Projektdokumentation

Gruppe 1
KSWS Data Provenance

Anja Wolpers, Anne-Sophie Waterstradt, Judith-Henrike Overath,
Melinda Heuser, Leonie Förster

Inhaltsverzeichnis

1. Einleitung

Anne-Sophie Waterstradt

Wir werden eine grafische Benutzeroberfläche für das Forschungsprojekt ProSA entwickeln. Das Ziel von ProSA ist die Bestimmung der für ein Anfrageergebnis notwendigen, minimalen Quelldaten im Rahmen von Provenance Management. ProSA nutzt das Java-Tool ChaTEAU, um aus der Quell-Datenbank und einer SQL-Anfrage die minimale Teildatenbank zu bestimmen.

ChaTEAU bekommt eine s-t tgd in Form eines XML-Dokuments und auf dessen Basis wird mit seiner Implementierung der Chase-Algorithmus zwei Mal ausgeführt. Nach der ersten Ausführung hat man das Anfrageergebnis beziehungsweise die Ergebnisinstanz. Beim zweiten Ausführen von ChaTEAU wird als Eingabe die Inverse Anfrage genommen und als Ergebnis erhält man dann die minimale Teildatenbank. Außerdem werden mithilfe des Provenance-Typs und der SQL-Anfrage Provenance-Berechnungen durchgeführt, dessen Ergebnisse dann mit der minimalen Teildatenbank kombiniert werden. Diese wird dann in der von uns entwickelten Benutzeroberfläche von ProSA ausgegeben.

2. Grundlagen

Anja Wolpers

Zunächst führen wir in die Grundbegriffe ein, auf denen das Projekt basiert. Das Ziel dieses Projektes ist Provenance-Management. Provenance bezeichnet im Allgemeinen die Dokumentation der Herkunft von Artefakten. In diesem Fall geht es insbesondere um die Herkunft und Bearbeitungshistorie von Datenbank-Einträgen. Was Grundlage für die Auswertung einer Anfrage an eine Datenbank ist, kann so nachvollzogen werden. Die Daten-Provenance unterteilen Herschel, Diestelkämper und Lahmar [4] in drei verschiedene Arten: „How“-, „Why“- und „Where“-Provenance. „Where“-Provenance dokumentiert nur, welche Relationen für die Ausführung einer Anfrage genutzt wurden, „Why“-Provenance fügt noch die Information hinzu, welche Tupel aus welchen Relationen kombiniert wurden und „How“-Provenance gibt auch die konkreten Datenbankeinträge an und wie diese in der Anfrage miteinander kombiniert wurden.

ProSA ist ein Konzept von Auge und Heuer [3], das die Provenance-Arten mit minimalen Teildatenbanken kombiniert. Eine minimale Teildatenbank ist der Teil der Datenbank, der für die Bearbeitung einer bestimmten Anfrage benötigt wird. Um diese zu erhalten, haben Auge und Heuer einen Ablauf beschrieben, bei dem die Anfrage invertiert als s-t tgd an ChaTEAU gesendet wird. Eine s-t tdg kann in diesem Kontext in etwa als andere Schreibweise einer Anfrage an eine Datenbank verstanden werden. Sie beschreibt, wie ein (oder mehrere) Quellschema in ein Zielschema überführt wird. Im folgenden Beispiel ist eine Projektion auf die Attribute a und c zu sehen:

$$R(a, b, c) \rightarrow T(a, c)$$

Dabei steht auf der Linken Seite der s-t tgd das Quellschema. Dieser Bestandteil wird auch Body der s-t tgd genannt. Rechts vom Pfeil ist das Zielschema zu finden, das auch Head der s-t tgd genannt wird. Im Head einer gültigen s-t tgd können neben Datenbankschemata auch Quantoren stehen - im Gegensatz zum Body.

ChaTEAU ist auch ein Tool von Auge und Heuer, in dem sie ihren Chase-Algorithmus implementieren. Der Chase-Algorithmus integriert Abhängigkeiten in eine Instanz. Bei der Anwendung innerhalb von ProSA werden Anfragen in Form von s-t tgds in Datenbankschemata integriert. Auch, wenn der Chase nicht garantiert terminiert, terminiert er für s-t tgds immer und dadurch auch immer, wenn er im Kontext von ProSA ausgeführt wird.

3. Gesamtablauf von ProSA

Anne-Sophie Waterstradt

Wie in Abbildung 1 schematisch dargestellt, erhält die von uns entwickelte ProSA-Benutzeroberfläche als Eingabe eine SQL-Anfrage und eine Datenbank-Instanz, sowie einen Provenance-Typ. Dieser kann „Where“, „Why“ oder „How“ sein. Die SQL-Anfrage und die Datenbank-Instanz werden dann an den SQL-Parser weiter gegeben. Der SQL-Parser wurde von KSWS Gruppe 2 parallel zu uns entwickelt und implementiert. Der SQL-Parser erzeugt dann aus der SQL-Anfrage und der Datenbank-Instanz eine XML-Datei, welche die Anfrage als s-t tgd enthält. Diese XML-Datei wird dann an ChaTEAU übergeben.

In ChaTEAU wird der Chase-Schritt ausgeführt und liefert als Ausgabe das Anfrageergebnis. Währenddessen wird in ProSA im Invertierer die ursprüngliche SQL-Anfrage invertiert. Die invertierte Anfrage wird als s-t tgd zusammen mit dem Anfrageergebnis in einer XML-Datei an den Backchase-Schritt übergeben. Das Ergebnis des Backchase-Schritts ist dann die minimale Teil-Datenbank. Die minimale Teil-Datenbank wird dann als XML-Datei zurück an ProSA in den Kombinierer übergeben. Parallel zu den Chase-Schritten wurde im Provenancer das Provenance-Result berechnet. Der Provenancer erhält dafür als Eingabe den Provenance-Typ und die ursprüngliche SQL-Anfrage. Die Ausgabe des Provenancers ist dann eine Tupelliste wenn es „Where“-Provenance war, eine Zeugenbasis wenn es „Why“-Provenance war und ein Polynom wenn es „How“-Provenance war. Dieses Ergebnis wird dann als eine CSV-Datei weitergegeben.

Diese Ausgabe des Provenancers wird dann zusammen mit der minimalen Teil-Datenbank als XML-Datei, welche wie eben erläutert Resultat des Backchase-Schritts ist, an den Kombinierer übergeben. Die Ausgabe des Kombinierers und damit auch von ProSA ist dann die minimale Teil-Datenbank erweitert um Provenance.

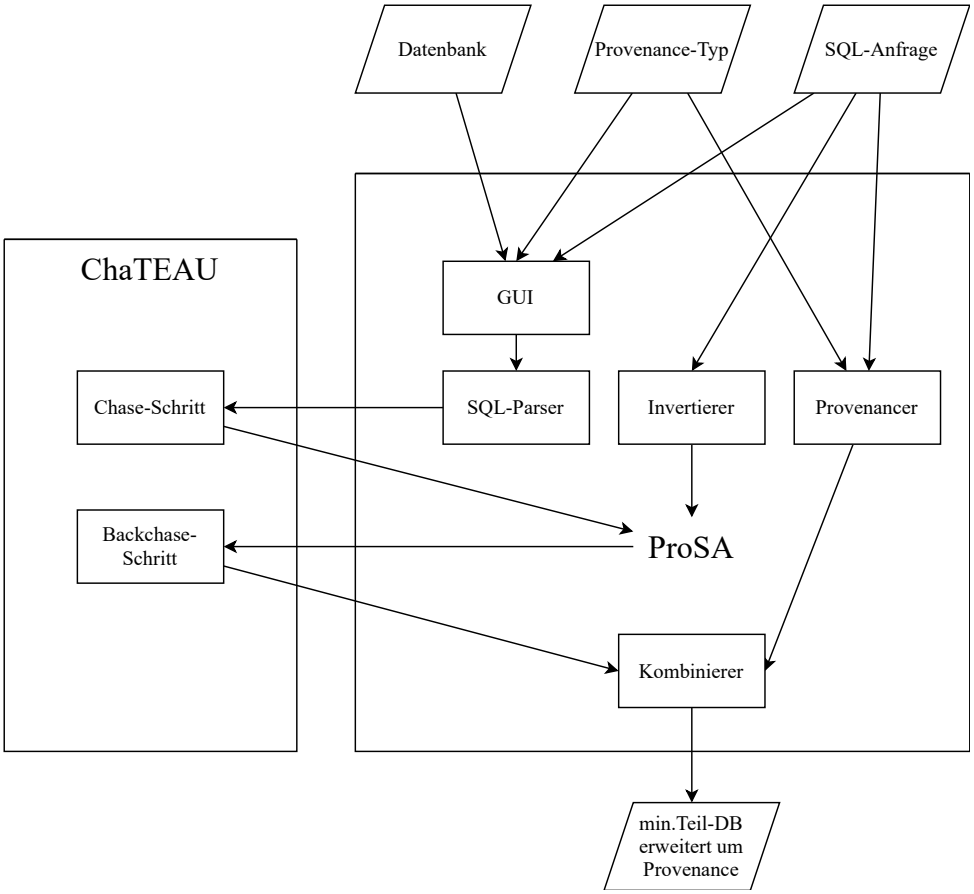


Abbildung 1: Gesamtablauf

4. Aufgabenstellung

Anne-Sophie Waterstradt

Unsere erste Aufgabe war es, uns in das Thema Data Provenance allgemein einzuarbeiten. Außerdem mussten wir die Tools ProSA und ChaTEAU verstehen, um später ein sinnvolles Konzept für die grafische Benutzeroberfläche von ProSA erstellen zu können. Des Weiteren mussten wir den bereits vorhandenen Code von ProSA verstehen um mit diesen weiterentwickeln zu können.

Hauptaufgabe des Projektes war die Entwicklung und Implementierung des GUIs für ProSA. Die GUI haben wir vollständig umgesetzt. Alle Elemente lassen sich funktionsfähig nutzen, nur die Funktion des Buttons „Run ProSA“ ist noch nicht vollständig implementiert. Er startet bereits den Chase-Schritt, eine Einbindung des SQL-Parsers fehlt aber noch. Dieser wird von der KSWS Gruppe 2 parallel zu uns entwickelt und implementiert. Die Einbindung des SQL-Parsers in unser Projekt haben wir vorbereitet. Außerdem haben wir Konzepte für den Invertierer entwickelt und den Backchase-Schritt vorbereitet. Provenancer und Kombinierer müssen zukünftig noch umgesetzt werden.

Das Tool ChaTEAU ist bereits implementiert und gehört daher nicht zu unserer Projektaufgabe. Aus ProSA soll ChaTEAU lediglich aufgerufen werden. Die Anbindung von ChaTEAU über eine entsprechende Schnittstelle haben wir bereits umgesetzt. Weiterhin haben wir die Einbindung und Anzeige der Datenbank in dem GUI, das Logging für das GUI und das Exportieren von Ergebnissen implementiert.

5. Graphical User Interface

Melinda Heuser

Im folgenden Abschnitt wird der Prototyp des GUIs unseres Projektes erläutert, sowie das fertige GUI, welches auf Basis dieses Prototypen erstellt wurde.

5.1. Prototyp

Für das GUI für ProSA haben wir uns, der Übersichtlichkeit wegen, für ein Fenster mit vier Tabs entschieden. Beim Design haben wir uns auch an dem GUI von ChaTEAU orientiert.

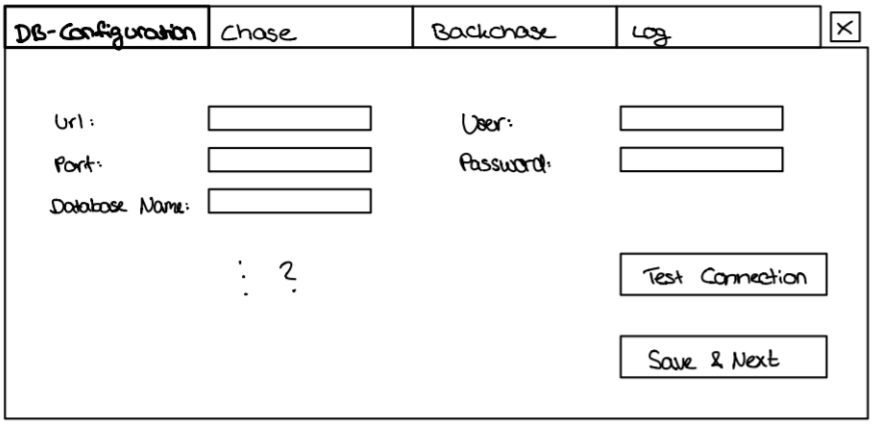


Abbildung 2: DB-Konfiguration

Der erste Tab, der beim Aufrufen der Benutzeroberfläche zu sehen ist, ist ein Fenster mit den allgemeinen Datenbankkonfigurationen (siehe Abbildung 2). Mit dem „Test Connection“-Button lässt sich die Konfiguration überprüfen. Die Konfigurationen werden mit einem Klick auf „Save & Next“ gespeichert und man kommt auf den zweiten Tab.

Im zweiten Tab (siehe Abbildung 3) findet sich der Chase. Hat man bereits eine Datenbank konfiguriert, ist dieser Tab die Startseite für den Nutzer. Hier wird auf der linken Seite des Tabs die originale Datenbank durch das Programm mittels der eingegebenen Daten des Konfiguration-Tabs generiert sowie die SQL-Query eingegeben und der Povenance-Typ ausgewählt. Über den Button „Run ProSA“ wird der Chase ausgeführt und anschließend das Ergebnis auf der rechten

Seite des Tabs im Feld „Result“ angezeigt. Mit einem Klick auf den Pfeil oben Rechts im Tab lassen sich die SQL-Query und die minimale Teil-Datenbank abspeichern. Über den Button „Next Step“ wird zum Backchase-Tab gewechselt.

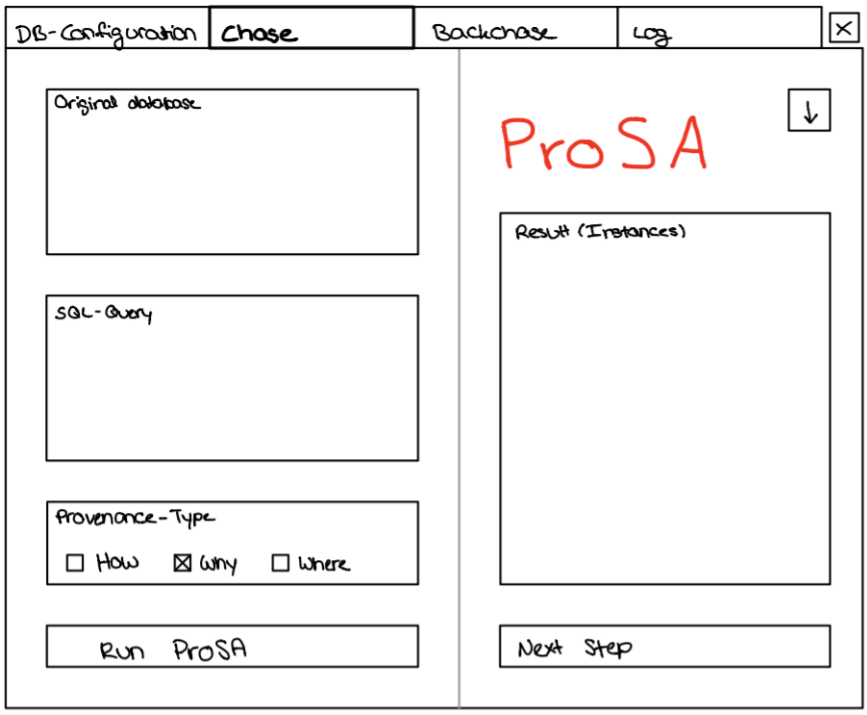


Abbildung 3: Chase

Im Backchase-Tab (siehe Abbildung 4) befinden sich ebenfalls auf der linken Seite die Eingaben und auf der rechten Seite die Ausgaben. Die Eingaben werden durch das Ausführen von „Run ProSA“ im vorherigen Tab generiert und mit Hilfe dieser werden das Provenanceergebnis sowie die minimale Teil-Datenbank berechnet. Über den Pfeil in der oberen rechten Ecke des Tabs lassen sich, wie im vorherigen Tab, die SQL-Query und die minimale Teil-Datenbank abspeichern. Dies wird in einer einzelnen Datei abgespeichert. Über „Previous Step“ gelangt man zurück zum Chase-Tab, über „Show Log“ gelangt man zum Log.

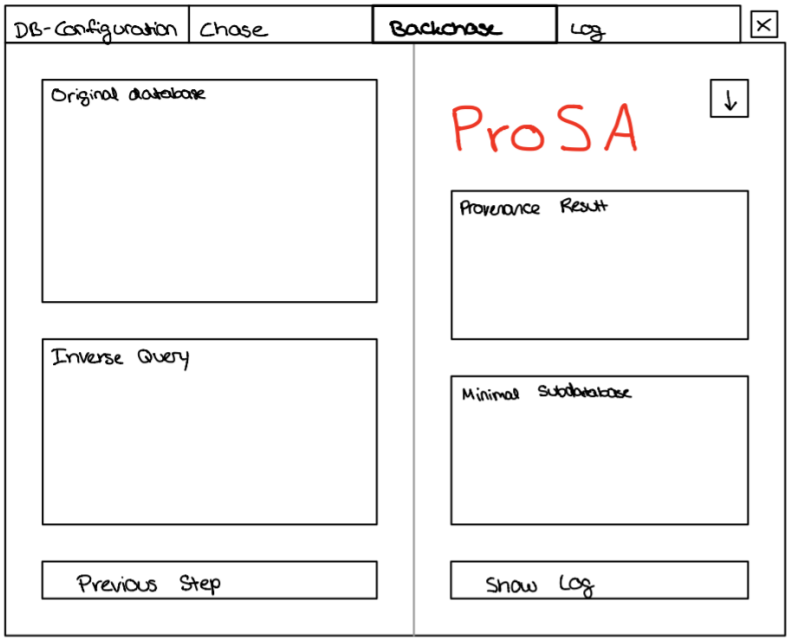


Abbildung 4: Backchase

Der letzte Tab ist der Log-Tab (siehe Abbildung 5). Hier wird lediglich das Log ausgegeben. Es kann über den Pfeil unten mittig abgespeichert werden. Über „Back to Chase“ gelangt man zurück zum Chase-Tab. Der Übersichtlichkeit halber haben wir uns beim Log für einen eigenen Tab entschieden.

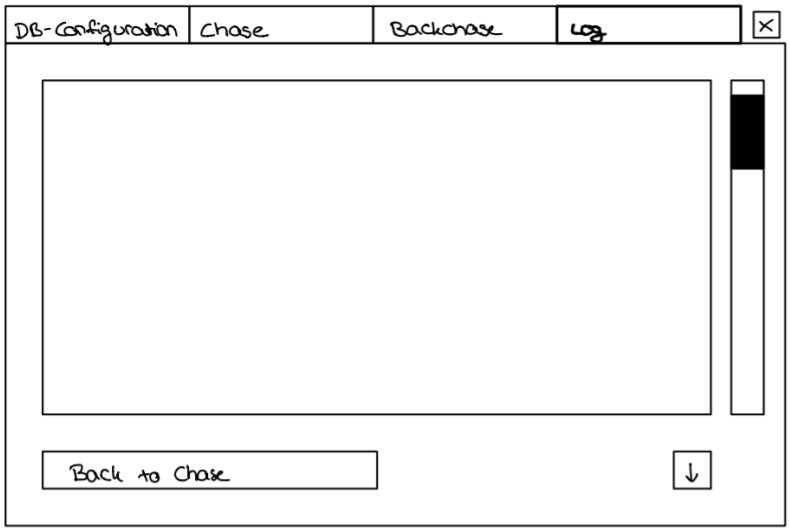


Abbildung 5: Log

5.2. Umsetzung des User Interfaces

Die Umsetzung des User Interfaces erfolgte mittels des Tools SceneBuilder. Das Tool gibt den Code für die erstellten Elemente und Methoden (siehe Abschnitt 8.1 - MainGuiController) sowie die FXML-Datei (eine Datei, in der unser GUI-Design mit allen Elementen und Informationen dazu gespeichert ist, aus der JavaFX beim Start diese Informationen laden kann) aus. Auf Basis des generierten Codes und der Methoden wurde die Implementation aufgebaut.

Der grundlegende Aufbau des User Interfaces entspricht dem des Prototypen: es gibt vier Tabs. Einen für die Konfiguration, einen für den Chase, einen für den Backchase sowie einen für das Log. Die Inhalte der Tabs wurden teilweise überarbeitet. Inwiefern sie überarbeitet wurden und was die Gründe dafür waren, wird in diesem Abschnitt erläutert.

Beim Konfiguration-Tab wurde zunächst ein einleitender Text ergänzt, der das Grundprinzip von ProSA erläutert. Der Übersichtlichkeit halber wurden die Felder zum Angeben der Daten, die zum Aufsetzen der Datenbankverbindung notwendig sind, untereinander angeordnet. Mit einem Klick auf den Knopf „Test Connection“ wird die Verbindung überprüft. Hier erscheint ein Feedback in Form eines kurzen Satzes, ob die Verbindung erfolgreich war oder ob sie nicht aufgebaut werden konnte.

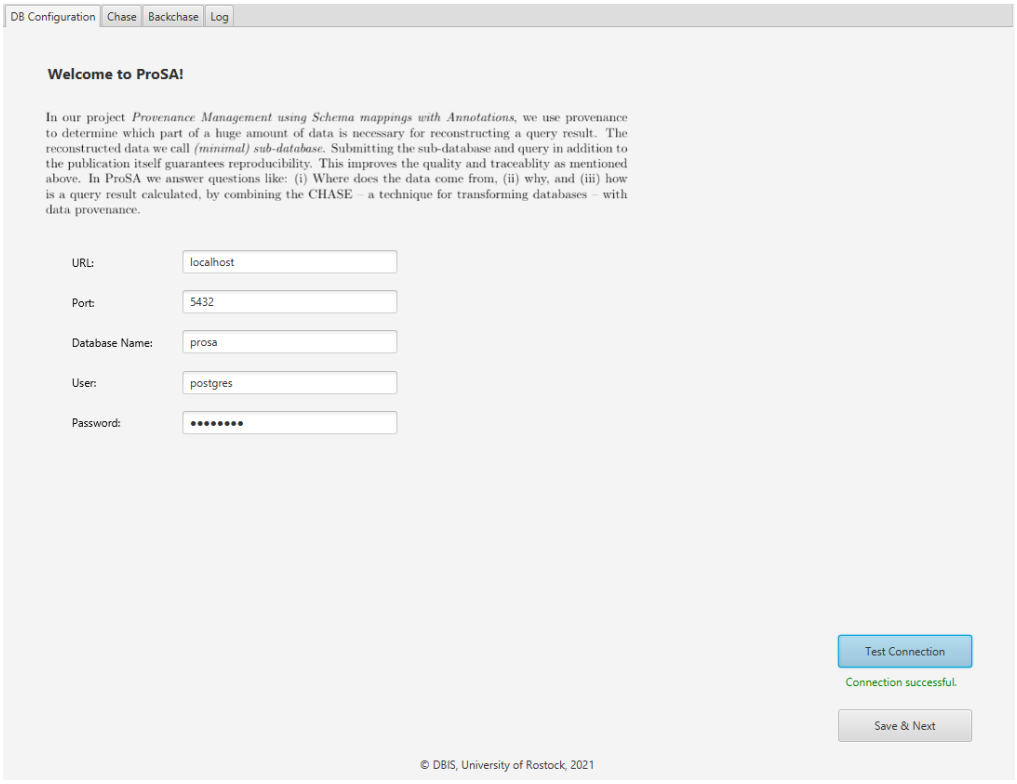


Abbildung 6: Konfiguration

Auch im Chase-Tab wurden Felder umsortiert, um die Übersichtlichkeit zu verbessern: das Feld für die SQL-Query und das der originalen Datenbank wurden vertauscht. So ist das große Feld in dem die Datenbank zu sehen ist unten und die gesamte Eingabe erscheint übersichtlicher, auch da das Feld für die Datenbank auf gleicher Höhe mit dem Feld des Ergebnisses beginnt. Des Weiteren wurde eine kurze Beschreibung des Chase-Schrittes ergänzt.

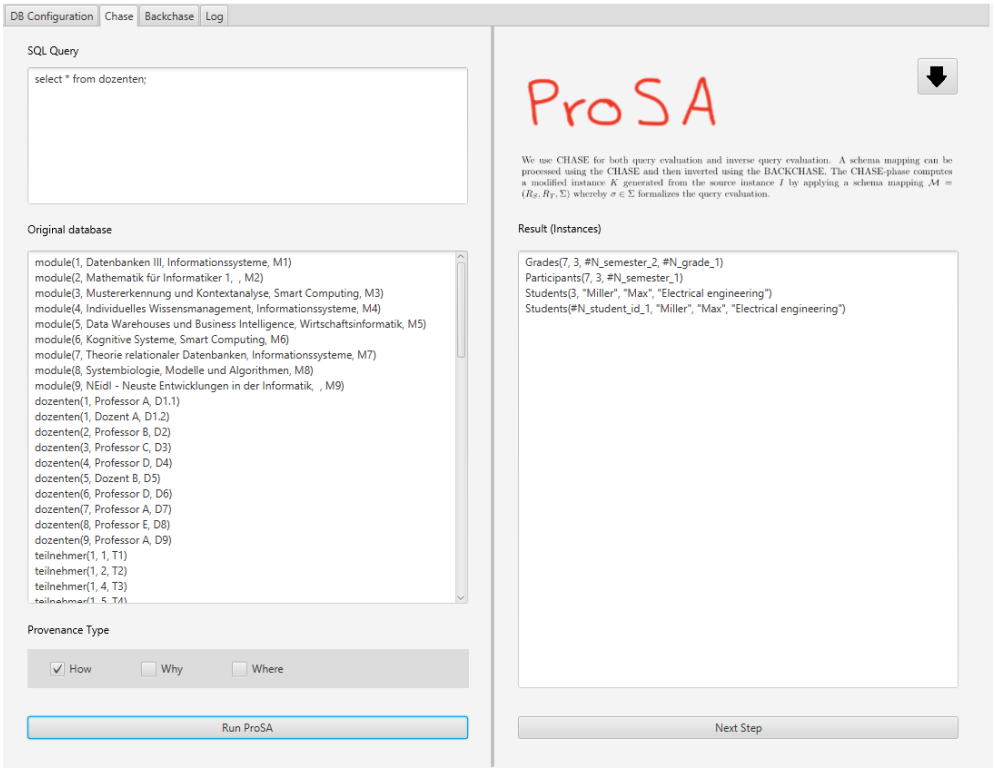


Abbildung 7: Chase (Anmerkung: Query und Ergebnis passen hier nicht zusammen, da der Parser noch nicht funktioniert. Sie sollen hier nur als Beispiel dienen.)

Beim Backchase-Tab wurden analog zum Chase-Tab die Felder für die originale Datenbank und das für die inverse Anfrage vertauscht, ebenfalls der Übersichtlichkeit und Einheitlichkeit wegen. Außerdem gibt es auch in diesem Tab nun eine kurze Erklärung des Schrittes, der auf diesem Tab durchgeführt wird.

Beim Log-Tab wurde zusätzlich zum allgemeinen ProSA-Log noch eines für Chateau hinzugefügt, um die beiden Ausführung des Chase genauer protokolliert zu haben und getrennt vom Gesamtablauf betrachten zu können.

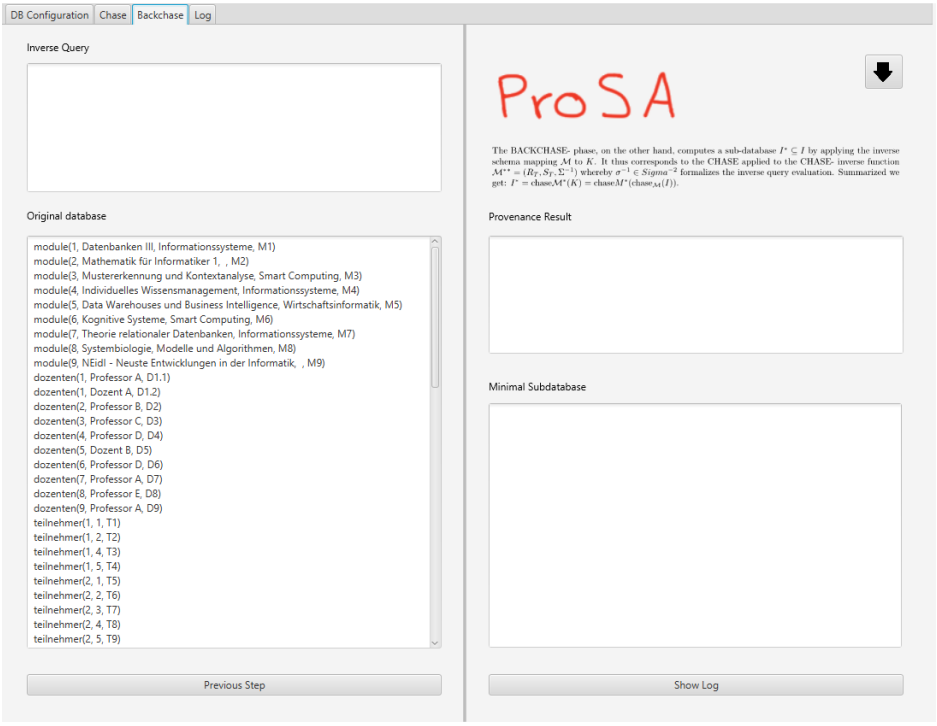


Abbildung 8: Backchase

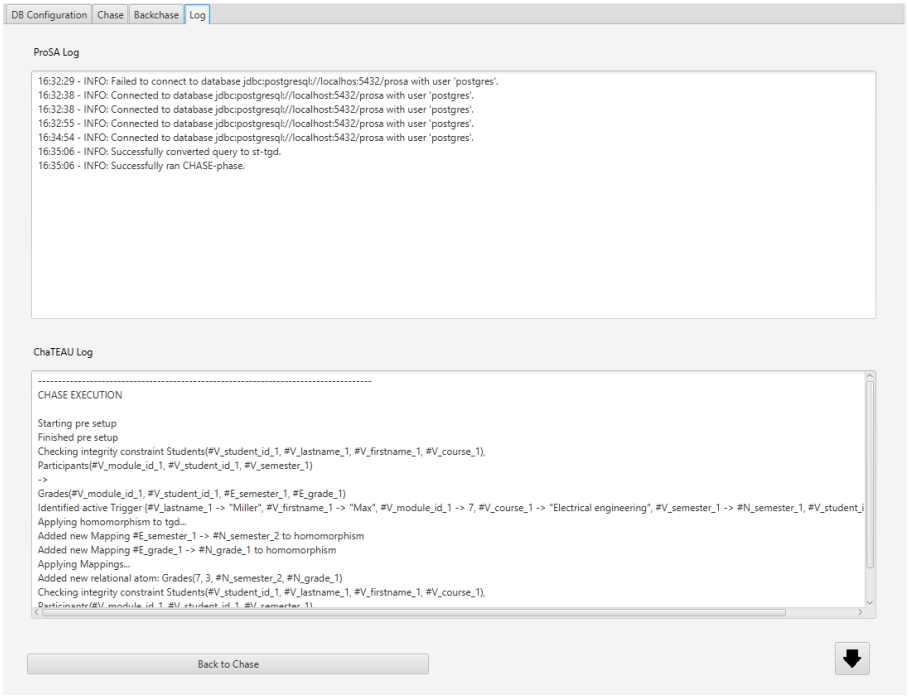


Abbildung 9: Log

6. User Manual

Anja Wolpers

Im folgenden Abschnitt erklären wir kurz, was bei der Installation unserer Anwendung zu beachten ist, wie die in der Anwendung enthaltene Beispieldatenbank einzurichten ist und wie die Anwendung genutzt werden kann.

6.1. Installation

Unser Java-Projekt verwaltet seine Abhängigkeiten mit Maven. Wir nutzen „postgresql“ für die Verwaltung der Datenbankverbindung und „openjfx“ als Graphische Oberfläche. „json-simple“ nutzen wir, um Konfigurationsdaten zu speichern. Für den Chase- und Bachchase-Schritt benötigen wir eine Abhängigkeit zu ChaTEAU. Da ChaTEAU ein weiteres Projekt vom Datenbank-Lehrstuhl ist, muss dies jedoch im Gegensatz zu den zuvor genannten Abhängigkeiten noch durch Maven installiert werden. Dazu muss das Projekt von `https://git.informatik.uni-rostock.de/ta093/chateau-ksws` zunächst geladen werden. Die Installation kann auf drei verschiedene Wege erfolgen:

Installation mit IntelliJ

In dem mit IntelliJ geöffneten ChaTEAU-Projekt (mit Projekt-SDK Java 15) ist auf der rechten Seite ein Maven-Menü, in dem unter „Lifecycle“ das Projekt durch einen Doppelklick auf „installieren“ installiert werden kann.

Installation in der Kommando-Zeile

Um in der Kommandozeile installieren zu können, muss zunächst Maven für das jeweilige Betriebssystem installiert sein. Zusätzlich muss in Path JAVA_HOME auf eine Java 15 Installation gezeigt werden. Daraufhin kann durch den Aufruf von „mvn clean install -U“ in dem Ordner des Projektes das Projekt installiert werden.

Installation mit Installations-Skript

In unserem Projekt liegt unter resources/installer jeweils ein Skript zum Herunterladen und Installieren von ChaTEAU. Um dieses Skript ausführen zu können, muss jedoch auch Maven installiert sein und in Path JAVA_HOME auf eine Java 15 Installation zeigen.

Um das Projekt schließlich ausführen zu können, muss die Projekt-SDK Java 15 sein.

Aufsetzen der Beispieldatenbank

Wenn die Beispiel-Datenbank genutzt werden soll, muss diese erst aufgesetzt werden. Dazu wird postgresql benötigt. Dies kann aus dem Internet geladen und installiert werden. Dort sollte eine Datenbank mit dem Namen „prosa“ und den Zugangsdaten aus dem folgenden Abschnitt „DB Configuration“ angelegt werden. Diese Datenbank kann dann beispielsweise in IntelliJ in das Projekt geladen und über „database-setup.sql“ aus dem Projekt mit Einträgen gefüllt werden.

6.2. Anleitung zur Benutzung des Progammes

Navigation

Durch Buttons unten im Programm-Fenster kann die jeweils nächste oder vorherige Seite im Programm-Ablauf erreicht werden. Nur die Seite „DB Configuration“ kann nicht durch einen Zurück-Button auf der Folgeseite sondern nur durch die Tabs oben im Programm-Fenster erreicht werden. Die anderen Seiten können auch über die Tabs erreicht werden.

DB Configuration

Bei der ersten Nutzung des Programmes oder zur Änderung der Datenbank-Verbindung sollten die Textfelder auf dem Reiter „DB Configuration“ ausgefüllt werden. Die benötigten Zugangsdaten werden beim Aufsetzen der Datenbank vergeben. Die Test-Datenbank kann zum Beispiel mit folgenden Zugangsdaten aufgesetzt werden:

```
dbms=postgresql
url=localhost
port=5432
user=<postgesql user>
password=<postgresql user password>
dbname=prosa
```

ProSA ausführen

In dem Chase-Tab kann im Feld „SQL Query“ eine SQL-Anfrage eingegeben werden. Diese sollte keine group-by-Klausel enthalten und auch keine Aggregationen. Diese Funktionen werden von der bisherigen Implementation noch nicht berücksichtigt. Weiter unten kann der Provenance Typ ausgewählt werden, auch wenn diese Funktion der Anwendung auch noch nicht implementiert ist. Schließlich kann mit dem Button „Run ProSA“ die Ausführung von ProSA gestartet werden. Darauf hin werden in den bisher leeren Textfeldern auf dem Chase- und Backchase-Tab die

(Zwischen-) Ergebnisse des ProSA-Durchlaufs angezeigt. Eine genauere Beschreibung der Inhalte der Tabs ist in Abschnitt 5.1 zu finden.

Fehlermeldungen

Bei Eingabe- oder Anwendungsfehlern oder auch Fehlern bei der Ausführung von ChaTEAU werden diese im Log gespeichert und bei Eingabefehlern wird der Nutzer auch außerhalb des Logs in der graphischen Oberfläche auf diese hingewiesen.

Wenn eine Verbindung zur Datenbank mit den Eingaben auf dem „DB Configuration“-Tab nicht möglich ist, wird die Fehlernachricht „Failed to connect to database <dbUrl> with user '<user>'.“ in das Log geschrieben. Zusätzlich wird der Nutzer auch in dem Chase und Backchase-Tab im „Original Database“-Textfeld darauf hingewiesen, dass er die Datenbankkonfiguration überprüfen soll.

Die andere Stelle, an der Eingabefehler aufkommen können, ist die Eingabe der SQL-Anfrage. Wird keine Anfrage eingegeben, wird dies in das Log und in das „Result“-Textfeld auf dem Chase-Tab geschrieben. Bei einer syntaktisch nicht korrekten Anfrage wird dies auf den selben Wegen mitgeteilt, allerdings wird diese Art von Fehler in der Methode ausgelöst, die durch einen neuen Parser ersetzt werden soll. Somit kann sich die Ausprägung dieser Fehlermeldung mit dem neuen Parser ändern.

7. Einschränkungen

Anja Wolpers

Da die von uns implementierte Anwendung nur zur Veranschaulichung des Konzeptes dient und zum Teil noch abhängig von laufender Forschung ist, müssen vom Nutzer bestimmte Einschränkungen in Kauf genommen werden.

Es können nur verhältnismäßig kleine Datenbanken in die Anwendung geladen werden. Dies liegt daran, dass sowohl in ProSA als auch in der Bibliothek, die für die Umwandlung von Anfragen in s-t-tgds genutzt werden soll, die Datenbank zeitweise komplett im Hauptspeicher gehalten wird. Dem ist auch geschuldet, dass die Datenbank in der graphischen Oberfläche auch nur in einem Textfeld dargestellt wird, in dem eine größere Datenbank unübersichtlich würde. Diese Einschränkung ist dadurch gerechtfertigt, dass diese Anwendung nur dazu dient, das Konzept in seinen Teilschritten zu veranschaulichen. Für diese Zielsetzung sind große Datenbanken schon alleine nicht geeignet, weil dann das Konzept schon alleine durch die Unübersichtlichkeit der Datenbank nicht nachvollzogen werden kann.

Die Anwendung unterstützt derzeit nur postgresql Datenbanken. Auch diese Einschränkung lässt sich damit erklären, dass die Anwendung nur das Konzept veranschaulichen soll und somit keine

Kompatibilität mit mehr Datenbanksystemen im Rahmen unseres Projektes nötig ist. Zudem können entsprechend kleine Datenbanken, für die die Anwendung ausgelegt ist, recht einfach in postgressql umgezogen werden.

Wie schon in Abschnitt 6.2 erwähnt, können nur SQL Anfragen ohne group-by-Klauseln und ohne Aggregationen bearbeitet werden. Diese Einschränkung entsteht dadurch, dass die Bibliothek, die die Anfrage in eine s-t tgd umwandelt, diese Bestandteile nicht unterstützt. Zusätzlich ist auch die Forschung zur Invertierung von Anfragen bzw. s-t tdgs noch unvollständig im Bereich der Gruppierungen. Dies wäre für die Provenance-Informationen zu der Anfrage jedoch nötig.

Zudem ist die Komponente, die die Umwandlung der SQL-Anfragen in s-t tgds übernehmen wird, zum Zeitpunkt des Programmierstopps noch nicht fertig gestellt. So wird derzeit eine fest in das Programm geschriebene s-t tgd an ChaTEAU gesendet – unabhängig von der SQL-Anfrage des Nutzers. Dementsprechend ist das derzeit angezeigte Anfrage-Ergebnis losgelöst von der Beispiel-Datenbank und eingegeben Anfrage zu sehen, bis ein zur aktuellen Version von ChaTEAU passender Parser fertiggestellt und als Abhängigkeit in diese Anwendung eingebunden ist.

8. Implementation

Judith-Henrike Overath

Im Folgenden soll eine Übersicht über unsere Implementation gegeben werden, sodass diese verstanden, angepasst und erweitert werden kann sowie eingeführte Konventionen weiterhin genutzt werden können. Die umgesetzten Klassen sind in der Abbildung 10 dargestellt. Ihre Funktionalitäten werden nun näher erläutert.

8.1. Controllerstruktur

Im *main/java*-Ordner des Projektes sind mehrere packages zu finden. Wir haben das GUI inklusive Funktionalitäten im package *controller* umgesetzt. Abseits davon haben wir nur die `main`-Methode angepasst, sodass diese die `main`-Methode des `GuiLaunchers` aufruft. Dieses Auslagern der `main`-Methode war zwingend notwendig, da es anderweitig Fehlermeldungen gab. Nach unserer Recherche war dies ein bekannter Bug von JavaFX, so wie wir es mit dem aus SceneBuilder generiertem Code genutzt haben.

Für jeden Funktionalitätsbereich haben wir einen eigenen Controller erstellt. So ist der `LogController` beispielsweise für das Logging verantwortlich. Alle Controller mit GUI-Funktionalitäten sind im package *controller.guiController* zu finden.

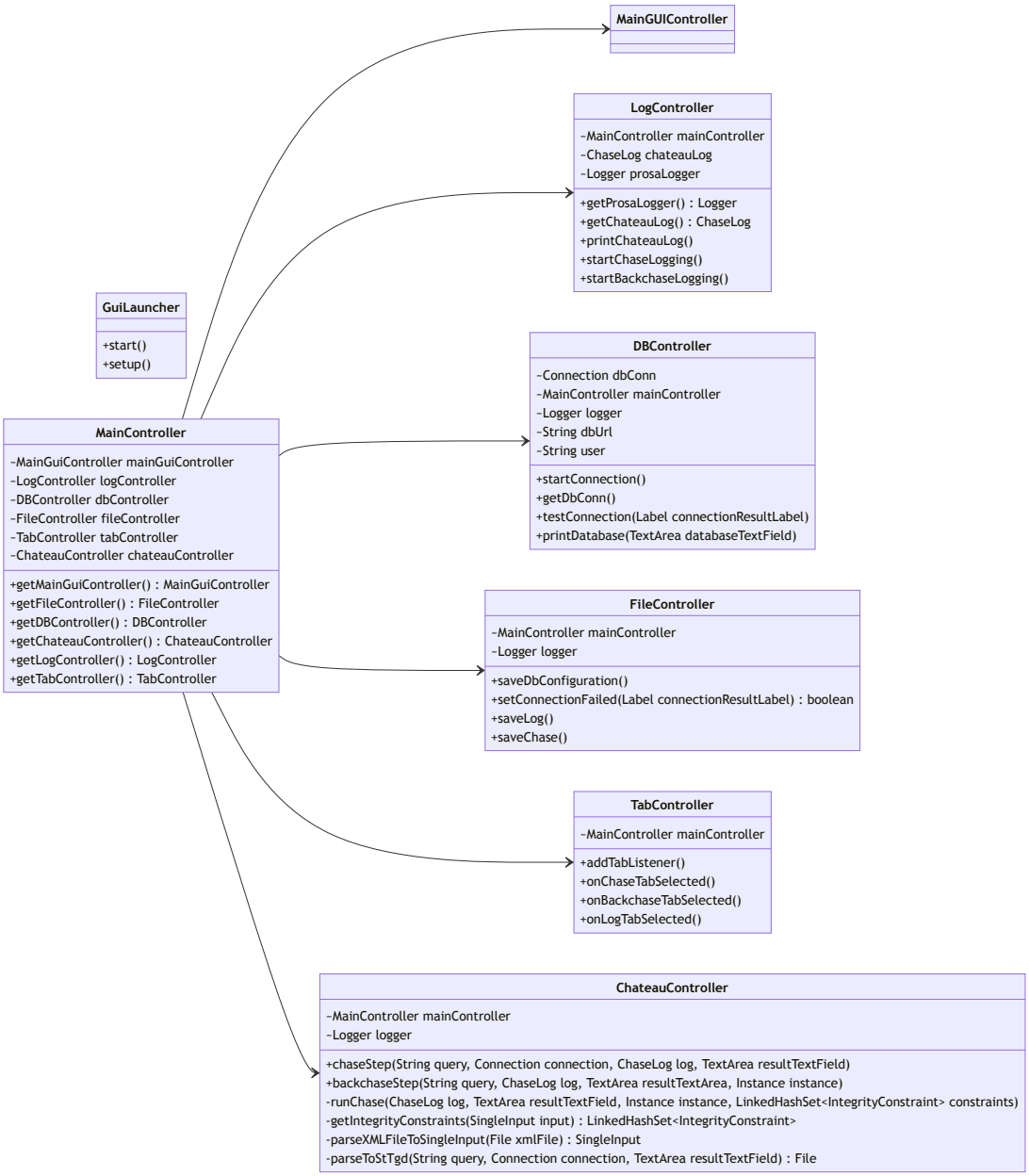


Abbildung 10: Klassendiagramm unserer Controllerstruktur

GuiLauncher

Die Klasse `GuiLauncher` repräsentiert unsere JavaFX-Application. Sie erbt von der Klasse `javafx.application.Application` und überschreibt dessen Methode `start`. In dieser wird der Aufbau unseres GUIs aus der prosaGUI.fxml geladen, `setup` aufgerufen und das GUI gestartet. In `setup` haben wir zusätzliche Funktionalitäten eingebaut, die beim Starten des GUIs notwendig sind. Hier wird ein `mainGuiController` und ein `mainController` erzeugt sowie die Datenbankkonfiguration geladen (`FileController.loadDBConfiguration`). Falls diese existiert wird

eine Verbindung zur Datenbank aufgebaut und der Chase-Tab gestartet (`DBController.startConnection` und `MainGUIController.startChaseTab`). Außerdem werden die Tab-Listener initialisiert (`TabController.addTabListener`).

MainController

Der `MainController` hat die Aufgabe, alle anderen Controller miteinander zu verbinden. In seinem Konstruktor wird eine Instanz aller anderen Controller als Variablen des `MainController`s gespeichert. Alle anderen Controller können über den `MainController` mit `get`-Methoden aufeinander zugreifen.

MainGuiController

Alle Objekte des GUIs, auf die wir im Code Zugriff benötigen, wie beispielsweise Tabs, Buttons und TextFields, sind in der Klasse `MainGuiController` als private Attribute gespeichert. Außerdem sind hier `handle`-Methoden für Buttons implementiert, welche aufgerufen werden, wenn der entsprechende Button gedrückt wird. Beispielsweise wird in `handleSave`, wenn der Nutzer im Datenbankkonfiguration-Tab auf „Save“ drückt die Konfiguration gespeichert (`FileController.saveDbConfiguration`), eine neue Verbindung zur Datenbank aufgebaut (`DBController.startConnection`) und der Chase-Tab gestartet (`startChaseTab`). Mit `handleRun` sollten alle Hauptaufgaben des Programms durchgeführt werden. Aktuell wird hier nur der Chase aufgerufen (`ChateauController.runChase`). Die weiteren Schritte müssen noch eingefügt werden.

Des Weiteren haben wir in diesem Controller nach Bedarf einige `get`-Methoden implementiert, sodass auch in den anderen Controllern ein Zugriff auf TextFields, Tabs, TextAreas, etc. möglich ist. Außerdem haben wir einzelne Methoden eingeführt, die duplizierten Code vermeiden sollen, wie etwa `startChaseTab`, in welcher der Chase-Tab ausgewählt und die Methode `onChaseTabSelected` des `TabController`s aufgerufen wird.

LogController

Der `LogController` liefert Funktionalitäten zum Logging und speichert das `chateauLog` und den `prosaLogger` als Klassenattribute. Andere Klassen können die Logs mithilfe von `get`-Methoden erreichen und speichern diese oft in ihrem Konstruktor in eigene Attribute ab. Das `chateauLog` ist entsprechend der Konvention von ChaTEAU ein Objekt der Klasse `ChaseLog` und dient dem Logging der beiden Chase-Ausführungen, wobei beide Logs in diesem Objekt gespeichert werden. Die Ausführungen können mit `startChaseLogging` und `startBackchaseLogging` getrennt werden. In diesen Methoden wird dem `chateauLog` ein Header für die jeweilige Phase hinzugefügt. Mit Hilfe der Methode `printChateauLog` kann dann das `chateauLog` in dem GUI ausgegeben

werden. Dem Objekt kann mithilfe der `ChaseLog`-Methode `addEntry` Einträge hinzugefügt werden.

Für unser eigenes ProSA-Log haben wir entsprechend Java-Konventionen die Klasse `Logger` genutzt. Ihr können mithilfe von `log` neue Einträge hinzugefügt werden. Wir hielten die Klasse für besser geeignet, da man dort mit Log-Level arbeiten kann und man ihr eigene Handler hinzufügen kann. Wir haben im Konstruktor des `LogController`s dem `prosaLogger` einen Handler hinzugefügt, der, wenn ein neuer Eintrag geloggt wird, dies automatisch mit der Uhrzeit, dem Log-Level und der Nachricht in dem GUI ausgibt. So muss nicht wie für das `chateauLog` beim Aufrufen des Log-Tabs das Log ausgegeben werden, da dies automatisch im Hintergrund geschieht.

Für das Log-Level im `prosaLogger` haben wir eigene Konventionen eingeführt:

- `INFO` – Information für den Nutzer (kein Fehler)
- `WARNING` – erwarteter Fehler, ausgelöst durch den Nutzer (beispielsweise ungültige Datenbankkonfiguration)
- `SEVERE` – unerwarteter Programmfehler, der nicht auftreten sollte

DBController

Die Aufgabe des `DBController`s ist es, die Verbindung zur Datenbank zu verwalten. Dazu besitzt er ein Attribut `dbConn`, in welcher die `Connection` in der Methode `startConnection` gespeichert wird. Dazu wird die Datenbankkonfiguration aus der gespeicherten JSON-Datei eingelesen und eine entsprechende Verbindung aufgebaut. Andere Controller können auf die `dbConn` mittels einer `get`-Methode zugreifen. Falls eine Verbindung nicht erfolgreich aufgebaut werden konnte, ist die Methode `setConnectionFailed` hilfreich, um dies in einem beliebigen Label des GUIs auszugeben. Im `DBController` ist in der Methode `testConnection` auch die Funktionalität des Testens der Datenbankkonfiguration umgesetzt. Es wird versucht, eine Verbindung zur Datenbank aufzubauen und das Ergebnis im `connectionResultLabel` ausgegeben. Weiterhin ist hier `printDatabase` zu finden. Diese Methode wird genutzt, um in dem GUI die Datenbank beim Chase- und Backchase-Tab anzuzeigen. Falls keine Verbindung aufgebaut werden konnte, wird hier der Hinweis „Failed to connect to database. Please check your DB Configuration.“ ausgegeben.

FileController

Der `FileController` ist für das Speichern und Laden von Dateien zuständig. Dies betrifft sowohl die interne Datenbankkonfiguration als auch exportierbare Dateien. `saveDbConfiguration` kann aufgerufen werden, um die in dem GUI eingegebene Konfiguration in die Konfigurationsdatei im JSON-Format zu speichern. Entsprechend kann mit `loadDbConfiguration` die Konfiguration

aus der Datei wieder eingelesen und in dem GUI angezeigt werden. `saveLog` führt das Speichern des ProSA- und ChaTEAU-Logs in eine Datei aus. Mit `saveChase` wird die SQL-Anfrage und minimale Teildatenbank aus dem GUI abgespeichert. Bei beiden Methoden haben wir die Klasse `FileChooser` genutzt, um dem Nutzer zu ermöglichen, selbst einen Speicherort und Namen für die Datei zu wählen.

TabController

Wenn ein Tab aufgerufen wird, müssen eventuell Inhalte dafür geladen oder gespeichert werden. Dies ist im `TabController` umgesetzt. So wird in `onChaseTabSelected` die Datenbank im Chase-Tab ausgegeben (`DBController.printDatabase`). Gleiches wird in `onBackchaseTabSelected` für den Backchase-Tab umgesetzt. Bei Auswahl des Log-Tabs kann mit `onLogTabSelected` das ChaTEAU-Log ausgegeben (`LogController.printChateauLog`) werden. Diese Methoden werden sowohl außerhalb der Klasse als auch in `addTabListener` genutzt. Außerhalb der Klasse können sie hilfreich sein, um nicht genau wissen zu müssen, welche Aspekte für welchen Tab beachtet werden müssen. Beim Start des GUIs wird `addTabListener` aufgerufen und fügt einen neuen Listener hinzu, der gewünschte Funktionalitäten beim Wechseln des Tabs über die Tab-Leiste oben automatisch umsetzt. War der alte Tab die Datenbankkonfiguration, so wird diese gespeichert (`FileController.saveDbConfiguration`) und die Verbindung neu aufgebaut (`DBController.startConnection`). Für neue Tabs wird gegebenenfalls eine der drei anderen genannten Methoden des `TabController`s aufgerufen.

ChateauController

Die Chase-Ausführungen werden im `ChateauController` in `chaseStep` und `backchaseStep` durchgeführt. Im Chase-Step wird zunächst die Query in eine s-t tgd im XML-Format mit `parseToStTgd` geparsed. Da die aktuelle Implementierung keine korrekte s-t tgd für die aktuelle ChaTEAU-Version liefert, nutzen wir diese s-t tgd momentan nicht sondern haben übergangsweise eine Beispieldatei statisch eingelesen. Die XML-Datei wird dann mithilfe der ChaTEAU-Methode `parseXMLToSingleInput` eingelesen und aus dieser die Instanz und deren Integritätsbedingungen extrahiert. Im Chateau-Log wird ein Header für den Chase-Schritt ausgegeben (`LogController.startChaseLogging`) und der Chase für die Instanz und deren Integritätsbedingungen mit `runChase` gestartet. Ähnlich sollte auch der Backchase-Schritt in `backchaseStep` implementiert werden, der aktuell noch nicht funktionsfähig ist. Hier müsste zunächst die Query invertiert werden und für dessen Instanz und Integritätsbedingungen der Chase erneut aufgerufen werden.

Mit `runChase` soll eine beliebige Chase-Ausführung ermöglicht werden. Hier haben wir uns stark am ChaTEAU-Code orientiert und dessen Klasse `ChaseThread` genutzt. Ein Objekt der Klasse wird mit den gegebenen Informationen erzeugt und der Thread gestartet. Dann wird asynchron

zum restlichen Programmverlauf darauf gewartet, dass der `ChaseThread` terminiert. Wenn dieser fertig ist, wird entsprechend das Ergebnis ausgegeben.

`getIntegrityConstraints` und `parseXMLFileToSingleInput` sind für den beschriebenen Ablauf Hilfsfunktionen, die die Lesbarkeit erhöhen sollen. In `parseToStTgd` ist der Parser eingebunden, hier werden alle möglicherweise auftretenden Exceptions gefangen und entsprechend geloggt.

9. Erweiterungsmöglichkeiten

Anja Wolpers

Der aktuelle Prototyp führt bisher nur den Chase-Schritt des Algorithmus aus. Um auch den Backchase-Schritt in die Anwendung zu integrieren, ist der nächste Schritt, die Anfrage an die Datenbank zu invertieren [1]. Dazu schlagen wir im Abschnitt 9.1 ein Konzept vor und vergleichen es mit anderen Möglichkeiten, die uns bekannt sind. Nach der Invertierung muss die invertierte Anfrage als s-t tgd an ChaTEAU gesendet werden. Das Ergebnis wird dann in dem „Minimal Subdatabase"-Textfeld dargestellt.

Zusätzlich muss noch die minimale Teildatenbank um Provenance-Informationen erweitert und so in dem „Provenance-Result"-Textfeld dargestellt werden.

Schließlich wird noch der Parser eingebunden werden müssen, der die SQL-Anfragen des Nutzers in s-t tgds umwandelt.

9.1. Invertierung von Anfragen

Leonie Förster

Für die Invertierung von Datenbankanfragen im Kontext von ProSA eigenen sich grundsätzlich drei verschiedene Konzepte. Die erste Möglichkeit ist das Invertieren der SQL-Anfrage und die zweite Variante ist das Invertieren der Anfrage als s-t tgd. Die Kombination der beiden Ideen stellt die dritte Möglichkeit dar. Dabei liegt die Anfrage zu Beginn in SQL vor und wird im Programmablauf auf jeden Fall in eine s-t tgd umgewandelt. Ziel ist es am Ende die invertierte Anfrage als s-t tgd vorliegen zu haben. Im Folgenden werden die Vor- und Nachteile der verschiedenen Konzepte zur Invertierung verglichen.

9.1.1. Vergleich der verschiedenen Ansätze

SQL

Die Invertierung mit der SQL-Anfrage durchzuführen ist vermutlich die schwierigste Variante. Hierbei wird die SQL-Anfrage auf Basis von SQL invertiert. Der Vorgang allerdings ist sehr komplex und vermutlich nur schwierig algorithmisch umzusetzen. Zudem müsste die invertierte

SQL-Anfrage zum Schluss wieder in eine s-t tgd umgewandelt werden. Vorteilhaft dabei ist, dass der Parser von Gruppe 2 bereits für diese direkte Umwandlung von SQL in s-t tgd geeignet ist und bei dieser Variante keine Anpassungen am Parser notwendig sind.

s-t tgd

Die Invertierung mit auf Basis von st-tgs ist weniger kompliziert als die Umwandlung über SQL. Dabei wird die s-t tgd auf Basis der entsprechenden Regeln invertiert. Vorteilhaft ist dabei, dass die Anfrage bereits als aus SQL geparste s-t tgd vorliegt. Allerdings müsste für eine effiziente Umwandlung die s-t tgd selbst geparst werden, damit die verschiedenen Bestandteile innerhalb eines Algorithmus verarbeitet werden können. Hierfür ist allerdings bisher keine Implementierung vorhanden. Außerdem ist die Implementierung der Regeln für die Invertierung umständlicher, wenn nicht sicher ist um welche Art von Anfrage es sich ursprünglich gehandelt hat. Das könnte vor Allem bei komplexeren Anfragen zunehmend schwieriger werden.

hybrid

Bei der Kombination der Varianten wird die SQL-Anfrage geparst und dann schrittweise in mehrere s-t tgds umgewandelt, die sich auf die jeweiligen Teilanfragen der ursprünglichen SQL-Anfrage zurückführen lassen. Das genaue Vorgehen wird im Abschnitt 9.1.2 erläutert. Vorteil hierbei ist, d!ss das Parsen der SQL-Anfrage bereits implementiert ist (Beispielimplementation: JSQL). Insgesamt ist diese Variante vermutlich auch am einfachsten als Algorithmus umzusetzen. Nachteil gegenüber der SQL-Invertierung ist allerdings, dass der Parser von Gruppe 2 angepasst beziehungsweise erweitert werden müsste.

Zusammengefasst lässt sich sagen, dass die Invertierung über SQL die komplizierteste ist. Die anderen beiden Varianten wären in ihrer Implementierung vermutlich recht ähnlich, jedoch ist die hybride Möglichkeit die am einfachsten Umzusetzende. Daher wird das Konzept dafür im folgenden Abschnitt genauer beschrieben.

9.1.2. Konzept für Kombination aus SQL- und s-t tgd-Invertierung

Die Beschreibung unseres Konzeptes gliedert sich in zwei Abschnitte, die Theorie und die Praxis. Dabei betrachtet der praktische Teil vor Allem die algorithmische Umsetzung und die möglichen Vereinfachungen, während der theoretische Teil die strikten Grundregeln beschreibt.

Theorie

Zur Veranschaulichung wird für die Beschreibung des Konzeptes von einem Beispiel begleitet.

Eingabe: SQL-Anfrage (ohne Gruppierung oder Aggregationen), Datenbankschema

```
SELECT a,b
FROM R NATURAL JOIN S
WHERE a = 3
```

Im **ersten Schritt** wird die SQL-Anfrage unter Beachtung der Reihenfolge der Operationen geschachtelt. In der Basisinstanz, also ein `SELECT-FROM-WHERE`-Block (kurz SFW-Block), wird die Anfrage so geschachtelt, dass die äußerste Anfrage die Projektion darstellt, die Mittlere die Selektion und die Innerste den Verbund. Sollten Teile davon nicht in der ursprünglichen Anfrage vorkommen, müssen diese auch nicht in der Schachtelung berücksichtigt werden.

Bei mehreren SFW-Blöcken bleibt die Schachtelung der Anfrage erhalten und nur die Basisinstanzen werden verschachtelt.

```
SELECT a,b
FROM (
        SELECT *
        FROM (
                SELECT *
                FROM R NATURAL JOIN S)
        WHERE a = 3)
```

Der **zweite Schritt** wandelt die verschachtelte SQL-Anfrage in eine s-t tgd um. Das geschieht beginnend mit dem innersten Block. Dabei wird der Head der s-t tgd der Body der s-t tgd für den nächsten Block, sodass die umgewandelten s-t tgd Blöcke miteinander verkettet sind.

Wenn die Heads von mehreren Vorgängern identisch sind, dann wird trotzdem nur ein neuen Body für die nächste Anfrage benutzt.

$$R(a,c) \wedge S(b,c) \rightarrow T(a,b,c) \qquad \textit{natürlicher Verbund}$$
$$T(a,b,c) \wedge a = 3 \rightarrow P(3,b,c) \qquad \textit{Selektion}$$
$$P(3,b,c) \rightarrow Q(3,b) \qquad \textit{Projektion}$$

Im **dritten Schritt** werden die zuvor aufgestellten s-t tgds invertiert. Dafür werden die Basisregeln zur Invertierung (siehe Abschnitt 12 im Anhang) genutzt. Begonnen wird dabei mit der letzten s-t tgd und analog zum Schritt davor, wird der neue Head der invertierten s-t tgd zum Body der nächsten s-t tgd. Die Quantoren werden dafür jedoch nicht in den Body übernommen,

um sicherzustellen, dass es sich auch weiterhin um eine s-t tgd handelt. Auch Bedingungen werden dabei nicht übernommen. Eine Ausnahme bilden allerdings Bedingungen der From $a = x$. Diese werden in die folgenden s-t tgds durch Ersetzung übernommen.

Invertierung Projektion

Ausgangsformel: $\underline{P(3, b, c)} \rightarrow Q(3, b)$

Regel: $R(a, b, c) \rightarrow T(a, c) \Longrightarrow T(a, c) \rightarrow \exists\, B : R(a, B, c)$

invertiert: $Q(3, b) \rightarrow \underline{\exists\, C} : P(3, b, C)$

Invertierung Selektion

Ausgangsformel: $T(a, b, c) \wedge a = 3 \rightarrow P(3, b, c)$

Regel: $\exists\, x \in \mathbb{R} : R(a, b) \wedge a = x \rightarrow T(a, b) \Longrightarrow T(a, b) \rightarrow R(a, b)$

invertiert: $P(3, b, C) \rightarrow T(3, b, C)$

Invertierung Verbund

Ausgangsformel: $R(a, c) \wedge S(b, c) \rightarrow T(a, b, c)$

Regel: $R(a, b, c) \wedge S(b, d) \rightarrow T(a, b, c, d) \Longrightarrow T(a, b, c, d) \rightarrow R(a, b, c) \wedge S(b, d)$

invertiert: $T(3, b, C) \rightarrow \underline{R(3, C) \wedge S(b, C)}$

Der **letzte Schritt** ist die Zusammenführung der s-t tgds, um die gesuchte invertierte Anfrage zu erhalten. Dafür wird der Body der ersten Invertierung der Body der zusammengeführten s-t tgd und der Head der letzten Invertierung gemeinsam mit den Quantoren von allen vorherigen Heads (die, die durch Ersetzung umgesetzt wurden, werden dabei nicht berücksichtigt) der neue Head der zusammengeführten s-t tgd.

invertierte s-t tgd: $Q(3, b) \rightarrow \exists\, C : R(3, C) \wedge S(b, C)$

Ausgabe: $Q(3, b) \rightarrow \exists\, C : R(3, C) \wedge S(b, C)$

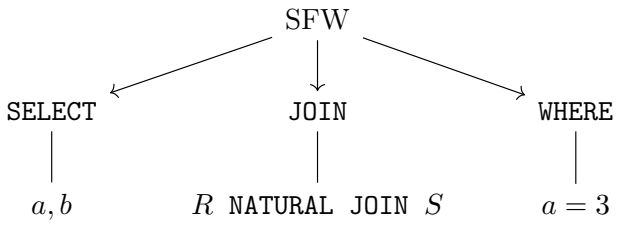
Praxis

Eingabe: SQL-Anfrage (ohne Gruppierung oder Aggregationen), Datenbankschema

```
SELECT a,b
FROM R NATURAL JOIN S
WHERE a = 3
```

Im **ersten Schritt** muss die eingegebene SQL-Anfrage geparst werden, damit eine sinnvolle Weiterverwendung der Anfrage im Algorithmus möglich ist. Dabei soll die Struktur der Anfrage sowie die konkreten Attribute, Tabellen, etc. in Form eines SFW-Baumes oder einer anderen Speicherstruktur, die diese Informationen enthält, überführt werden. Es muss also beispielsweise

möglich sein aus dem SFW-Baum abzuleiten welche Attribute bei `SELECT` ausgewählt werden. Analog muss das auch für alle anderen Bestandteile der Anfrage möglich sein. Bei geschachtelten Anfragen muss dabei auch die Schachtelung abgebildet werden. Dafür bietet sich die Java-Library *JSQL* an, diese kann jedoch auch durch eine ähnliche Bibliothek ersetzt werden.



Für jeden Teilbaum, muss dann im **zweiten Schritt** erst der `JOIN`-Teil, dann der `WHERE`-Teil und zum Schluss der `SELECT`-Teil in s-t tgd überführt werden. Dabei wird wieder jeweils der Head der letzten s-t tgd zum Body der folgenden s-t tgd. Wenn die Heads von mehreren Vorgängern identisch sind, dann wird nur ein neuer body genutzt. In der Speicherung muss auch an dieser Stelle weiterhin mitgeführt werden, zu welchem Block die jeweilige s-t tgd gehört. Das dient der Vereinfachung der Invertierung im nächsten Schritt.

$$R(a, c) \wedge S(b, c) \rightarrow T(a, b, c) \qquad \texttt{JOIN}$$
$$T(a, b, c) \wedge a = 3 \rightarrow P(3, b, c) \qquad \texttt{WHERE}$$
$$P(3, b, c) \rightarrow Q(3, b) \qquad \texttt{SELECT}$$

Beim **dritten Schritt** ändert sich im Vergleich zum Theorie Teil nicht allzu viel. Auch hier werden wieder die Basisregeln zur Invertierung (siehe Abschnitt 12 im Anhang) genutzt und die zuvor aufgestellten s-t tgds von hinten nach vorne invertiert. Dabei wird der Head der letzten s-t tgd der Body der neuen ersten s-t tgd und auch hier wird dann der Head der neuen invertierten s-t tgd zum Body der folgenden invertierten s-t tgd. Da bekannt ist, welche s-t tgd auf welche Operation zurückzuführen ist, können an der Stelle die Regeln zur Invertierung gezielt angewendet werden. Der Algorithmus muss also nicht selbst entscheiden, welche Regel passt. Eine weitere Vereinfachung in der Praxis ist, dass die Quantoren mitgeführt werden können, dabei ist jedoch nicht garantiert, dass es sich durchgängig um eine s-t tgd handelt.

Invertierung Projektion

Ausgangsformel: $\underline{P(3,b,c)} \rightarrow Q(3,b)$

Regel: $R(a,b,c) \rightarrow T(a,c) \Longrightarrow T(a,c) \rightarrow \exists\, B : R(a,B,c)$

invertiert: $Q(3,b) \rightarrow \exists\, C : P(3,b,C)$

Invertierung Selektion

Ausgangsformel: $T(a,b,c) \wedge a = 3 \rightarrow P(3,b,c)$

Regel: $\exists\, x \in \mathbb{R} : R(a,b) \wedge a = x \rightarrow T(a,b) \Longrightarrow T(a,b) \rightarrow R(a,b)$

invertiert: $\exists\, C : P(3,b,C) \rightarrow \exists\, C : T(3,b,C)$

Invertierung Verbund

Ausgangsformel: $R(a,c) \wedge S(b,c) \rightarrow T(a,b,c)$

Regel: $R(a,b,c) \wedge S(b,d) \rightarrow T(a,b,c,d) \Longrightarrow T(a,b,c,d) \rightarrow R(a,b,c) \wedge S(b,d)$

invertiert: $\exists\, C : T(3,b,C) \rightarrow \underline{\exists\, C : R(3,C) \wedge S(b,C)}$

Im **letzten Schritt** muss jetzt nur der Body der ersten s-t tgd mit dem Head der letzten s-t tgd verbunden werden um die vollständig invertierte s-t tgd zu erhalten. Da nicht durchgängig garantiert werden kann, dass es sich tatsächlich um formal korrekte s-t tgds handelt muss an dieser Stelle überprüft werden, ob es sich bei der invertierten s-t tgd tatsächlich um eine richtige s-t tgd handelt.

invertierte s-t tgd: $Q(3,b) \rightarrow \exists\, C : R(3,C) \wedge S(b,C)$

Ausgabe: $Q(3,b) \rightarrow \exists\, C : R(3,C) \wedge S(b,C)$

9.1.3. Invertierung in Backchase

Das Ergebnis der Invertierung ist die Grundlage für den Backchase-Schritt. Dieser soll zukünftig ebenso wie der Chase-Schritt aus ProSA heraus benutzbar sein. Dafür muss die entsprechende Methode in ChaTEAU aufgerufen und die s-t tgd übergeben werden. Außerdem soll die Invertierte Anfrage als s-t tgd sowie das Ergebnis, also die minimale Teildatenbank, in dem GUI angezeigt werden.

9.1.4. Einbindung der Invertierung in ProSA

Anja Wolpers

Das Modul zur Invertierung kann als eigenes Projekt oder als Teil-Modul des ProSA-Projektes realisiert werden. Wenn es als Projekt realisiert wird, muss es wie auch der Parser und ChaTEAU eingebunden werden, um genutzt werden zu können. Soll das Modul innerhalb des Projektes geschrieben werden, empfehlen wir, dies in Form eines neuen Controllers zu tun. Dann kann die Invertierungs-Methode von dort aufgerufen werden und der Themenbereich bleibt in sich geschlossen in einer Klasse.
Die Konvertierung ist wie oben beschrieben Teil des Backchase-Schritts. Dementsprechend muss die Konvertierungs-Methode auch in der Backchase-Methode im ChateauController aufgerufen werden und das Ergebnis dann als Argument an die Methode `runChase` übergeben werden.

```
backchaseStep(query, log, resultTextArea, inverseQueryTextArea, databaseInstance){
   invert input query
   display inverted query in inverseQueryTextArea
   start Backchase logging
   call runChase using the inverted query
   return backchase result
}
```

9.2. Provenance-Ergebnis

Anja Wolpers

Ein weiterer Teil des ProSA-Algorithmus, der noch implementiert werden muss, ist die Dokumentation von Daten-Provenance. Dazu müssen unter Berücksichtigung der Wahl der Provenance-Stufe („Where“, „Why“ oder „How“) bei der Anfrage-Ausführung Tupelidentifikatoren mitgeführt werden. Diese müssen dann der Provenance-Stufe entsprechend dargestellt bzw. in Zusammenhang gebracht werden. Diese Informationen können auch dazu genutzt werden, um Informationen in der minimalen Subdatenbank zu rekonstruieren, die im Laufe der Anfrage-Bearbeitung verloren gegangen sind [2].

9.3. Einbindung vom SQL-zu-s-t tgd-Parser

Anja Wolpers

Zur Einbindung des Parsers muss erst das Java-Projekt des Parsers heruntergeladen und unter Maven installiert werden und schließlich die Abhängigkeit zu dem Projekt in die pom-Datei unseres Projektes hinzugefügt werden. Dann kann entsprechend der API des Parsers seine Funktionalität in den Programmablauf integriert werden.

Der Parser kann von `https://git.informatik.uni-rostock.de/drift/sql2sttgd` heruntergeladen werden. Dann kann er wie auch das ChaTEAU-Projekt über IntelliJ oder die Kommandozeile unter Maven installiert werden. Eine genauere Anleitung ist im Abschnitt 6.1 zu finden.

Um den Parser als Abhängigkeit in ProSA einzubinden, müssen die Maven-Informationen aus der pom-Datei des Parsers als Abhängigkeit in die pom-Datei von ProSA eingefügt werden.

Schließlich kann der Parser im Programmablauf genutzt werden. Dazu muss in der Klasse `ChateauController` in der Methode `chaseStep` die Variable `stTgd` mit einem durch den Parser generierten File gleichgesetzt werden. Als Argument sollte dem Parser die SQL-Anfrage als String `query` und die Datenbankverbindung `connection` übergeben werden können. Derzeit wird erst die Anfrage des Nutzers mit einem veralteten, in der Anwendung inkludiertem Parser gelesen. Dieser Aufruf kann gelöscht werden. Danach wird ein schon ins richtige Format für ChaTEAU konvertiertes File geladen, um die Kommunikation mit ChaTEAU wie im finalen Programmablauf geplant stattfinden lassen zu können. Auch diese Zeile muss gelöscht werden.

Abschließend sollte noch die Log-Funktion von ProSA genutzt werden und auch Fehler sollten angezeigt und geloggt werden. Als Beispiel dafür kann die Methode `parseToStTgd` in der Klasse `ChateauController` dienen, die durch den neuen Parser ersetzt werden wird.

Teil I.
Anhang

10. Arbeitsweise

Anne-Sophie Waterstradt

Unser Fortschritt wurde hauptsächlich von wöchentlich Treffen getragen, in denen wir den aktuellen Stand des Projekts besprochen haben. Bei Bedarf haben wir uns zusätzlich getroffen. In diesem großen wöchentlichen Treffen haben wir besprochen, was in der kommenden Woche alles gemacht werden sollte und haben diese Aufgaben dann unter uns verteilt. Meistens haben an den Aufgaben mehrere von uns zusammen gearbeitet. Dafür haben wir uns nach Bedarf in Teilgruppen getroffen.

11. Zeitplan

Anne-Sophie Waterstradt

Wie in den Abbildungen 11 und 12 zu erkennen, lässt sich unser Zeitplan in drei Phasen einteilen. Zunächst haben wir ein Konzept erarbeitet, in der zweiten Phase haben wir dieses implementiert und in der letzten Phase haben wir dann unseren Projektbericht geschrieben. In der Konzeptphase wollten wir zunächst die bereits vorhandenen Teile unseres Projekts verstehen. Auf Grund der Tatsache, dass ProSA und ChaTEAU auf vielen Grundlagen aufbauen, mit welchen wir uns bisher noch nicht beschäftigt hatten, haben wir für diese Phase viel Zeit eingeplant. Danach wollten wir einen Papierprototypen für die Benutzeroberfläche entwickeln. Für diese Phase haben wir vier Wochen eingeplant. In den letzten zwei Wochen dieser Phase sollte gleichzeitig die Erstellung des Konzeptdokuments stattfinden.

In der zweiten Phase haben wir uns vorgenommen unseren Papierprototypen zu implementieren. Dafür haben wir fünf Wochen eingeplant. Die Phase des Debuggens beginnt schon zwei Wochen vor Ende das Implementierens und dauert insgesamt bis zu drei Wochen.

In der letzten Phase hatten wir vor unseren Projektbericht zu schreiben. Dafür haben wir drei Wochen eingeplant. In der letzten Woche des Schreibens soll dann die Phase des Testlesens beginnen. Für das Testlesen des Projektberichts haben wir zwei Wochen eingeplant.

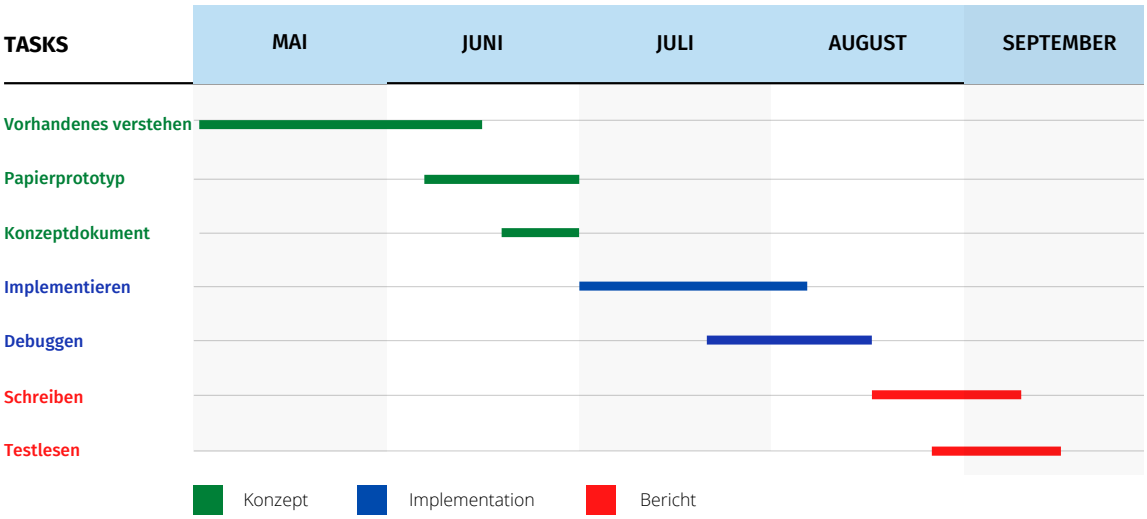


Abbildung 11: Ursprüngliches GANTT-Diagramm

11.1. Umsetzung des Zeitplans

Die Konzeptphase konnten wir genauso umsetzen wie geplant. Mit der Implementierungsphase konnten wir daher pünktlich anfangen. Allerdings hat die Implementierung etwa eine Woche länger gedauert als geplant. Dementsprechend hat sich das Debuggen auch länger hinausgezögert als geplant. Mit der Phase des Debuggens wurden wir zwei Wochen später fertig als geplant. Dies hat auch die Phase des Berichtschreibens verzögert. Damit wurden wir auch erst zwei Wochen später fertig als ursprünglich geplant.

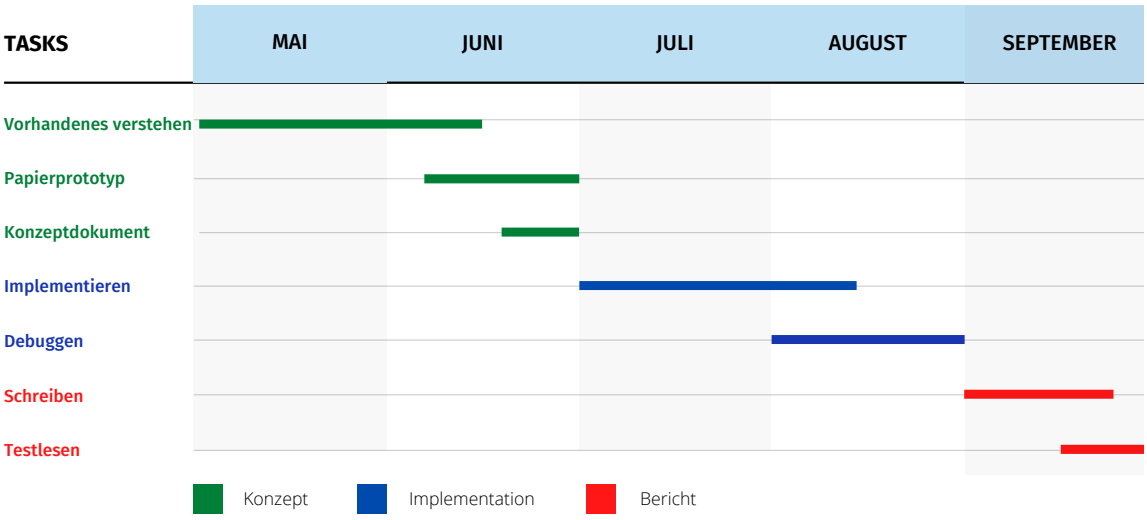


Abbildung 12: Aktualisiertes GANTT-Diagramm

12. Invertierung der Evaluationsoperatoren

Die folgenden Invertierungen wurden uns im Rahmen unseres KSWS-Projektes von Tanja Auge vorgegeben. Sie bilden die Grundlage für unser vorgestelltes Konzept zur Invertierung von s-t tgds.

Projektion

Formel: $R(a,b,c) \rightarrow T(a,c)$

Invertierung: $T(a,c) \rightarrow \exists\, B : R(a,B,c)$

Selektion

Formel: $\exists\, x \in \mathbb{R} : R(a,b) \wedge a\, \mathcal{O}\, x \rightarrow T(a,b),\ \mathcal{O}\ \in\ \{<, \leq, =, \geq, >, \neq\}$

Invertierung: $T(a,b) \rightarrow R(a,b)$

natürlicher Verbund

Formel: $R(a,b,c) \wedge S(b,d) \rightarrow T(a,b,c,d)$

Invertierung: $T(a,b,c,d) \rightarrow R(a,b,c) \wedge S(b,d)$

Kopieren

Formel: $R(a,b,c) \rightarrow T(a,b,c)$

Invertierung: $T(a,b,c) \rightarrow R(a,b,c)$

Umbennung

Formel: $R(a,b,c) \rightarrow T(d,e,f) \wedge a = d, b = e, c = f$

Invertierung: $T(d,e,f) \rightarrow R(a,b,c) \wedge d = a, e = b, f = c$

Vereinigung

Formel: $R(a,b,c) \rightarrow T(a,b,c),\ S(a,b,c) \rightarrow T(a,b,c)$

Invertierung: $T(a,b,c) \rightarrow R(a,b,c),\ T(a,b,c) \rightarrow S(a,b,c)$

Schnitt

Formel: $R(a,b,c) \wedge S(a,b,c) \rightarrow T(a,b,c)$

Invertierung: $T(a,b,c) \rightarrow R(a,b,c) \wedge S(a,b,c)$

Differenz

Formel: $R(a,b,c) \wedge \neg S(a,b,c) \rightarrow T(a,b,c)$

Invertierung: $T(a,b,c) \rightarrow R(a,b,c),\ T(a,b,c) \rightarrow \exists\, A,B,C : S(A,B,C)$

Arithmetische Operationen: *skalare Multiplikaiton, Division, Addition, Subtraktion*

Formel: $\exists\, c \in \mathbb{R} : R(a,b) \wedge d = a\, \mathcal{O}\, c \rightarrow T(d,b),\ \mathcal{O}\ \in\ \{\cdot, :, +, -\}$

Invertierung: $\exists\, c \in \mathbb{R} : T(d,b) \wedge a = d\, \mathcal{O}'\, c \rightarrow R(a,b),\ \mathcal{O}'$ Umkehrfunktion zu $\mathcal{O}$

Literatur

[1] Tanja Auge. „Extended Provenance Management for Data Science Applications.“ In: *PhD@ VLDB* 2652 (2020).

[2] Tanja Auge und Andreas Heuer. „Inverse im Forschungsdatenmanagement: Eine Kombination aus Provenance Management, Schema-und Daten-Evolution“. In: (2018).

[3] Tanja Auge und Andreas Heuer. „ProSA—Using the CHASE for Provenance Management“. In: *European Conference on Advances in Databases and Information Systems*. Springer. 2019, S. 357–372.

[4] Melanie Herschel, Ralf Diestelkämper und Houssem Ben Lahmar. „A survey on provenance: What for? What form? What from?“ In: *The VLDB Journal* 26.6 (2017), S. 881–906.