KSWS-Bericht Gruppe 3: Qualitätssicherung für ChaTEAU

Michael Albus, Lukas Görtz, Moritz Hanzig, Eduard Buch, Eric Maier

September 2021

Inhaltsverzeichnis

1 Einleitung

[Michael]

Im Sommersemester 2021 im Modul KSWS hatten wir als Gruppe 3 als Oberthema die Qualitätssicherung des Programms ChaTEAU. Die Aufgabe bezog sich auf das Programm aus Sicht eines Programmierers und umfasste deshalb, den Quellcode des Programms durchzusehen und hinsichtlich Benutzbarkeit, Verständlichkeit und Erweiterbarkeit zu überprüfen, die Funktionalität der Anwendung zu testen und gegebenenfalls zu überarbeiten oder reparieren, sowie eine Dokumentation zu erstellen.

Begleitend gab es einen Vorlesungsteil, der die mit ChaTEAU umgesetzten theoretischen Grundlagen behandelte.

Wir nutzen für die Arbeit das GitLab der IEF, wo die Änderungen auf Code-Ebene einsehbar sind und die Diagramme in voller Auflösung vorliegen.

Die Ergebnisse unserer Arbeit werden in diesem Bericht festgehalten.

2 Arbeits-/Vorgehensweise

[Moritz]

Wir haben uns ein recht strenges Verfahren überlegt, um unsere *Arbeit selbst* in der Qualität abzusichern. Dazu gehört zum Beispiel, keine Änderung am Code ohne zugehöriges ausführliches Issue durchzuführen. Zu jedem Issue haben wir einen separaten Branch im Gitlab-Repository erstellt und vor jedem Merge der Änderung in unseren Master-Branch sichergestellt, dass die Änderungen durch mindestens eine zweite Person *approved*, d.h. abgesegnet wurden. Somit sind bei allen Änderungen der Grund bzw. das Problem ersichtlich, weshalb diese Änderung am Code nötig ist. Das Repository selbst ist somit eigentlich die ausführlichste Dokumentation unserer Arbeit. Dieser Bericht liefert allerdings einige zusätzliche Erläuterungen und Diskussionen, die wir geführt haben.

3 Kurzübersicht ChaTEAU

[Lukas]

ChaTEAU steht für „**Cha**se for **T**ransforming, **E**volving, and **A**dapting databases and queries, **U**niversal approach“. ChaTEAU ist ein Tool zum Durchführen des Chase-Algorithmus auf Datenbankschemata. Dies geschieht mithilfe von Instanzen und Anfragen zusammen mit TGDs, EGDs und st-TGDs. Um im Normalfall die Terminierung zu garantieren, sind fünf verschiedene Tests implementiert, welche die meisten Fälle der Nicht-Terminierung abfangen (Rich Acyclicity, Weak Acyclicity, Safety, Acyclicity, Acyclicity with EGD rewriting).[1] Mit dem Chase-Algorithmus können Abhängigkeiten in Datenbankschemata eingearbeitet werden. Dafür wird das Schema als Tableau übergeben. Ein Tableau ist eine Darstellung in Tabellenform, bei der die Attribute in einer Spalte und die Datenbankschemata in einer Zeile stehen[2]. Weiterhin lassen sich Abhängigkeiten und Sichten in Anfragen einarbeiten, indem die Sichten als st-TGD formuliert sind. Data Cleaning, Datenintegration und Datenaustausch lassen sich durch das Einarbeiten von Abhängigkeiten in Instanzen erreichen.

[1] Näheres in [2]

[2] vgl. S.3,[6]

Im Rahmen dieses Berichtes nutzen wir den Chase beispielhaft für das Data Cleaning: Dieses beinhaltet mehrere Funktionen: Nullwerte werden gefüllt, falsche Werte korrigiert und Redundanzen entfernt.

Am Beispiel einer Studenten-Datenbankinstanz lassen sich die Auswirkungen visualisieren: Student Erik Erikson hat die Matrikelnummer 123 und studiert Informatik. Sein Bruder Olaf Erikson studiert ebenfalls Informatik. Die Angabe der Fakultät des Bruders fehlt. Des Weiteren nimmt Erik am Studienfach Datenbanken teil, hier muss noch die Angabe des Unterrichtsraumes (Raum 1) integriert werden. Der Nachname von Erik ist ursprünglich in der Unterrichtstabelle falsch geschrieben. Student Paul Maier hat geheiratet und heißt nun Paul Müller.

S(Vorname, Nachname, Matrikelnummer, Studiengang, Fakultät)		
S(Erik, Erikson, 123, Informatik, IEF)	$\xrightarrow{Chase}$	S(Erik, Erikson, 123, Informatik, IEF)
S(Olaf, Erikson, 124, Informatik, ?)	$\xrightarrow{Chase}$	S(Olaf, Erikson, 124, Informatik, IEF)
S(Paul, Maier, 125, Informatik, IEF)	$\xrightarrow{Chase}$	S(Paul, Müller, 125, Informatik, IEF)
S(Erik, Erikson, 123, Informatik, IEF)	$\xrightarrow{Chase}$	
U(Matrikelnummer, Vorname, Nachname, Unterrichtsfach, Unterrichtsraum)		
U(123, Erik, Ericson, DB, ?)	$\xrightarrow{Chase}$	U(123, Erik, Erikson, DB, Raum 1)

Nach der Ausführung des Chase wurden die Nullwerte ersetzt, die redundante Speicherung von Tupeln entfernt, neue Informationen integriert, alte Werte ausgetauscht und falschgeschriebene Werte korrigiert.

Im Allgemeinen bekommt der Chase-Algorithmus eine Instanz und Integritätsbedingungen als Input und liefert eine Instanz als Output. Hierbei durchläuft das Programm eine Schleife, bis sich von einem Schleifendurchlauf zum Nächsten nichts mehr ändert: Für jede Integritätsbedingung wird ein Trigger erstellt und nun werden alle aktiven Trigger ausgeführt. Wenn ein Trigger eine (st-)TGD ist, werden benötigte Nullwerte erzeugt und danach neue Fakten hinzugefügt. Wenn es sich um eine EGD handelt, werden Werte gleichgesetzt und wenn nötig abgebrochen.

4 Kurzbeschreibung Chase-Algorithmus

[Lukas,Eduard]

Der Chase-Algorithmus arbeitet eine gegebene Menge an Abhängigkeiten $\star$ in ein Objekt $\bigcirc$ ein: $\text{CHASE}_\star(\bigcirc) = \circledast$[3]

Eingebettete Abhängigkeiten (EDs):
Eine ED ist ein prädikatenlogischer Ausdruck der Form $\forall x \forall y : \varphi(x, y) \rightarrow \exists z : \psi(x, z)$. x,y und z sind Tupel aus Variablen, $\varphi(x, y)$ und $\psi(x, z)$ Konjunktionen über atomare Formeln. Diese Konjunktionen bilden die Abhängigkeit, wobei φ den Rumpf und ψ den Kopf bilden.

[3] Definitionen entnommen und zusammengefasst nach [2], [3] und [7]

Spezielle EDs:

- **EGD**
  Eine gleichheitserzeugende Abhängigkeit hat die Form $\forall x : F_1(x) \rightarrow x_1 = x_2$. Die Variablen werden nicht existenzquantifiziert und der Kopf besteht aus einem Gleichheitsatom.
  EGDs werden in Tableaus als Schablonen benutzt, um Nullwerte zu füllen.
- **TGD**
  Eine tupelgenerierende Abhängigkeit hat die Form $\forall x : F_1(x) \rightarrow \exists y : F_2(x, y)$. F_1 und F_2 sind Konjunktionen von atomaren Funktionen über Variablen aus x bzw. x und y. Eine TGD besitzt keine Gleichheitsatome. Sind Variablen aus dem Kopf existenzquantifiziert, ist die TGD eingebettet, andernfalls voll.
  TGDs erzeugen im Tableau neue Tupel, die existenzquantifizierten Variablen erhalten nummerierte Nullwerte.
- **st-TGD**
  Eine Source-to-Target TGD ist ein Spezialfall der TGD, bei dem F_1 eine Konjunktion über ein Quellschema S und F_2 eine Konjunktion über ein Zielschema T ist.

Trigger:
Ein Trigger für eine Abhängigkeit existiert, wenn mithilfe eines Homomorphismus der Rumpf der Abhängigkeit auf einen Teil des Objektes abgebildet wird, beziehungsweise dass das Muster des Rumpfes im Objekt vorkommt.
Ein Trigger ist aktiv, wenn durch Anwendung des Kopfes der Abhängigkeit unter dem gleichen Homomorphismus neue Werte oder Tupel entstehen, also die Abhängigkeit im Objekt noch nicht erfüllt ist.
Pseudocode Chase:[4]

```
FOREACH Trigger t für eine Abhängigkeit g ∈ ⋆ DO
    IF t ist aktiv THEN
        IF g hat Form ∀x : F1(x) → ∃y : F2(x, y) (TGD) THEN
            F2 anwenden, neue Tupel in ◯ aufnehmen
        ELSE IF g hat Form ∀x : F1(x) → x1 = x2 (EGD) THEN
            IF x1, x2 sind Konstanten und x1 ≠ x2 THEN
                CHASE schlägt fehl
            ELSE IF x1 ist Konstante THEN
                x2 ← x1
            ELSE IF x2 ist Konstante
                    ∨ x1, x2 sind Nullwerte THEN
                x1 ← x2
```

Bedeutung:
Für jede Abhängigkeit wird über ihre Trigger iteriert. Das Ergebnis des Chase ist von der Reihenfolge der Triggeraufrufe abhängig, somit ist der Chase nichtdeterministisch. Zuerst wird überprüft, ob der momentan betrachtete Trigger aktiv ist. Ist dies nicht der Fall, ist die Abhängigkeit erfüllt, da die Anwendung des Triggers keinen Effekt hätte.

[4] siehe Seite 13 [5]

Wenn ein Trigger aktiv ist, wird danach unterschieden, was für eine Form von Abhängigkeit vorliegt.

Falls die Abhängigkeit eine TGD ist, wird dessen Kopf angewandt. Die daraus entstehenden Tupel werden in das Objekt (z.B. in die Instanz) aufgenommen.

Falls die Abhängigkeit eine EGD ist, wird je nachdem, ob die gleichzusetzenden Werte x_1 und x_2 Konstanten oder Nullwerte sind, etwas anderes getan. Daraus entstehen vier Fälle, wobei die letzten beiden im Pseudocode zusammengefasst sind.

1. Falls beide Werte Konstanten sind, also bereits Werte haben, und diese unterschiedlich sind, schlägt der CHASE-Algorithmus fehl, da Konstanten nicht gleichgesetzt werden können.
2. Falls nur x_1 eine Konstante ist, werden die Vorkommen von x_2 durch x_1 ersetzt.
3. Falls nur x_2 eine Konstante ist, werden die Vorkommen von x_1 durch x_2 ersetzt.
4. Falls beide Nullwerte sind, werden die Vorkommen von x_1 durch x_2 ersetzt.

In den Fällen 2 bis 4 ist die Abhängigkeit nun erfüllt und der nächste Trigger wird ausgewählt.

Beispiel einer EGD: `(example_2_I_EGD.xml)`
Instanz, auf die Chase angewandt werden soll:
Participants(2, 3)
Participants(7, 3)
Students(3, "Miller", #N_firstname_1, #N_course_2)
Students(3, #N_lastname_1, "Max", #N_course_1)

Abhängigkeit:
Students(#V_student_id_2, #V_lastname_1, #V_firstname_1, #V_course_1),
Students(#V_student_id_2, #V_lastname_2, #V_firstname_2, #V_course_2)
->
#V_firstname_1 = #V_firstname_2,
#V_lastname_1 = #V_lastname_2,
#V_course_1 = #V_course_2
In Worte gefasst: Wenn zwei Tupel der *Students* Relation dieselbe *student_id* haben, dann sollen auch deren *firstname*, *lastname* und *course* identisch sein. Es bedeutet also, dass zwei verschiedene Studenten sich nicht dieselbe Student_Id teilen können.

Durchführung Chase:
Um es einfach zu halten, werden im Folgenden nur die beiden aktiven Trigger behandelt.
Erster aktiver Trigger:
Students(#V_student_id_2, #V_lastname_1, #V_firstname_1, #V_course_1)
→
Students(3, #N_lastname_1, "Max", #N_course_1)
und
Students(#V_student_id_2, #V_lastname_2, #V_firstname_2, #V_course_2)
→
Students(3, "Miller", #N_firstname_1, #N_course_2)

Die Gleichungen im Kopf der Abhängigkeit werden durchlaufen:

1. *#V_firstname_1 = #V_firstname_2*
   - *#V_firstname_1* wurde abgebildet auf *"Max"* (Konstante)
   - *#V_firstname_2* wurde abgebildet auf *#N_firstname_1* (Nullwert)
   - Das ist Fall 2 einer EGD, die Vorkommen von *#N_firstname_1* werden ersetzt durch *"Max"*
2. *#V_lastname_1 = #V_lastname_2*
   - *#V_lastname_1* wurde abgebildet auf *#N_lastname_1* (Nullwert)
   - *#V_lastname_2* wurde abgebildet auf *"Miller"* (Konstante)
   - Das ist Fall 3 einer EGD, die Vorkommen von *#N_lastname_1* werden ersetzt durch *"Miller"*
3. #V_course_1 = #V_course_2
   - *#V_course_1* wurde abgebildet auf *#N_course_1* (Nullwert)
   - *#V_course_2* wurde abgebildet auf *#N_course_2* (Nullwert)
   - Das ist Fall 4 einer EGD, die Vorkommen von *#N_course_1* werden ersetzt durch *#N_course_2*

Zweiter aktiver Trigger:
Students(#V_student_id_2, #V_lastname_1, #V_firstname_1, #V_course_1)
→
Students(3, "Miller", #N_firstname_1, #N_course_2)
und
Students(#V_student_id_2, #V_lastname_2, #V_firstname_2, #V_course_2)
→
Students(3, #N_lastname_1, "Max", #N_course_1)

Im Vergleich zum ersten aktiven Trigger wurden hier die beiden Tupel der Instanz getauscht. Es kommt hier somit das gleiche Ergebnis heraus, mit dem Unterschied, dass hier *#N_course_2* durch *#N_course_1* ersetzt wird.

Nachdem das redundante Tupel entfernt wurde, entsteht das folgendes Ergebnis:
Participants(2, 3)
Participants(7, 3)
Students(3, "Miller", "Max", #N_course_1)

Beispiel einer TGD: `(example_3_I_TGD.xml)`
Instanz, auf die Chase angewandt werden soll:
Grades(7, 3, #N_semester_1, #N_grade_1)
Participants(2, 3)
Participants(7, 3)
Students(3, "Miller", "Max", "Electrical engineering")
Students(1, "May", "Maria", "Electrical engineering")

Abhängigkeit:
Students(#V_student_id_1, #V_lastname_1, #V_firstname_1, #V_course_1),
Participants(#V_module_id_1, #V_student_id_1)
->
Grades(#V_module_id_1, #V_student_id_1, #E_semester_1, #E_grade_1)
In Worte gefasst: Wenn ein Tupel der *Students* Relation und ein Tupel der *Participants* Relation dieselbe *student_id* haben, dann soll es ein entsprechendes Tupel der *Grades* Relation geben. Es bedeutet also, dass ein Student, der an einem Modul teilnimmt, dafür eine Note erhalten soll.

Durchführung Chase:
Der Einfachheit halber werden wieder nur die aktiven Trigger betrachtet.
Aktiver Trigger:
Participants(#V_module_id_1, #V_student_id_1)
→
Participants(2, 3)
und
Students(#V_student_id_1, #V_lastname_1, #V_firstname_1, #V_course_1)
→
Students(3, "Miller", "Max", "Electrical engineering")

Durch das Anwenden des Kopfes der Abhängigkeit entsteht folgendes Tupel, welches in die gegebene Instanz mitaufgenommen wird:
Grades(2, 3, #N_semester_2, #N_grade_2)

Insgesamt wird also folgendes Ergebnis erzeugt:
Grades(2, 3, #N_semester_2, #N_grade_2)
Grades(7, 3, #N_semester_1, #N_grade_1)
Participants(2, 3)
Participants(7, 3)
Students(3, "Miller", "Max", "Electrical engineering")
Students(1, "May", "Maria", "Electrical engineering")

5 Diagramme

[Eric, Michael, Moritz]

Wir haben zu Dokumentationszwecken Diagramme wie ein Klassendiagramm und Sequenzdiagramme erstellt. Sie sind hier beispielhaft in einer kompakteren und übersichtlicheren Version aufgeführt und in voller Größe im Wiki des GitLab abgelegt. Abb. 1 zeigt eine vereinfachte Version des Klassendiagramms von ChaTEAU. Hier ist deutlich das Herzstück von ChaTEAU, die *Chase*-Klasse zu sehen, in welcher der Chase-Algorithmus implementiert ist. Klassen für die Ein- und Ausgabefunktionalität sind hier ausgespart. Dafür zeigt das Klassendiagramm übersichtlich die Komponenten für die Kernfunktionalität des Chase: Es werden `integrityConstraints` in eine `Instance` einarbeitet, indem er `homomorphisms` zwischen deren Termen findet.

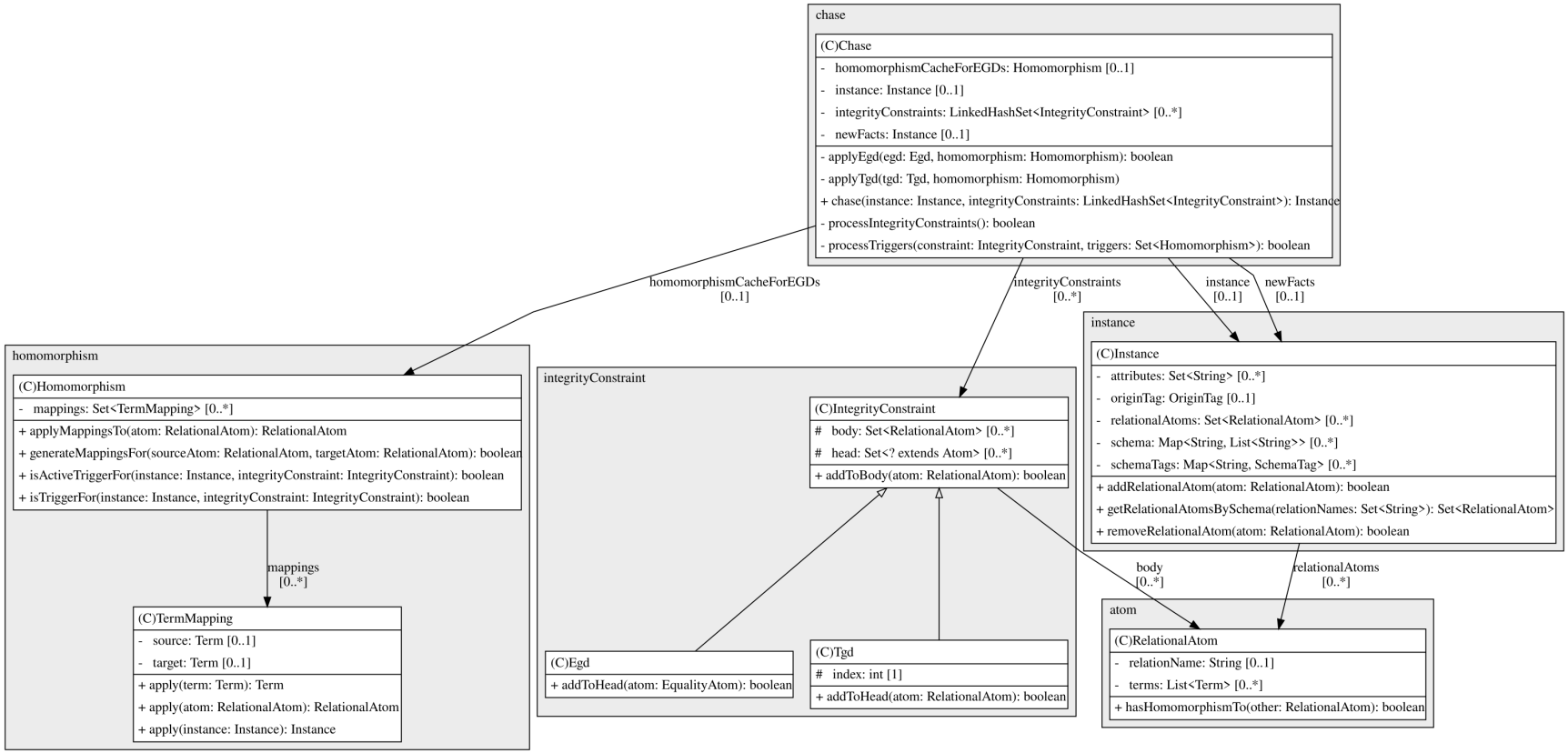


Abbildung 1: Klassendiagramm

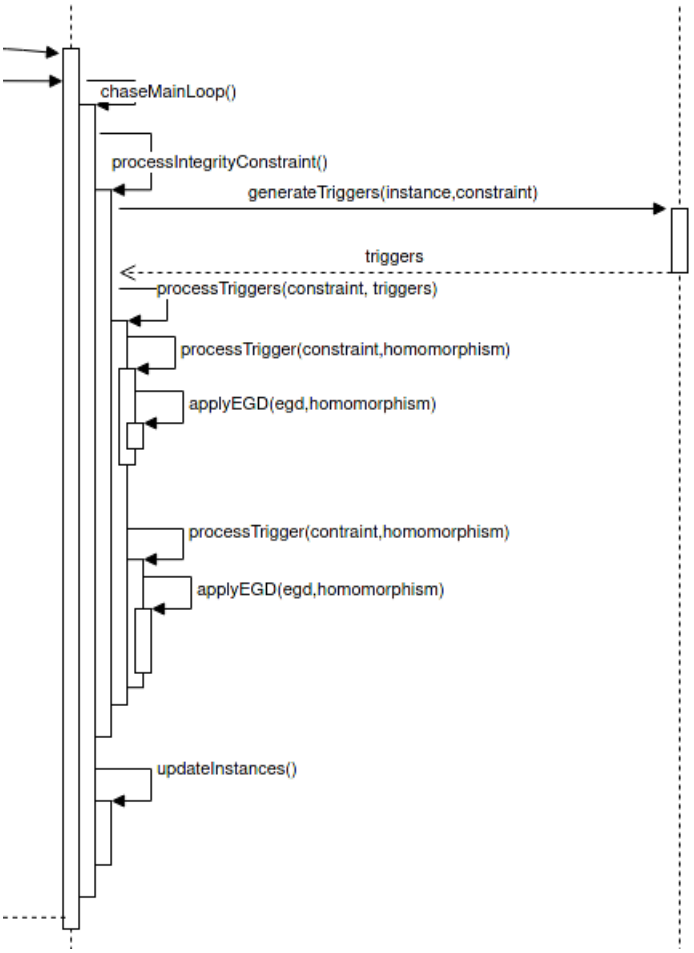


Abbildung 2: Sequenzdiagramm EGD - Chase-Teil

6 Verschiedene Datentypen der Elemente in den Instanz-Tupeln

[Moritz,Lukas]

Bei der Beschäftigung mit dem Format der XML-Eingabedateien fiel auf, dass es ein XML-Attribut `type` für die einzelnen Elemente der Instanz-Tupel gibt. Darüber können verschiedene Attributwerte verschiedenen Datentypen zugeordnet und in ChaTEAU geparst werden. Derzeit kann ChaTEAU nur die Datentypen `int`, `double` und `String` erkennen und entsprechend verarbeiten. Dabei wurden zuerst Ungenauigkeiten beseitigt: Bei einer gegebenen, aber unbekannten Datentypangabe wurde keine Fehlermeldung erzeugt, und eine fehlende Datentypangabe standardmäßig als String interpretiert (vgl. Issue #8 https://git.informatik.uni-rostock.de/ta093/chateau-ksws/-/issues/8).
Weiterhin war eine Überlegung, die unterstützten Datentypen durch Nutzung von Java Ge-

nerics zu erweitern. Die Idee wurde jedoch wieder verworfen, da zwar durch Einlesen eines Strings mittels `Class.forName(...)` generisch die entsprechende Java-Klasse aus dem im XML angegebenen Datentyp ableitbar sind, aber das Parsen trotzdem noch spezifisch für jeden Typ implementieren muss. Daher sind generische Klassen nicht umzusetzen. Wenn die Unterstützung von mehr Datentypen in Zukunft explizit gewünscht ist, wäre die Nutzung von objektrelationalem Mapping z.B. mit Hibernate zu empfehlen, um nicht jeden Parser einzeln zu implementieren.
Es gibt auf der anderen Seite auch das durchaus stichhaltige Argument, dass die Unterstützung verschiedener Datentypen überhaupt nicht im Fokus von ChaTEAU als Prototyp liegen sollte, da der Chase-Algorithmus auch in der Theorie überhaupt keine Datentypen unterscheidet. Hier erhöht schon die Unterscheidung von `int`, `double`, und `String` nur unnötigerweise die Komplexität des Programms.
Insofern ist die Beschränkung auf *wenige, aber einige* Datentypen ein guter Mittelweg, um zu zeigen, dass die Unterstützung in der Praxis möglich ist (denn dafür gibt es Anwendungsfälle), aber die Komplexität auch nicht zu groß ist.

7 Allgemeinverständliche Zusammenfassungen der Issues

[Michael, Eduard]

No.1 - Unnötige Implementierungsdetails für Interfaces entfernen Innerhalb der Programmierung können bestimmte Datentypen oder Funktionalitäten mit unterschiedlichen Implementierungen durch den Programmierer umgesetzt werden. Diese Implementierungen haben teils unterschiedliche Eigenschaften, können aber über bestimmte garantierte Eigenschaften oder Fähigkeiten zusammengefasst werden. Es ist günstig, bei der Programmierung darauf zu achten, dass man für interne Zwecke die zu verwendende Implementierung so wenig wie möglich einschränkt. Man schreibt am besten nur die zu garantierenden Eigenschaften/Fähigkeiten vor. Beispielweise gibt der Programmierer vor, dass eine Liste verwendet werden soll, ohne dass er die Nutzung einer bestimmten Implementierung einer Liste erzwingt. Damit macht den Code leichter wart- und erweiterbar und erleichtert man nachfolgenden Programmierern die Arbeit. Eine weiterführende Erklärung findet sich zum Beispiel im Java Kurs von Abts[5].

Bearbeiter: Moritz

No.2 - Java Version Von der Programmiersprache Java existieren verschiedene Versionen, die sich hinsichtlich ihrer Fähigkeiten und der Länge des Supportzeitraums unterscheiden. Wir haben hier von der Java-Version 16 auf 11 umgestellt, weil diese alle bisher verwendeten Konstrukte in der Programmierung unterstützt und eine Version mit besonders langem Supportzeitraum ist („LTS“). Die Funktionalität wird dadurch nicht verändert, jedoch die Versorgung mit Updates wegen Sicherheitslücken in der verwendeten Programmiersprache für eine längere Zeit gewährleistet(bei V.16 war es nur ein halbes Jahr).

Bearbeiter: Michael, Moritz

No.3 - Testcases ChaTEAU verwendet als Eingabeformat Dateien im Format XML. Diese Dateien können hinsichtlich ihrer inneren Struktur und ihrer inneren Logik Fehler aufweisen. Ein Programm sollte auch mit fehlerhaften Nutzereingaben umgehen können,

[5] Seite 211 [1]

was hier getestet wurde und auch in Zukunft für weitere Situationen getestet werden sollte - Tests sind niemals erschöpfend. Mit nicht standardkonformem XML ging ChaTEAU unterschiedlich um: von Darstellungsfehlern, die nicht die Funktion beeinträchtigen bis hin zu Fehlermeldungen betreffend die Eingabe.

Bearbeiter: Eduard, Eric, Michael, Moritz

No.4 - Pfadtrenner Für das Schreiben des Ergebnisses in eine Datei wurde vormals ein Programmteil in der Datei *OutputWriter.java* verwendet, der unter Betriebssystemen außer Windows einen Fehler bei der Benennung der Datei verursacht(Es wird ein Teil des Verzeichnispfades als Dateiname verwendet, da die Pfadangaben bspw. in unixartigen Betriebssystemen anders gemacht werden als unter Windows). Es stellte sich heraus, dass diese Funktionalität inzwischen durch einen anderen Programmteil übernommen wird und lediglich noch der alte Programmteil im Programmcode existiert (mit einer entsprechenden Markierung). Insofern ist dieser Issue ohne weiteren Eingriff erledigt, führt aber zu Issue No.5.

Bearbeiter: Eric, Moritz

No.5 - Cleanup Wie unter anderem in Issue No.4 aufgefallen, exisitierten im Programmcode nicht mehr verwendete Teile(„deprecated“. Hierbei handelte es sich um Teile unterschiedlichen Typs:

- Klassen
- Methoden
- XML-Eingabedateien

Je nach Typ wurden diese Teile einfach nicht mehr verwendet oder waren - wie im Falle der Eingabedateien - gar nicht mehr verwendbar. Diese Teile wurden im Rahmen dieses Issues entfernt, um den Code leichter wartbar zu machen und auch um Fehler und Missverständnisse bei neuen Programmierern und Verwendern zu vermeiden. Für weitere, als deprecated markierte Teile, die jedoch erhalten bleiben sollen, da sie bspw. noch für eine laufende Masterarbeit gebraucht werden, wurden eindeutige Erklärungen hinzugefügt.

Bearbeiter: Moritz

No.6 - DTD erstellen ChaTEAU verwendet für die Eingabe XML-Dateien und erwartet eine Struktur mit bestimmten Elementen und Attributen. Dafür gab es noch keine Definition. Dies formulierten wir in Form einer DTD, die wir per XML-Header in den Beispieldateien mit diesen verknüpft haben.

Bearbeiter: Eduard, Michael

No.7 - InputReader Falls in der Eingabe die Tupel weniger Attribute enthielten als die zugehörige Relation, ließ sich kein sinnvolles Ergebnis erzielen. Um einen korrekten Umgang damit zu gewährleisten, prüft ChaTEAU nun diesen Fall und im Fehlerfall wird eine Exception geworfen.

Bearbeiter: Eric, Moritz

No.8 - Falsche Typdefinition Die Attribute konnten bislang vom Typ String, Double und Integer sein. Wenn beim Typ etwas anderes als Double oder Integer angegeben war, wurde auch bei unsinnigen/fehlerhaften Angaben standardmäßig String gewählt. Dieses Verhalten wurde dahingehend geändert, dass stattdessen eine Exception geworfen wird.

Bearbeiter: Lukas

No. 9 - Mögliche NullpointerException in Constant.java Für die Behandlung von Konstanten wurde im Programm ein Konstrukt verwendet, das unter bestimmten Umständen einen Fehler erzeugen kann, der das Programm zum Absturz bringt. Diese Möglichkeit wird jetzt mit einer Exception aufgefangen.

Bearbeiter: Moritz

No.10 - Tupel mit verschiedenen IDs werden als identisch eingelesen Im Eingabeformat-XML ist vorgesehen, die Instanzen mittels Identifiern unterschieden. Dieser Mechanismus wird innerhalb des Programms gar nicht verwendet, sondern die Tupel anhand ihrer Attributwerte unterschieden. Bearbeiter: Eric, Lukas, Moritz

No.11 - InputReader: Nullwerte in Instance Tupel mit null statt constant und type Im Zuge von Issue No.6 änderten wir, wie in den Eingabedateien Nullwerte in Tupeln dargestellt werden. Dies führte zu unerwünschtem Verhalten im InputReader. Dieses Verhalten korrigierten wir.

Bearbeiter: Eduard

No.12 - XML-Examples Korrektur weiterer, bisher nicht berücksichtigter XML-Eingabe-Dateien. In No.6 berücksichtigten wir nur Dateien, die bis dahin existiert hatten, neu hinzugekommene wurden hier angepasst.

Bearbeiter: Eduard

No.13 - DTD-Check In Issue No.6 erstellten wir eine DTD, welche aber nicht beim Einlesen überprüft wurde. Diese DTD-Validierung fügten wir hinzu. Falls sie fehlschlägt, wird eine Exception geworfen.

Bearbeiter: Eduard

No.14 - Cleanup Im Rahmen dieses Issues wurde der gesamte Code auf Lesbarkeit und Verständlichkeit hin überprüft und ggf. korrigiert, um die Wartbarkeit zu steigern und die Einarbeitung für neue Programmierer zu erleichtern. Das Stichwort lautet „Clean Code“, ein Konzept, das gute Lesbarkeit und Verständlichkeit von Programmcode mittels Regeln zu garantieren versucht.

Bearbeiter: Eric, Michael

No.15 - Testfälle Die Eingabedateien im alten XML-Format deckten bestimmte Fehlerfälle ab. Um sicherzustellen, dass die Prüfziele des Testfalles auch weiterhin gesichert werden, wurde kontrolliert, ob das erreicht wurde und ggf. nachgearbeitet.

Bearbeiter: Eduard, Lukas, Moritz

No.16 - Logger ChaTEAU bietet Protokollierung des Programmablaufs mit Exportfunktion. Diese Funktionalität überarbeiteten wir, so dass sie in sich konsistenter ist.

Die Exportfunktion wurde repariert - nun wird das gesamte Log in die Exportdatei geschrieben. Auch bei der Nutzung ohne Kommandozeile gibt es ein wie vor eine Loggingausgabe vermischt mit der Ausgabe der Resultate.

Bearbeiter: Eric, Michael

No.17 - Tester Die Klasse `Tester.java` hatte einen Bug, der dafür sorgte, dass die Ergebnisse von verschiedenen Terminierungstests davon abhingen, in welcher Reihenfolge die Tests ausgeführt wurden. Dies behoben wir durch Zurücksetzung des Zustands dieses Testers.

Bearbeiter: Moritz

No.18 - Verbesserung Tester In No.17 wurde der Bug durch Zurücksetzen des Testers behoben. Hier wurde eine verbesserte Lösung umgesetzt. Nun ist intern ausgeschlossen, dass ein und dieselbe Instanz(einfacheres Wort?) eines Testers mehrfach verwendet werden kann und damit durch eventuell noch vorhandene Zustände nicht kontrollierte Effekte entstehen.

Bearbeiter: Moritz

8 Von XML zur Chase-Instanz

[Eric]

ChaTEAU nimmt als Eingabe XML-Dateien an, welche nach den DTD-Regeln (Kapitel 9) formatiert sind. Für die Überführung einer XML-Datei in eine Chase-Instanz ist der `InputReader` zuständig. Dazu wird ein SAX Parser verwendet, um die XML-Daten vorerst in ein JDOM-Document zu laden. Dieses Document ist in einer Baumstruktur aufgebaut, das heißt es wird auf das Root-Element zugegriffen und über dieses die Child-Elemente ausgelesen.

Struktur einer XML-Datei:

```
<input>
    <schema>
        <relations>
            <relation name="...">
                <attribute name="..." type="..."/>
                ...
            </relation>
        </relations>
        <dependencies>
            <tgd|egd|st-tgd>
                <body>
                    <atom name="...">
                        <variable name="..." type="..." index="..." />
                        ...
                    </atom>
                </body>
```

```
                <head>
                    <atom name="...">
                        <variable name="..." type="..." index="..." />
                        ...
                    </atom>
                </head>
            </tgd|egd|st-tgd>
        </dependencies>
    </schema>
    <instance>
        <relationalAtom name="...">
            <tuple>
                <constant name="..." />
                ...
            </tuple>
        </relationalAtom>
    </instance>
</input>
```

Child-Elemente vom Root-Element

Das Root-Element besitzt 2 Child-Elemente (Schema und Instance/Query). Das Schema besteht wiederum aus Relations und Dependencies. Die Relations werden mit ihren Attribut-Namen und -Typen definiert, in unserem fortlaufendem Beispiel: Students mit Attribut student_id vom Typ integer, lastname vom Typ string usw..
Unter Dependencies wird die Art der Abhängigkeit (EGD, TGD, ST-TGD) angegeben und diese durch Body und Head definiert. Body/Head übergeben Variablen bestehend aus Namen, Typ und Index, wobei der Typ durch (`V` = „für alle“) und (`E` = „es existiert“) bestimmt wird.
Die Instance beinhaltet ein oder mehrere Tupel, welche zu einer definierten Relation passend sind. Diese geben Konstante oder Null Werte für die Attribute an (also die Einträge in einer Datenbank-Tabelle).
Eine Query wiederum, besteht (wie die Dependencies) aus einem Body und einem Head. Diese sind jedoch für unsere Zwecke nicht weiter relevant.

9 Überprüfen der Funktionalität

[Eduard, Moritz]

ChaTEAU akzeptiert als Eingabe XML-Dateien, die Schema, Abhängigkeiten und die Instanz/Anfrage beinhalten. Für Testzwecke existierten bereits viele Beispieldateien, die verschiedene Szenarien abdecken sollten. Zusätzlich zu diesen erstellten wir eigene Beispiele für bisher unbeachtete Randfälle. Zusammen nutzten wir diese, um zwei wichtige Teile von ChaTEAU zu überprüfen: das Einlesen von Eingabedateien und das Berechnen eines Ergebnisses.

Um das Einlesen von Eingabedateien zu testen, öffneten wir alle Beispiele einzeln in ChaTEAU. Daraufhin verglichen wir die „schönere Anzeige“ mit dem Inhalt der XML-Datei.

Zum Beispiel die Datei `example_1b_I_TGD.xml`:

- Anzeige eines Tupels der Instanz in ChaTEAU:
  Students(3, “Miller“, “Max“, “Electrical engineering“)
- Tupel in der XML-Datei:

```
<relationalAtom name=“Students“>
    <tuple>
        <constant name=“3“ />
        <constant name=“Miller“ />
        <constant name=“Max“ />
        <constant name=“Electrical engineering“ />
    </tuple>
</relationalAtom>
```

Hier sieht man leicht, dass die Anzeige in ChaTEAU mit dem Inhalt der XML-Datei übereinstimmt. Als Nächstes schauten wir, ob die Anwendung des implementierten Chase-Algorithmus ein korrektes Ergebnis berechnet. Da die gegebenen Beispiele sehr einfach gehalten waren, berechneten wir einfach ein eigenes Ergebnis und verglichen diese beiden. Falls diese übereinstimmten, war ChaTEAUs Ergebnis korrekt. Dies setzt natürlich voraus, dass wir den Algorithmus korrekt verstanden haben.

Als Resultat dieser Überprüfung kam heraus, dass alle bereits gegebenen XML-Dateien korrekt eingelesen und deren Ergebnis korrekt berechnet wurde. Dies zeigt selbstverständlich nicht, dass die Implementierung des Chase-Algorithmus korrekt ist, da wir keinen formalen Korrektheitsbeweis durchgeführt haben. Dennoch haben wir keine Zweifel daran, dass die Implementierung in den meisten Fällen das korrekte Ergebnis berechnet, da in den Eingabebeispielen in `src/test/resources` zumindest die grundlegenden Fälle abgedeckt sind.

Es gab allerdings auch noch unbrauchbare, veraltete Beispiele in `archive/outdated-examples`, die nicht mehr zu gebrauchen waren und von uns migriert wurden (siehe Issue #15). Außerdem haben wir uns noch weitere eigene Randfälle überlegt und entsprechende JUnit-Tests dafür geschrieben:

Eigene XML-Beispiele für Sonderfälle Wir haben uns in Issue #3 neue XML-Dateien mitsamt Unit-Tests für Sonderfälle überlegt. Die exakte Auflistung mitsamt deren Autorenschaft der einzelnen Fehlerfälle findet sich im Gitlab unter dem angegebenen Link und dem zugehörigen Mergerequest, aber grob zusammengefasst:

- Fälle, die nichts mit Chateau oder dem Chase an sich zu tun haben, sondern das XML selbst:
  - Einlesen einer leeren Datei
  - nicht *wohlgeformtes* XML
  - andere Zeichenkodierung als im XML-Header angegeben
- Dateien, deren Inhalt zwar korrektes XML ist, aber für ChaTEAU kaputt:
  - Instanzen, die in keine der gegebenen Relationen im Schema „passen“. Hierbei fiel uns auf, dass derzeit keine Fehlermeldung geworfen wird, wenn man ein Tupel in der Instanz hat, welches mehr oder weniger Attribute hat als im Schema angegeben. Dies wurde separat in Issue #7 bearbeitet.

– Ein Attribut in einem Tupel hat den Typ `String`, obwohl der Typ in der zugehörigen Relation `int` ist.

Beim Herumtesten und Überlegen dieser eigenen Beispiele kam die Idee auf, eine DTD zu definieren, die eine noch genauere Prüfung des Eingabe-XMLs ermöglicht und so weitere Fehler ausschließen kann. Diese DTD wurde in Issue #6 bearbeitet. Siehe auch Kapitel 9 für weitere Informationen zur DTD-Erstellung.

10 DTD für die Eingabe

[Eduard]

ChaTEAU erwartet in der Eingabedatei bestimmte XML-Elemente und Attribute. Jedoch gab es noch keine Definition, wie dessen Struktur auszusehen hat. Da dies aus unserer Sicht sehr hilfreich wäre, erstellten wir eine **D**ocument **T**ype **D**efinition (DTD). Diese definiert, wie die Eingabedateien im XML-Format strukturiert sind.

Dies ist sowohl für die Autoren der Eingabedateien als auch für die Programmierer von ChaTEAU eine Absicherung, dass, wenn eine XML-Datei der DTD entspricht, diese auch korrekt verarbeitet wird. Zusätzlich heißt es für die Programmierer, dass viele Randfälle, die zu Fehlern führen, schon bei der DTD-Validierung abgefangen werden.

Für die Erstellung der DTD erarbeiteten wir uns aus den existierenden Beispielen eine allgemeine Struktur, welche wir in der Form der DTD-Syntax formulierten. Dabei fielen uns einige Sachen auf.

Das XML-Element `constant` hatte in den Beispielen zwei allgemeine Formen. In der ersten Form hatte es nur das Attribut `name`, in der dessen Wert gespeichert wird. In der zweiten Form besitzte es die Attribute `name`, `type` und `index`. In dieser zweiten Form repräsentierte das Element eine Nullvariable. Das Attribut `name` war daher nicht der Wert, sondern der Name der leeren Variable. Das `type` Attribut wurde wahrscheinlich aus dem XML-Element `variable` übernommen, war bei `constant` jedoch redundant, da es immer denselben Wert "N" hatte und auch immer in Kombination mit `index` auftrat. Die Existenz des `index` Attributes allein ließ also schon auf die Nullvariable schließen. Deswegen entfernten wir das Attribut `type`. Das letzte Attribut `index` war aber noch nötig, da es dafür da ist, zwischen mehreren leeren, gleichnamigen Variablen unterscheiden zu können. Um die Lesbarkeit zu verbessern, definierten wir für Nullvariablen ein eigenes XML-Element namens `null`. Dies ersetzte die zweite Form des Elements `constant`, was jetzt immer nicht-leer ist.

- Tupel vor dieser Änderung:

  <tuple id="Student_2">
  <constant name="3" />
  <constant name="Miller" />
  <constant name="firstname" type="N" index="1" />
  <constant name="course" type="N" index="2" />
  </tuple>

- Tupel nach dieser Änderung:

  <tuple id="Student_2">
  <constant name="3" />

<constant name="Miller" />
<null name="firstname" index="1" />
<null name="course" index="2" />
</tuple>

In einigen Beispielen hatte das XML-Element `tuple` das Attribut `student-id`. Da es wenig Sinn macht, dies als Spezialfall für den Kontext Studierende bzw. Studium in die DTD aufzunehmen, ersetzten wir alle Vorkommen durch `id`, sodass es mit den restlichen Beispielen übereinstimmt.

Um die Eingabedateien mit der DTD zu verknüpfen, musste folgender XML-Header zum Anfang der Dateien hinzugefügt werden:

<?xml version="1.0" standalone="no" ?>
<!DOCTYPE input SYSTEM "input.dtd">

Dies bedeutet, dass die Eingabedatei als Wurzel das XML-Element `input` benutzt, dessen Definition in `input.dtd` zu finden ist. Ob die DTD eingehalten wurde, wurde aber noch nicht beim Einlesen der Datei überprüft. Glücklicherweise wurde das Einlesen mit der Klasse `SAXBuilder` implementiert, da die DTD-Validierung hier sehr leicht hinzugefügt werden kann, indem man `XMLReaders.DTDVALIDATING` als Parameter beim Instanziieren dieser Klasse in `InputReader.java` übergibt.

11 Überführung alter Beispiel-XML Dateien

[Lukas]

Dieser Abschnitt befasst sich mit den Änderungen im Issue #15.
Im Verlauf der Entwicklung von ChaTEAU sind mehrere Testdateien erstellt worden, die mit fortschreitender Entwicklung des Programms nach und nach unbrauchbar wurden, da das verwendetete Format nicht mehr durch neuere ChaTEAU-Versionen gelesen werden konnte. Da nicht ersichtlich ist, bis zu welcher ChaTEAU-Version eine alte Testdatei kompatibel ist und da auch keine zugehörigen JUnit-Tests zu ihnen existieren, verzichteten wir in den neu angelegten JUnit-Tests auf die Überprüfung des Chase-Ergebnisses und fokussierten uns stattdessen nur auf die Terminierungstests. Des Weiteren sind diese Testdateien speziell zum Testen eines Terminierungstests gedacht - deswegen ist das ChaTEAU-Ergebnis in diesem Test nicht relevant. Die originalen Dateien werden unverändert im Ordner `/archive` vorhanden bleiben, die angepassten neuen Versionen sind im Standard-Testordner (`src/test/resources`) zu finden. Die neu angelegten Dateien lassen sich in verschiedene Kategorien einteilen:

Kleine Änderung mit großer Wirkung

Durch die Veränderung eines einzelnen Quantors kann eine Eingabe von einer terminierenden zu einer nicht-terminierenden werden. Die dafür zuständige Testfunktion `testMinisculeChanges()` umfasst zwei Dateienpaare:
`example_6_WA(_terminiert_nicht)` sowie
`example_8b_Terminierung(_Fail)`. Die terminierenden Anfragen waren schon vorhanden, die nicht Terminierenden leiteten wir aus
`Beispiel_Terminierung_Fail.xml` und `Beispiel7_WA.xml` ab.

Safety-Kriterium
Die hier vorkommende Eingabe `example_8_SC` (im Archiv zu finden als `Beispiel_8_SC`) ist nicht stark/schwach azyklisch, jedoch terminiert der Chase trotzdem. Dies lässt sich durch den Test auf Azyklizität mithilfe des ADN++ Algorithmus (mit Constraint-Rewriting) zeigen. Mit dieser Anfrage wurde anscheinend das Safety-Kriterium erstmalig getestet, da es zwar nicht im Kommentar, aber im Namen der Datei erwähnt wird.

ADN++ Algorithmus
Die Datei `example_9_AC` ist im Archiv als `Beispiel_9_AC` zu finden. Mit dieser Anfrage wurde wahrscheinlich zum ersten Mal auf Azyklizität mithilfe des ADN++ Algorithmus getestet. Es wurde ein sehr spezieller Sonderfall gewählt, das sich darin zeigt, dass der Test auf Azyklizität mit Contraint-Rewriting je nach Prozessorleistung bis zu einer Minute benötigt.

Safe-Restriction-Kriterium
Die Terminierung der Anfrage (`example_12_SR`) lässt sich nicht mit dem Safety-Kriterium nachweisen, sondern nur mit dem Safe-Restriction-Kriterium. Dieses Kriterium ist zur Zeit noch nicht in ChaTEAU implementiert.

Stratifikation
Die Anfrage `example_11_Str` besitzt eine terminierende Sequenz und der Chase funktioniert, aber alle bisher vorhandenen Tests können dies nicht erkennen. Stratifikation erkennt diese Sequenz, ist aber nicht implementiert.

Lokale Stratifikation
`example_14_LS` beinhaltet eine ähnliche terminierende Sequenz zur vorherigen Anfrage, jedoch lässt diese sich nur mit der lokalen Stratifikation zeigen. Lokale Stratifikation ist nicht implementiert.

Induktive Restriktion
Das Safety- sowie das Safe-Restriction-Kriterium können nicht die Terminierung von `example_15_IR` erkennen. Hierfür wird das induktive Restriktions-Kriterium benötigt. Dieses lässt sich durch einen Test auf Azyklizität mit Constraint-Rewriting durchführen.

In Tabelle 1 ist die komplette Übersicht über die alten Beispieldateien aus dem Ordner `archive/outdated-examples` und deren Status bezüglich Nutzbarkeit/Migration in das neue XML-Format:

Tabelle 1: Migrationsstatus aller Dateien

Archiv	Neu	Bemerkung
Beispiel_Terminierung_Fail	example_8b_Terminierung_Fail	neu übersetzt
Beispiel1_I_TGD	example_1_I_TGD	bereits vorhanden
Beispiel1b	example_1b_I_TGD	bereits vorhanden
Beispiel2_I_EGD	example_2_I_EGD	bereits vorhanden
Beispiel3_I_TGD	example_3_I_TGD	bereits vorhanden
Beispiel4_I_EGD_Fail	example_4_I_EGD_Fail	bereits vorhanden
Beispiel5_Q_TGD	example_5_Q_TGD	bereits vorhanden
Beispiel5	example_5_terminiert _nicht	bereits vorhanden (Archiv hat Rechtschreibfehler)
Beispiel6_Q_TGD_EGD	example_6_Q_TGD_EGD	bereits vorhanden
Beispiel6_WA	example_6_WA	bereits vorhanden
Beispiel7_Q_Tableau	example_7_Q_Tableau	bereits vorhanden
Beispiel7_WA	example_6_WA_terminiert _nicht	neu übersetzt, passend umbenannt
Beispiel8_SC	example_8_SC	neu übersetzt, (Safety-Kriterium)
Beispiel8_Terminierung	example_8b_Terminierung	neu übersetzt
Beispiel9_AC	example_9_AC	neu übersetzt, (TGDs without criteria match)
Beispiel9_Terminierung _Andreas	example_9_Terminierung _Andreas	neu übersetzt
Beispiel10_Str	example_10_Str	neu übersetzt, (Stratifikationskriterium)
Beispiel11_Q _AktiveTriggerVC	example_11_Q _AktiveTriggerVC	neu übersetzt, Variation von example_11_Q_ActiveTriggerVE VC = allquantifizierte Variable auf Constant abbilden
Beispiel11_Str	example_11_Str	neu übersetzt, (Stratifikationskriterium)
Beispiel12_SR	example_12_SR	neu übersetzt, (Safe Restriction) Acyclicity Test terminiert nicht
Beispiel13_SwA		nicht übersetzt, (Superweak Acyclicity) ChaTEAU kann keinen SwA-Test
Beispiel14_LS	example_14_LS	neu übersetzt, (Local Stratification)
Beispiel15_IR	example_15_IR	neu übersetzt, (Inductive Restriction)
Beispiel16_EGD		nicht übersetzt, kein neuer Testfall
Beispiel17_Implication		nicht übersetzt, irrelevant
Beispiel18_Binaerbaum		nicht übersetzt, irrelevant
TestanfrageA1	benchmark_1	bereits vorhanden
TestanfrageA2	benchmark_2	bereits vorhanden
TestanfrageA3	benchmark_3	bereits vorhanden
TestanfrageAQuV	example_AQuV	bereits vorhanden
TestanfrageB1	benchmark_4	bereits vorhanden
TestanfrageB2	benchmark_5	bereits vorhanden
TestanfrageB3	benchmark_7	bereits vorhanden
Testanfrage_DH13	example_DH13	bereits vorhanden

12 Anleitung zur Benutzung von ChaTEAU

[Eduard,Lukas]

12.1 Installation und Ausführung

Zuerst muss der Quellcode runtergeladen werden per Klonen mit Git:

```
git clone https://git.informatik.uni-rostock.de/ta093/
chateau-ksws.git
```

(Hinweis: Dies ist die URL für das Git-Repository, auf dem wir gearbeitet haben. Das echte Repository von ChaTEAU hat eine andere URL.)

Nach dem Runterladen des Quellcodes muss dieser mit Hilfe von Maven kompiliert werden. Hierfür muss Java 11 und Apache Maven auf dem System installiert sein. Danach muss die Kommandozeile geöffnet und mit Hilfe von `cd` zum Wurzelverzeichnis des Projekts navigiert werden. Dann ist folgender Befehl auszuführen:

```
mvn clean compile resources:resources assembly:single
```

Der Befehl sollte mit „BUILD SUCCESS“ abschließen. Danach befindet sich im `target` Ordner eine Datei namens `chateau-(version)-jar-with-dependencies.jar`, wobei `(version)` die Versionsnummer von ChaTEAU ist. Diese Datei sollte man mit einem Doppelklick ausführen können. Unter Linux muss diese vorher in den Eigenschaften als ausführbar markiert werden. Alternativ kann sie auch wie folgt in der gleichen Kommandozeile ausgeführt werden:

```
cd target
java -jar chateau-(version)-jar-with-dependencies.jar
```

Dabei muss `(version)` durch die entsprechende Versionsnummer ersetzt werden. Danach öffnet sich ChaTEAU wie in Abbildung 3a.

12.2 Benutzung

Das typische Anwendungsszenario von ChaTEAU beinhaltet vier Schritte. Diese sind in der GUI durch die vier Tabs am oberen Rand repräsentiert. Durch Klicken gelangt man zu dem jeweiligen Tab. Alternativ gibt es auch die Knöpfe `Previous Step` und `Next Step` am unteren Rand, die man zur Navigation verwenden kann.

Im ersten Tab namens „Start“ kann jetzt eine Datei eingelesen werden, indem man auf den Knopf `Open ChaTEAU file` drückt, im Auswahlfenster zum Speicherort der Datei navigiert und diese öffnet. Hier wird als Beispiel die Datei `example_1_I_TGD.xml` benutzt. Nach den Öffnen sieht man die Inhalte der Datei lesbar dargestellt wie in Abbildung 3b.

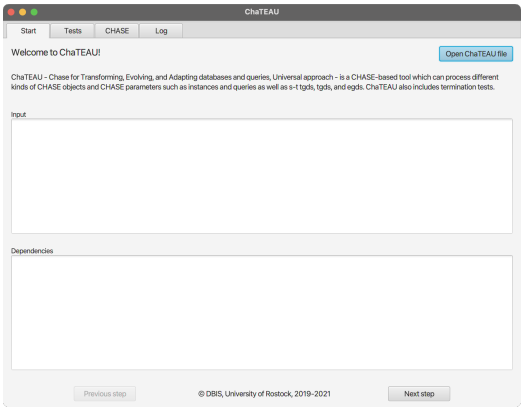
Im nächsten Schritt können verschiedene Terminierungstests im „Tests“-Tab durchgeführt werden, um zu erfahren, ob der Chase-Algorithmus möglicherweise nicht terminiert. Standardmäßig sind hier alle Tests ausgewählt. Ausführen kann man diese Tests mit dem Knopf `Run selected tests` oben rechts. Das mindestens einmalige Ausführen dieser

Tests ist für die Anwendung des Chase-Algorithmus im nächsten Schritt notwendig. Das Ergebnis sieht aus wie in Abbildung 3c.

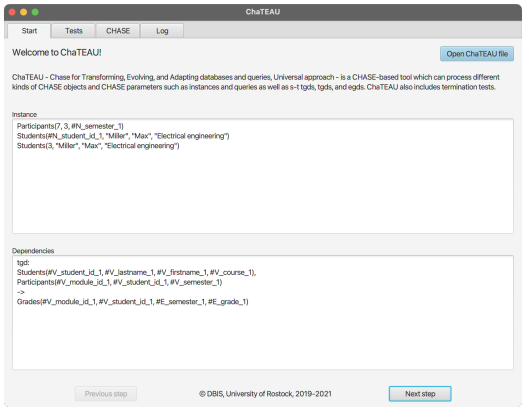
Der dritte Schritt ist die Ausführung des Chase-Algorithmus im „CHASE“-Tab. Hierzu drückt man oben rechts auf den Knopf `Run CHASE`. Falls einer der Terminierungstests besagte, dass der Chase-Algorithmus vielleicht nicht terminiert, erhält man jetzt ein Pop-Up (siehe Abbildung 4b), was darauf hinweist und fragt, ob man dennoch einen Durchlauf machen will. Tut man dies, wird nach dem Beginn der Ausführung der `Run CHASE` Knopf ersetzt durch einen `Stop CHASE` Knopf, mit dem man die Ausführung abbrechen und das bis dahin errechnete Ergebnis anzeigen lassen kann. Das Ergebnis wird in der untersten Textbox angezeigt, wie in Abbildung 3d ersichtlich. Man kann das Ergebnis durch den `Save result` Knopf oben rechts in einer Textdatei speichern.

Zuletzt kann man sich im „Log“-Tab (siehe Abbildung 4a) den Log von ChaTEAU angucken, in dem alles Gemachte protokolliert wurde. Diesen kann man durch `Save log` auch in einer Textdatei speichern.

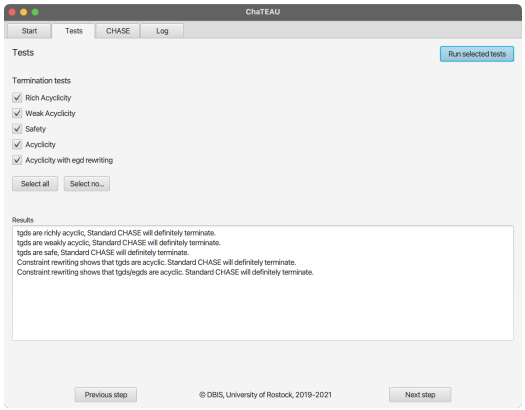
Ausführung des Beispiels in Bildern



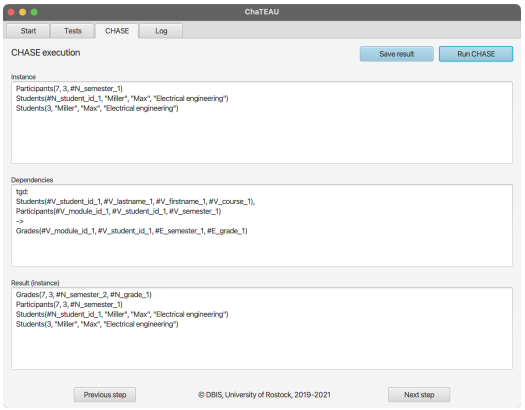
(a) Start des Programms



(b) Einlesen der XML Datei

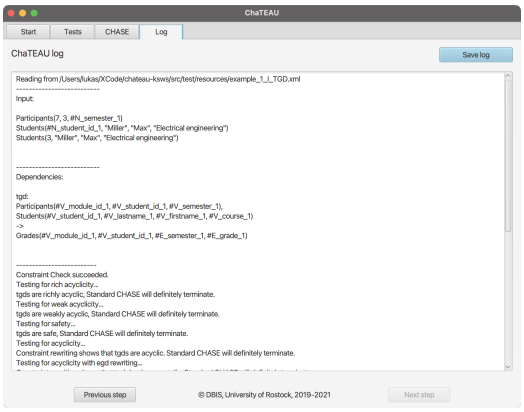


(c) Terminierungstests

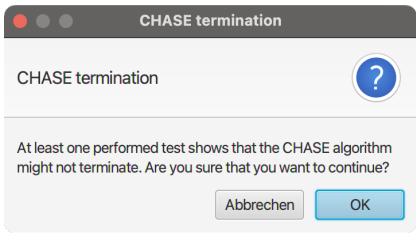


(d) Chase-Ergebnis

Abbildung 3: Durchlauf des Beispiels



(a) Log Ausgabe



(b) Popup bei fehlgeschlagenen Terminierungstest(s)

Abbildung 4: Log und Fehlermeldung

13 Cleanup - Durchsicht

[Eric, Michael]

Die Namen von Variablen und Methoden sollen als implizite Dokumentation des Codes dienen und sind für dessen Verständnis wesentlich. Die Wahl geeigneter Namen ist schwierig und es gibt unterschiedliche Ansichten über Regeln zu diesem Thema, von alten Traditionen ohne Begründung („das macht man eben so“) bis hin zu ausgefeilten Konzepten. Diese innere Dokumentation wird durch Kommentare und weitere Konzepte erweitert. Im Rahmen des Cleanups haben wir uns damit beschäftigt.

Für ein Projekt wie ChaTEAU, welches über mehrere Jahre und durch verschiedene Entwickler entstand, sollte sauberer Quellcode unter keinen Umständen vernachlässigt werden. Bei fast 10.000 Zeilen an Code nimmt die Einarbeitung neuer Entwickler viel Zeit in Anspruch. Um diesen Einstieg in der Zukunft leichter zu gestalten, wurde das Programm auf verschiedene Qualitätsmerkmale eines „Clean Codes“ überprüft. Dazu haben wir uns hauptsächlich auf die Verständlichkeit des Codes konzentiert und dabei folgende Punkte bearbeitet:

Variablenbenennung: Variablen sollten sinnvoll benannt werden, damit ersichtlich ist, zu welchem Zweck diese existieren. In ChaTEAU werden Variablen oft durch 'var' statt einer expliziten Typdefinition wie 'boolean' initialisiert, wobei die sinnvolle Benennung der Variablen umso wichtiger wird. Bei richtiger Anwendung kann dadurch die Lesbarkeit verbessert werden, ohne wichtige Informationen zu verlieren.
Ein Programmabschnitt aus ChaTEAU zeigt, wie 'var' benutzt wird, um eine Reihe von Deklarationen übersichtlicher zu gestalten:

```
var success = true;
var attributes = instance.getAttributes();
var schema = instance.getSchema();
var originTag = instance.getOriginTag();
```

Im Vergleich dazu die äquivalente Implementation ohne 'var':

```
boolean success = true;
Set<String> attributes = instance.getAttributes();
Map<String, List<String> > schema = instance.getSchema();
OriginTag originTag = instance.getOriginTag();
```

In diesem Fall sieht man schnell, dass die Variable `success` vom Typ `boolean` ist, denn sie erhält den Wert `true`. Ebenso ist ein dreifaches Vorkommen von `OriginTag` innerhalb einer Zeile nicht notwendig. Für die Variablen
`attributes` und `schema` ist die Typdefinition für das Verständnis der allgemeinen Funktionsweise von Bedeutung. Sollte doch einmal der Typ von Interesse sein, ist dieser bei fast allen Programmierumgebungen durch einen Klick einsehbar.

Methodenbenennung: Auch Methoden sollten aussagekräftig benannt werden, sodass anhand des Namens schon erkenntlich ist, wofür diese zuständig sind. Dabei gilt das Prinzip: Lieber ein Wort zu viel als zu wenig. Beispielsweise unterschieden sich in ChaTEAU zwei Methoden(`processTrigger` und `processTriggers`) nur um einen Buchstaben. Dies kann zu Missverständnissen bei der Einarbeitung führen, da diese Methoden auf den ersten Blick nur schwer zu unterscheiden sind. Daraufhin entschieden wir uns dazu, die Methode `processTriggers` zu `processTriggersLoop` umzubenennen.

Kommentare und Javadoc: Zwei stammverwandte Möglichkeiten, Quellcode gut zu dokumentieren, sind Kommentare und Javadoc. Ohne Kommentare ist die Einarbeitung in ein Programm für einen Programmierer sehr schwierig bis nahezu unmöglich.

Kommentare haben als Konzept verschiedene Probleme[6]:

- Schlechte Wartbarkeit: Sie werden bei Veränderungen des Codes kaum jemals angepasst; aber gerade Kommentare, die nicht mehr gültig sind, verursachen erhebliche Probleme bei der Einarbeitung in Code, da der Leser sich im Allgemeinen darauf verlässt.
- Umfang: Es ist prinzipiell schwer, den allgemein sinnvollen Umfang von Kommentaren festzulegen, allzu oft wird im Übermaß oder zu spärlich kommentiert.
- Kommentare werden oft gemacht, weil man es muss; sie sind bei Programmierern eher unbeliebt, da man selbst seinen Code versteht und „alles klar verständlich“ erscheint. Das führt neben zu wenigen Kommentaren zu schlechter Qualität derer, wie zum Beispiel mangelnder Aussagekraft oder deren Auftauchen an ungünstigen Stellen im Code.

Wir haben alle Kommentare hinsichtlich der o.a. Punkte evaluiert, Fehler berichtigt und uns auch über eine teilweise sehr schöne Kommentarkultur gefreut.

Mit **Javadoc** können automatisch Dokumentationen aus dem Java-Code erzeugt werden. Das einfache Prinzip: bestimmte Markierungen(„Tags“) machen ggf. ohnehin vorhandene Kommentare zu Javadoc-Elementen, welche dann von einem Javadoc-Parser ausgelesen und zu einer HTML-Datei zusammengesetzt werden können, welche eine sehr ausführliche und

[6] zusammengefasst nach[4]

standardisierte Dokumentation eines Quelltextes ist, in der en detail die einzelnen Methoden mitsamt Rückgabewerten und Parametern protokolliert werden.

Von den ursprünglichen Programmierern wurde Javadoc schon weitestgehend eingesetzt, so dass uns nur wenige Stellen nachzufeilen blieben.

14 Logger

[Michael, Eric]

ChaTEAU verfügt über eine Logging-Funktion. Hier werden auf einem eigenen Tab die einzelnen Schritte der Verarbeitung einer Eingabedatei protokolliert. Im Einzelnen bestand das Log vor den von uns vorgenommenen Änderungen aus dem Name der Eingabedatei, die Durchführung und Ergebnisse der einzelnen Terminierungstests, die Vorbereitung und Verarbeitung der Eingabe durch die Implementierung des Chase mit Auflistung der aktiven Trigger und der angewendeten Mappings. Es gibt keine Zeitangaben. Das Log des Log-Tabs kann mit einem Button in eine Textdatei gespeichert werden. Eine Überprüfung dieser Funktionalität ergab, dass nicht das gesamte angezeigte Log in die erzeugte Textdatei geschrieben wurde. Weiterhin entstand die Idee, dass in einem vollständigen Log auch die zweiteilige Eingabe und das Resultat der Berechnung sinnvoll wäre.

Die Implementierung des Logs ist eine Eigenentwicklung. Kurzzeitig haben wir die Idee verfolgt, eine vollständige, industrielle Standardlösung - beispielsweise den Logger von Apache Commons - einzubauen. Diese hätte sogar die Möglichkeit der Protokollierung in verschiedener Detailtiefe(Loglevel) ermöglicht. Letztendlich sprach die Tatsache, dass ChaTEAU immer in einer beaufsichtigten und kontrollierten Umgebung immer nur kurzzeitig und ohne unkontrollierte Interaktion beliebiger Nutzer läuft, gegen ein exzessives Logging, wie man es zum Beispiel bei einem Webserver macht. Die Möglichkeit verschiedener Loglevel wurde dahingehend überlegt, als dass ein Log nur mit den Eingaben und der Ausgabe, sowie ein vollständiges Log umschaltbar hätte implementiert werden können. Auf dem Chase-Tab gibt es aber schon eine sehr sinnvoll gebaute Ausgabe der Eingaben und Ergebnisse eines Programmdurchlaufes mit einer gut funktionierenden Exportfunktion. Hier hätte sich also mit der vorigen Idee der Nutzung von zwei Logleveln kein Mehrwert ergeben.

Somit haben wir die Logging-Funktionalität dahingehend verändert, dass nun im vollständigen Log auch die Eingaben und das Resultat in gewohnter Form enthalten sind, sodass die Loglevel implizit vorhanden sind. Wir haben uns dabei um eine - in den Grenzen von reinen Textdateien mögliche - gute Lesbarkeit bemüht.

Der Export enthält nun das gesamte angezeigte Log.

Auch bei der Benutzung des Programms ohne die Nutzeroberfläche wird das Log mit Eingabe und Ergebnis auf der Kommandozeile ausgegeben. Hier lässt sich aufgrund des prinzipiellen Mangels an Eingabemöglichkeiten keine Exportmöglichkeit einbauen.

15 Offene Baustellen

[Moritz,Lukas]

`Tester.java` Die Methoden zur Durchführung der verschiedenen Terminierungstests in `Tester.java` sind doppelt negativ benannt, was unnötige Verwirrung stiftet. Zum Bei-

spiel liefert `checkWeakAcyclicity(...) true`, wenn die Integritätsbedingungen *nicht* schwach azyklisch sind.

`ConstraintRewriting.java`

Die Klasse `ConstraintRewriting.java` hat nur ein einziges Feld `graph`, welches der einzige Grund dafür ist, weshalb nicht alle Methoden der Klasse `static` sein können. Hier wäre es stylistisch eleganter, gar nicht erst eine Instanz der Klasse `ConstraintRewriting` erzeugen zu müssen, sondern den Konstruktor `private` zu halten und nur noch statische Funktionen anzubieten. Zurzeit werden erzeugte Instanzen sowieso nur einmalig verwendet, daher kann die Erzeugung des Felds `graph` genauso gut in die (statisch sein sollende) Methode `prepareAdn` und/oder `prepareConstraintAdn(...)` verschoben werden. So werden auch gleich Fehler wie versehentlich gespeicherte Zustände in Instanzattributen wie in z.B. Issue #17 vermieden.

`example_12_SR.xml`

Der Test auf Azyklizität ohne EGD-Umschreiben terminiert nicht. Der Test auf Azyklizität mit EGD-Umschreiben wirft eine Exception, welche nicht näher bestimmbar ist.

16 Fazit und Ausblick

[Moritz]

Fazit Der Großteil der gefundenen Mängel und Verbesserungen betreffen das Eingabeformat des XML. Hier konnte das Einlesen von XMLs insbesondere durch Nutzung einer DTD robuster gemacht und aussagekräftigere Fehlermeldungen geliefert werden, wenn mit der Eingabe irgendetwas nicht stimmt.
An funktionalen Fehlern wurde nur die Kleinigkeit in der Komponente für die Terminierungstests gefunden, behoben und weitere falsche Benutzung dieser Komponente durch Umstrukturierung des Design-Patterns verhindert.

Der Code war schon recht gut dokumentiert, es gab korrekte (wenn auch etwas veraltete) Klassen- und Flussdiagramme, von denen jeweils aktuellere Versionen erstellt wurden. Mit unserer Anleitung zur Benutzung von ChaTEAU sollte der Einstieg in das Projekt nun etwas vereinfacht sein. An anderen Stellen führte eine unzureichende Dokumentation allerdings dazu, dass veraltete Programmteile ohne ersichtlichen Grund nicht funktionierten. An diesen Stellen haben wir entsprechend nachgebessert.

Ebenso konnten wir nur wenige Programmierpraktiken unter dem Gesichtspunkt *Clean Code* bemängeln Diesbezüglich haben hier die Erweiterbarkeit verbessert und so die fortdauernde Arbeit am Projekt ein wenig vereinfacht.

Erwähnenswert sind auch die zahlreichen neu erstellten XML-Dateien und zugehörigen Unit-Tests, die bei der Weiterentwicklung von ChaTEAU mehr Fehlerquellen automatisiert erkennen sollten.

Ansonsten können wir ChaTEAU einen durchaus soliden Zustand bescheinigen.

Ausblick ChaTEAU soll noch um zahlreiche Funktionen erweitert werden. Hierzu zählt zum Beispiel die Umformung einer Anfrage, sodass ausschließlich eine gegebene Menge an Sichten für ihre Beantwortung genutzt wird (*Answering Queries using Views*) und eine

Kopplung von ChaTEAU mit dem Projekt ProSA mitsamt seiner neuen GUI und seinem neuen SQL-Parser, welche ebenfalls in diesem KSWS-Seminar neu implementiert wurden. Gerade für diese letzte Aufgabe, welche eine gute Erweiterbarkeit von ChaTEAU voraussetzt, sollte unser Beitrag eine signifikante Arbeitserleichterung darstellen.

Eine Übersicht über weitere für ChaTEAU ausgeschriebene Abschlussarbeiten findet sich hier, es gibt also noch eine Menge zu tun und wir hoffen mit unserem Beitrag den Einstieg in das Projekt merklich erleichtert zu haben.

Literatur

[1] Dietmar Abts. *Grundkurs Java*. Wiesbaden: Springer Verlag, 2020.

[2] Andreas Oliver Görres. „Erweiterung des CHASE-Werkzeugs ChaTEAU um ein Terminierungskriterium“. In: (2020). URL: http://eprints.dbis.informatik.uni-rostock.de/1009/1/Goerres_Masterthesis_03-03-2020_final.pdf.

[3] Martin Jurklies. „CHASE und BACKCHASE:Entwicklung eines Universal-Werkzeugs für eine Basistechnik der Datenbankforschung“. In: (2018). URL: http://eprints.dbis.informatik.uni-rostock.de/1009/1/Goerres_Masterthesis_03-03-2020_final.pdf.

[4] Robert Martin. *Clean Code*. Bonn: MITP Verlag, 2009.

[5] Florian Rose. „Erweiterung des CHASE-Werkzeugs ChaTEAU um eine BACKCHASE-Phase“. In: (2020). URL: http://eprints.dbis.informatik.uni-rostock.de/1032/1/Masterarbeit_Florian_Rose_v2.pdf.

[6] Nic Scharlau. „Der CHASE-Algorithmus: Ein Überblick“. In: (2021).

[7] Jakob Zimmer. „Vereinheitlichung des CHASE auf Instanzen und Anfragen am Beispiel ChaTEAU“. In: (2019). URL: http://eprints.dbis.informatik.uni-rostock.de/1008/1/Bachelorarbeit_Jakob_Zimmer.pdf.