      <relation name="tutors" tag="S">
        <attribute name="moduleno" type="int" />
        <attribute name="tutor" type="string" />
        <attribute name="id_tutors" type="string" />
      </relation>
      <relation name="modules" tag="S">
        <attribute name="moduleno" type="int" />
        <attribute name="title" type="string" />
        <attribute name="major" type="string" />
        <attribute name="id_modules" type="string" />
      </relation>
      <relation name="modulesWithTutor" tag="T">
        <attribute name="moduleno" type="int" />
        <attribute name="title" type="string" />
        <attribute name="major" type="string" />
        <attribute name="tutor" type="string" />
        <attribute name="id_modules" type="string" />
      </relation>
    </relations>
    <dependencies>
      <sttgd>
        <body>
          <atom name="modules">
            <variable name="moduleno" type="V" index="1" />
            <variable name="title" type="V" index="1" />
            <variable name="major" type="V" index="1" />
            <variable name="id_modules" type="V" index="1" />
          </atom>
          <atom name="tutors">
            <variable name="moduleno" type="V" index="1" />
            <variable name="tutor" type="V" index="1" />
            <variable name="id_tutors" type="V" index="1" />
          </atom>
        </body>
        <head>
          <atom name="modulesWithTutor">
            <variable name="moduleno" type="V" index="1" />
            <variable name="title" type="V" index="1" />
            <variable name="major" type="V" index="1" />
            <variable name="tutor" type="V" index="1" />
            <variable name="id_modules" type="V" index="1" />
          </atom>
        </head>
      </sttgd>
    </dependencies>
  </schema>
  <instance></instance>
</input>
```

Abbildung 8.10: MOVE Column von tutor aus Tutors zu Modules

```
<input>
  <schema>
    <relations>
      <relation name="tutors" tag="T">
        <attribute name="moduleno" type="int" />
        <attribute name="tutor" type="string" />
        <attribute name="id_tutors" type="string" />
      </relation>
      <relation name="modules" tag="T">
        <attribute name="moduleno" type="int" />
        <attribute name="title" type="string" />
        <attribute name="major" type="string" />
        <attribute name="id_modules" type="string" />
      </relation>
      <relation name="tutorsWithTitle" tag="S">
        <attribute name="moduleno" type="int" />
        <attribute name="title" type="string" />
        <attribute name="tutor" type="string" />
        <attribute name="id_tutors" type="string" />
      </relation>
      <relation name="modulesAlt" tag="S">
        <attribute name="moduleno" type="int" />
        <attribute name="title" type="string" />
        <attribute name="major" type="string" />
        <attribute name="id_modules" type="string" />
      </relation>
    </relations>
    <dependencies>
      <sttgd>
        <body>
          <atom name="tutorsWithTitle">
            <variable name="moduleno" type="V" index="1" />
            <variable name="title" type="V" index="1" />
            <variable name="tutor" type="V" index="1" />
            <variable name="id_tutors" type="V" index="1" />
          </atom>
        </body>
        <head>
          <atom name="modules">
            <variable name="moduleno" type="V" index="1" />
            <variable name="title" type="V" index="1" />
            <variable name="major" type="E" index="1" />
            <variable name="id_modules" type="E" index="1" />
          </atom>
          <atom name="tutors">
            <variable name="moduleno" type="V" index="1" />
            <variable name="tutor" type="V" index="1" />
            <variable name="id_tutors" type="V" index="1" />
          </atom>
        </head>
      </sttgd>
      <sttgd>
        <body>
          <atom name="modulesAlt">
            <variable name="moduleno" type="V" index="1" />
            <variable name="title" type="V" index="1" />
            <variable name="major" type="E" index="1" />
            <variable name="id_modules" type="E" index="1" />
          </atom>
        </body>
        <head>
          <atom name="modules">
            <variable name="moduleno" type="V" index="1" />
            <variable name="title" type="V" index="1" />
            <variable name="major" type="E" index="1" />
            <variable name="id_modules" type="E" index="1" />
          </atom>
        </head>
      </sttgd>
    </dependencies>
  </schema>
  <instance></instance>
</input>
```

Abbildung 8.11: CopyColumn.xml als Beispiel

Datenträger Anhang

Die Abgabe dieser Arbeit erfolgt im digitalen Format. Dafür wird die verwendete Literatur, die TeX-Dateien dieser Bachelorarbeit und die Implementierung als Quelltext online bereitgestellt.

Studip-Veranstaltung: Erreichbar unter folgendem Link:
https://studip.uni-rostock.de/dispatch.php/course/overview?cid=40cf5874f6c5e1e0919dae32bd5deb4a
Die Studip-Veranstaltung beinhaltet im Ordner **Literatur** die verwendete Literatur im pdf Format und die TeX-Dateien dieser Bachelorarbeit befinden sich im Ordner **Weitere Dokumente**. Das Buch [HSS18] kann jedoch nicht bereitgestellt werden.

GitLab: Die Implementierung als Quelltext befindet sich auf GitLab und ist unter folgendem Link erreichbar:
https://git.informatik.uni-rostock.de/ta093/prosa/-/tree/eric-prosa-evolution

Literaturverzeichnis

[AHH22] Auge, Tanja ; Hanzig, Moritz ; Heuer, Andreas: ProSA Pipeline: Provenance conquers the Chase. In: *Universität Rostock, ADBIS* (2022)

[Aug20] Auge, Tanja: Extended Provenance Management for Data Science Applications. In: *VLDB 2020 PhD Workshop, August 31st, 2020* (2020)

[Aug22] Auge, Tanja: ProSA-Notizen aus der laufenden Dissertation, Unveröffentlicht, 2022

[CMHZ09] Curino, Carlo A. ; Moon, Hyun J. ; Ham, MyungWon ; Zaniolo, Carlo: The PRISM Workwench: Database Schema Evolution Without Tears. In: *DEMO paper at ICDE* (2009)

[CMZ08] Curino, Carlo A. ; Moon, Hyun J. ; Zaniolo, Carlo: Schema Evolution in Wikipedia - Toward a Web Information System Benchmark. In: *Conference Paper DBLP* (January 2008)

[CTMZ08] Curino, Carlo A. ; Tanca, Lezizia ; Moon, Hyun J. ; Zaniolo, Carlo: Graceful database schema evolution: the 568 PRISM workbench. In: *VLDB Endow., 1(1):761–772* (2008)

[GMS12] Greco, S. ; Molinaro, C. ; Spezzano, F.: Incomplete Data and Data Dependencies in Relational Databases. In: *Synthesis Lectures on Data Management, Morgan&Claypool Publishers* (2012)

[Han22] Hanzig, Moritz: Ein Framework für ProSA. In: *Bachelorarbeit, Universität Rostock, DBIS* (2022)

[HSS18] Heuer, Andreas ; Saake, Gunter ; Sattler, Kai-Uwe: *Datenbanken - Konzepte und Sprachen, 6. Auflage.* MITP, 2018

[Man20] Manthey, Erik: Beschreibung der Veränderungen von Schemata und Daten am IOW mit Schema- Evolutions-Operatoren. In: *Bachelorarbeit* (2020)

[Ros20] Rose, Florian: Erweiterung des CHASE-Werkzeugs ChaTEAU um eine BACKCHASE-Phase. In: *Masterarbeit, Universität Rostock, DBIS* (2020)

[Sch21] Scharlau, Nic: CHASE - Ein Überblick. In: *Gebietsseminar, Universität Rostock, DBIS* (2021)

[Spo22] Spolwind, Dennis: Inverse Anfragen in ProSA. In: *Masterarbeit, Universität Rostock, DBIS* (2022)

[WWO+21] Wolpers, Anja ; Waterstradt, Anne-Sophie ; Overath, Judith-Henrike ; Heuser, Melinda ; Förster, Leonie: Data Provenance. In: *KSWS* (2021)

Abbildungsverzeichnis

Selbstständigkeitserklärung

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

Rostock, den 30. August 2022